The Point Code of a $(22, 33, 12, 8, 4)$-Balanced Incomplete Block Design

by

Richard T. Bilous

A thesis
presented to the University of Manitoba
in partial fulfilment of the
requirements for the degree of
Ph.d.
in
Computer Science

Winnipeg, Manitoba, Canada, 2001

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

The Point Code of a (22, 33, 12, 8, 4)- Balanced Incomplete Block Design

BY

Richard T. Bilous

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree of Doctor of Philosophy

Abstract

A (v, b, r, k, λ)-balanced incomplete block design (or simply a BIBD) is a family of b sets, called blocks, each consisting of k elements taken from a set of v elements, called varieties, such that every variety occurs in exactly r blocks and every pair of varieties occur together in exactly λ blocks. The incidence matrix of a (v, b, r, k, λ)-BIBD is a $v \times b$ binary matrix A whose rows are indexed by the varieties, typically 1 to v, and whose columns are indexed by the block names, typically 1 to b. Entry $a_{i,j}$ of A contains a 1 if variety i is in block j, otherwise entry $a_{i,j}$ contains a 0.

There are several well-known necessary conditions for the existence of a BIBD with parameters (v, b, r, k, λ). However, these conditions are not sufficient. The parameters with the smallest v that obeys the conditions for which it is not known whether or not a BIBD exists is $(22, 33, 12, 8, 4)$. The problem we will be investigating in this thesis is "does a $(22, 33, 12, 8, 4)$-BIBD exist?" This has been, and remains, an open problem for over 60 years.

Our approach to this problem is based on the fact that if a $(22, 33, 12, 8, 4)$-BIBD exists, then so does its point code. The point code of a (v, b, r, k, λ)-BIBD B is the subspace of $V_b(2)$ that is determined by the span of the rows of the incidence matrix of B. It is known that the point code of a $(22, 33, 12, 8, 4)$-BIBD is a length 33 doubly-even self-orthogonal code over $GF(2)$.

Our method for investigating whether or not a $(22, 33, 12, 8, 4)$-BIBD exists works as follows: First, we find a list L of length 33 doubly-even self-orthogonal codes with the property that a $(22, 33, 12, 8, 4)$-BIBD exists if and only if L contains a code that contains the incidence matrix of such a design. We then try to determine, for each code C in L, whether or not C contains the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. If none of the codes in L contains such an incidence matrix, then we know a $(22, 33, 12, 8, 4)$-BIBD does not exist.

In this thesis, we will prove that any complete list L of inequivalent $(33, 16)$ doubly-even self-orthogonal codes over $GF(2)$, that do not contain a coordinate of zeros, has the property that a $(22, 33, 12, 8, 4)$-BIBD exists if and only if L contains a code that contains the incidence matrix of such a design. We have enumerated such a list L of inequivalent $(33, 16)$ doubly-even self-orthogonal codes. We have also found the automorphism group of each code in L. The number of codes in L is 594.

Our problem is now to determine whether or not there exists a code in L that contains the incidence matrix of such a design. Thus far, we have been able to theoretically prove that 116 of the codes in L cannot contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. For the 478 remaining codes, we have written a computer program to search for an incidence matrix in each code. At the time of writing, we have used our program to search 155 of the 478 remaining codes, none of which contained an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD.

Included in the codes in L that we have eliminated, both through theory and with our search algorithm, are all the codes that contain three or more weight 4 words that all have a value of 1 in a particular coordinate. This in turn tells us that the point code of a $(22, 33, 12, 8, 4)$-BIBD cannot contain three such weight 4 words.

We are now left with 323 undecided codes in L. Therefore, it is still not known whether or not a $(22, 33, 12, 8, 4)$-BIBD exists. However, we do believe our computer program demonstrates that it may not be infeasible to search these remaining codes.

Acknowledgements

I would like to thank my supervisor Dr John van Rees for his advice and support throughout the development of this thesis. I would also like to thank the members of my committee, Dr Edward Shwedyk, Dr John Bate, and Dr Clement Lam, for taking the time to read this thesis. I am also thankful to Malcolm Greig for reading this thesis and making many helpful suggestions.

For financial support, I would like to thank to Natural Sciences and Engineering Research Council of Canada and the University of Manitoba.

Contents

Chapter 1

Introduction

In this chapter, we will discuss the problem we will be investigating in this thesis: "does a $(22, 33, 12, 8, 4)$-balanced incomplete block design exist?" This has been, and remains, an open problem for over 60 years.

In Section 1.1, we will give some of the basic coding and block theory definitions and notations that we will use throughout the thesis. In Section 1.2, we will give a formal definition of the problem: "does a $(22, 33, 12, 8, 4)$-balanced incomplete block design exist?" In Section 1.3, we will discuss some of the work that has previously been done on this problem. In Section 1.4, we will discuss the approach we will use to investigate this problem. Finally, in Section 1.5, a brief description of the contents of each of the remaining chapters in the thesis is given.

1.1 Definitions And Notation

In this section, we will discuss the basic coding and design theory we will need. For a more detailed discussion on coding theory see [13]. For a more detailed discussion on design theory see [1].

1.1.1 Linear Codes

Let $GF(m)$ denote the field of m elements. Let $V_n(m)$ denote the vector space of all n-vectors over $GF(m)$. An (n, k) *linear code* C over $GF(m)$ is a k dimensional subspace of $V_n(m)$. The integer n is called the *length* of C. The integer k is called the *dimension* of C. The n-vectors in C are called *codewords*.

In this thesis we will only be working with the linear codes over $GF(2)$,which are also known as the *binary linear codes*. Unless stated otherwise, whenever we use the term *code* we are referring to the linear codes over $GF(2)$.

Let $\vec{u}$ and $\vec{v}$ be vectors in $V_n(2)$. We use the notation $\vec{u} + \vec{v}$ to denote the vector whose i^{th} component is equal to the sum in $GF(2)$ of the i^{th} components of $\vec{u}$ and $\vec{v}$. It is well known that a set $C \subseteq V_n(2)$ is a linear code if and only if for all $\vec{u}, \vec{v} \in C$, we have $\vec{u} + \vec{v} \in C$.

Example 1.1. Consider the following set of eight 6-vectors over $GF(2)$:

$$C = \left\{ \begin{array}{cccc} (0,0,0,0,0,0) & (0,0,0,0,1,1) & (0,0,1,1,0,0) & (1,1,0,0,0,0) \\ (0,0,1,1,1,1) & (1,1,0,0,1,1) & (1,1,1,1,0,0) & (1,1,1,1,1,1) \end{array} \right\}$$

For any pair of vectors $\vec{u}$ and $\vec{v}$ in C, we have $\vec{u} + \vec{v} \in C$. Therefore C is a linear code. The length of C is 6. The vectors $(0,0,0,0,1,1)$, $(0,0,1,1,0,0)$, and $(1,1,0,0,0,0)$ form a basis for C, which implies the dimension of C is 3. Thus, C is a $(6,3)$ code.

Let C be an (n, k) code. The number of codewords in C is 2^k. We will usually represent the codewords in C as binary strings of length n. We will usually represent C with a *generator matrix*. A generator matrix G is any $k \times n$ binary matrix whose rows form a basis for C.

Example 1.2. Let C be the $(6,3)$ code of Example 1.1. The codewords 000011, 001100,

and 110000 form a basis for C. Therefore, the matrix:

$$G = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

is a generator matrix for C.

Let C be an (n, k) code, let $\vec{c}$ be a codeword in C, and let G be a generator matrix for C. Then there is a one-to-one correspondence between the coordinates of C, the components of $\vec{c}$, and the columns of G. Throughout this thesis, we will implicitly make use of this correspondence. For example, if we begin a discussion with a phrase such as "coordinate j of C," and then later use the phrase "column j of G," then we are referring to the column in G that corresponds to coordinate j of C.

Let C be an (n, k) code. The *weight* of a codeword $\vec{c} \in C$, written $w(\vec{c})$, is the number of non-zero components in $\vec{c}$. For the binary codes, $w(\vec{c})$ is just the number of ones in $\vec{c}$. The *weight distribution* of C is a count of the number of codewords in C with weight w, for $w = 0, 1, \ldots, n$. We typically represent the weight distribution of C by the sequence $(A_0, A_1, \ldots, A_n)$, where A_w is the number of weight w codewords in C. The *distance* between two codewords is the number of positions in which they differ. The *distance* of a code C is the smallest distance between any two distinct codewords. An (n, k, d) code is an (n, k) code with distance d. It is well known that the distance of a linear code C is equal to the smallest weight of any non-zero codeword in C.

Example 1.3. Let C be the $(6, 3)$ code of Example 1.1. The codewords in C are:

$$\vec{c}_1 = 000000 \quad \vec{c}_2 = 000011 \quad \vec{c}_3 = 001100 \quad \vec{c}_4 = 110000$$
$$\vec{c}_5 = 001111 \quad \vec{c}_6 = 110011 \quad \vec{c}_7 = 111100 \quad \vec{c}_8 = 111111$$

The weight of $\vec{c}_1$ is 0, the weight of $\vec{c}_2$, $\vec{c}_3$, and $\vec{c}_4$ is 2, the weight of $\vec{c}_5$, $\vec{c}_6$, and $\vec{c}_7$ is 4, and the weight of $\vec{c}_8$ is 6. The weight distribution of C is $(1, 0, 3, 0, 3, 0, 1)$. The distance of C is 2.

An (n, k) code C is a *composed code* if we can partition the coordinates of C into two sets, S_1 and S_2, such that: (1) the codewords in C that have a value of 0 in every coordinate in S_2 form an (n_1, k_1) code C_1, and (2) the codewords in C that have a value of 0 in every coordinate in S_1 form an (n_2, k_2) code C_2, where $n_1 = |S_1|$, $n_2 = |S_2|$, and $k_1 + k_2 = k$.

1.1.2 Permutation Conventions And Notation

Let M denote an ordered collection of n objects such as the rows of an $n \times m$ matrix or the coordinates of an (n, k) code. A *permutation* of M is a rearrangement of the objects in M.

The conventions and notation we use to carry out permutations of the objects in M are as follows: We label the n objects in M with the integers $1, 2, \ldots, n$. Let π be a permutation of the integers $1, 2, \ldots, n$. We use the notation πM to denote the collection of objects that results from applying the permutation π to the objects of M. That is, if:

$$\pi = \begin{pmatrix} 1 & 2 & 3 & \cdots & n \\ a_1 & a_2 & a_3 & \cdots & a_n \end{pmatrix},$$

then the i^{th} object in πM is object a_i in M, for $i = 1, 2, \ldots, n$. For example, if M are the rows of an $n \times m$ matrix, then πM is the $n \times m$ matrix whose i^{th} row is row a_i of M.

Products of permutations are carried out from right to left. That is, if π_1 and π_2 are two permutations of M, then $\pi_1 \pi_2 M = \pi_1(\pi_2 M)$.

1.1.3 Equivalent Codes And Automorphisms

Let C_1 and C_2 be (n, k) codes. Then C_1 and C_2 are said to be *equivalent* if there exists a permutation π of the coordinates of C_1 that takes C_1 into C_2. That is, C_1 and C_2 are equivalent if there exists a permutation $\pi = \begin{pmatrix} 1 & 2 & 3 & \cdots & n \\ a_1 & a_2 & a_3 & \cdots & a_n \end{pmatrix}$ such that for any

codeword $(u_1, u_2, \ldots, u_n)$ in C_1 we have $(u_{a_1}, u_{a_2}, \ldots, u_{a_n})$ is a codeword of C_2. If no such permutation exists then C_1 and C_2 are said to be *inequivalent.*

The equivalence of codes is an equivalence relation. That is, for linear codes C_1, C_2, and C_3, we have:

1. C_1 is equivalent to C_1,
2. if C_1 is equivalent to C_2 then C_2 is equivalent to C_1, and
3. if C_1 is equivalent to C_2 and C_2 is equivalent to C_3 then C_1 is equivalent to C_3.

Since the equivalence of codes is an equivalence relation, we can partition the set $\mathcal{C}(n,k)$ of all (n,k) codes into *equivalence classes.* That is, we can partition $\mathcal{C}(n,k)$ into a finite number of disjoint sets $\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \ldots$ with the property that two codes are elements of the same set if and only if they are equivalent.

An *automorphism* of a code C is a permutation of the coordinates of C that takes C into itself. The set of all automorphisms of C is called the *automorphism group* of C. As the name suggests, the automorphism group of C is a group. We use the notation $\mathcal{AUT}(C)$ to denote the automorphism group of C. We typically represent $\mathcal{AUT}(C)$ with a set of *generators* for the group. That is, a set Π of coordinate permutations with the property that $\pi \in \mathcal{AUT}(C)$ if and only if π can be written as the product of one or more (not necessarily distinct) permutations in Π.

1.1.4 Self-Orthogonal Codes

Let $\vec{u} = (u_1, u_2, \ldots, u_n)$ and $\vec{v} = (v_1, v_2, \ldots, v_n)$ be any two vectors in $V_n(2)$. The *dot product* of $\vec{u}$ and $\vec{v}$, written $\vec{u} \bullet \vec{v}$, is defined as $(\sum_{i=1}^{n} u_i v_i) \bmod 2$. The commutative and distributive laws hold for the dot product of vectors. That is, $\vec{u} \bullet \vec{v} = \vec{v} \bullet \vec{u}$, and $\vec{u} \bullet (\vec{v} + \vec{w}) = (\vec{u} \bullet \vec{v}) + (\vec{u} \bullet \vec{w})$.

The vectors $\vec{u}$ and $\vec{v}$ are called *orthogonal* if $\vec{u} \bullet \vec{v} = 0$. The distributive law can be used to show that if a vector $\vec{v}$ is orthogonal to each vector in a set S of vectors in $V_n(2)$, then $\vec{v}$ is orthogonal to all linear combinations of the vectors in S.

Let C be an (n, k) code. The *orthogonal complement* of C, written $C^{\perp}$, is the set of all vectors in $V_n(2)$ that are orthogonal to every codeword in C. It is well known that $C^{\perp}$ is an $(n, n-k)$ code called the *dual code* of C.

An (n, k) code C is called *self-orthogonal* if $C \subseteq C^{\perp}$. A self-orthogonal code C is called *self-dual* if $C = C^{\perp}$. Since $C^{\perp}$ has dimension $n-k$, if C is self-orthogonal then we must have $k \leq n/2$, and if C is self-dual then we must have $k = n/2$.

Example 1.4. Let C be the $(7, 3)$ code generated by:

$$G = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

The orthogonal complement of C is the $(7, 4)$ code generated by:

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Since the first three rows in H are the rows in G, we know $C \subset C^{\perp}$. Therefore, C is a self-orthogonal code.

If C is a self-orthogonal code then, since every vector in C must be orthogonal to itself, the codewords in C all have even weight. Therefore, if C is a self-orthogonal code then we know $\vec{1} \in C^{\perp}$. If C is also a self-dual code then we must have $\vec{1} \in C$. If every codeword in a self-orthogonal code C has weight $\equiv 0 \bmod 4$ then C is called *doubly-even*.

1.1.5 Column Supports

Let C be an (n, k) linear code. Let W be any set of n-vectors. The *support* of W in C is the set S of all coordinates j in C in which there exists at least one vector in W that has a value of 1 in coordinate j. We refer to S as the coordinates in C that are *supported* by W.

Example 1.5. Let C be any $(10, 4)$ code, and let

$$W = \left\{ \begin{array}{cccccccccc} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{array} \right\}.$$

Then the first six coordinates in C are the coordinates that are supported by W.

If A is a matrix whose rows are codewords in C then we use the phrase "the columns in A that are supported by W" to refer to the columns in A that correspond to the set of coordinates S.

1.1.6 Strings of 1's and 0's

In this thesis, we will be dealing with many structures that can be represented by a string of 1's and 0's, such as binary vectors and rows and columns in a binary matrix. Therefore, there is much terminology that is shared by these similar structures. In this subsection, we will describe the terminology that we will use for any structure that can be represented by a string of 1's and 0's.

Let S be a string of 1's and 0's. The *length* of S is the number of 1's and 0's in S. The *weight* of S is the number of 1's in S. Let S_1 and S_2 be any two strings of 1's and 0's that have equal length. We will use the phrase "S_1 intersects S_2 in m positions" to mean there are exactly m positions in S_1 and S_2 in which both have a value of 1. We will use the phrase "S_1 is greater than S_2" to mean that, from left to right, if p is the

first position in which S_1 and S_2 differ, then S_1 has a value of 1 in position p. In other words, S_1 is greater that S_2 if, when considered as integers in base 2, S_1 has a greater value than S_2. Similarly, we will use the phrase "S_1 is less than S_2."

Let M be an $m \times n$ binary matrix. We will use the phrase "the rows of M are in decreasing order" to mean that row i of M is greater than or equal to row $i+1$ of M, for $i = 1, 2, \ldots, m-1$. We will similarly use the phrase "the rows of M are in increasing order." Let M_1 and M_2 be any two $m \times n$ binary matrices. We will use the phrase "M_1 is less than M_2" to mean that, from top to bottom, if row i is the first row in which M_1 and M_2 differ, then row i of M_1 is less than row i of M_2. Similarly, we will use the phrase "M_1 is greater than M_2."

1.1.7 Balanced Incomplete Block Designs

A (v, b, r, k, λ)-*balanced incomplete block design*, or simply a BIBD, is a family of b sets, called *blocks*, each consisting of k elements taken from a set of v elements, called *varieties*, such that every variety occurs in exactly r blocks and every pair of varieties occurs together in exactly λ blocks. The integers (v, b, r, k, λ) are called the *parameters* of the design.

Example 1.6. Consider the following family of $b = 12$ blocks:

$$B = \left\{ \begin{array}{cccc} \{1,2,3\} & \{1,4,5\} & \{1,6,7\} & \{1,8,9\} \\ \{2,4,7\} & \{2,5,8\} & \{2,6,9\} & \{3,4,9\} \\ \{3,5,6\} & \{3,7,8\} & \{4,6,8\} & \{5,7,9\} \end{array} \right\}.$$

Each block consists of $k = 3$ varieties. The varieties are taken from the set $\{1, 2 \ldots, 9\}$ of $v = 9$ varieties. Each variety occurs in exactly $r = 4$ blocks. Each pair of varieties occurs together in exactly $\lambda = 1$ block. Therefore, B is a $(9, 12, 4, 3, 1)$-BIBD.

The *incidence matrix* of a (v, b, r, k, λ)-BIBD is a $v \times b$ binary matrix A whose rows are indexed by the varieties, typically 1 to v, and whose columns are indexed by the block

names, typically 1 to b. Entry $a_{i,j}$ of A contains a 1 if variety i is in block j, otherwise entry $a_{i,j}$ contains a 0. We will typically represent a BIBD by its incidence matrix.

Example 1.7. The matrix:

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

is the incidence matrix of the $(9, 12, 4, 3, 1)$-BIBD given in Example 1.6.

Let A be the incidence matrix of a (v, b, r, k, λ)-BIBD. Since each block in the design consists of k varieties, each column of A contains exactly k ones. Since each variety occurs in r blocks, each row of A contains exactly r ones. Since each pair of varieties occurs together in exactly λ blocks, for each pair of rows in A there are exactly λ columns in A in which both rows have a value of one.

Let B_1 and B_2 be any two (v, b, r, k, λ)-BIBDs. B_1 and B_2 are said to be *isomorphic* if there exist permutations π_v and π_b of the varieties and blocks, respectively, of B_1 that takes B_1 into B_2. Let A_1 and A_2 be the incidence matrices of B_1 and B_2, respectively. In terms of A_1 and A_2, the designs B_1 and B_2 are isomorphic if and only if there exist permutations of the rows and columns of A_1 that turn A_1 into A_2.

Let B be a (v, b, r, k, λ)-BIBD and let A be its incidence matrix. Suppose we consider the rows of A as binary b-vectors. Then the span of these rows forms a vector space C over $GF(2)$. C is a linear code, with length b, called the *point code* of B.

Let B_1 and B_2 be any two isomorphic (v, b, r, k, λ)-BIBDs and let A_1 and A_2 be their incidence matrices. Then since rows and columns of A_1 are simply permutations of the rows and columns of A_2, the point codes of B_1 and B_2 are equivalent. Similarly, if C_1 is a code that contains an incidence matrix A_1 for a BIBD B_1, then any code C_2 that is equivalent to C_1 will contain an incidence matrix A_2 for a BIBD B_2 that is isomorphic to B_1. In this context, the terms isomorphic and equivalent are essentially the same. However, design theorists tend to use the term isomorphic while code theorists tend to use the term equivalent.

1.2 The Problem

There are four well-known necessary conditions for the existence of a BIBD with parameters (v, b, r, k, λ). They are:

1. $rv = bk$,
2. $\lambda(v-1) = r(k-1)$,
3. Fisher's Inequality: $b \geq v$, and
4. the Bruck-Ryser-Chowla Condition: if $v = b$ then:
   (a) if v is even then $k - \lambda$ is a square, and
   (b) if v is odd then $z^2 = (k-\lambda)x^2 + (-1)^{(v-1)/2}\lambda y^2$ has a solution in the integers x, y, z not all equal to 0.

However, these conditions are not sufficient. The parameters with the smallest v that obey the above conditions for which it is not known whether or not a BIBD exists are $(22, 33, 12, 8, 4)$.

The problem we will investigate in this thesis is whether or not a $(22, 33, 12, 8, 4)$-BIBD exists. This has been a problem actively investigated for over 60 years.

1.3 History

In 1938, Fisher and Yates [6] produced tables listing all known small BIBDs. Their smallest undecided case in the number of varieties was for the parameters $(22, 33, 12, 8, 4)$. This case is still undecided today. In this section, we will review some of the research that has been done on the $(22, 33, 12, 8, 4)$-BIBD.

We begin with a result by Hamada and Kobayashi [9] that looks at the different ways in which the blocks of the design may intersect a given block. Suppose a $(22, 33, 12, 8, 4)$-BIBD B does exist. Let b_0 denote an arbitrary block in B. Let a_i denote the number of blocks that intersect block b_0 in i varieties, for $0 \leq i \leq 8$. That is, a_i is the number of blocks in B that contain i varieties that are also varieties in block b_0. Using simple counting arguments, Hamada and Kobayashi obtained the following equations:

$$\sum_{i=0}^{8} a_i = b - 1 = 32 \tag{1.1}$$

$$\sum_{i=1}^{8} i a_i = k(r-1) = 88 \tag{1.2}$$

$$\sum_{i=2}^{8} \binom{i}{2} a_i = \binom{k}{2}(\lambda - 1) = 84 \tag{1.3}$$

Equation 1.1 counts, in two different ways, the number of blocks in the design other than block b_0. Equation 1.2 counts, in two different ways, the number of times each of the varieties in block b_0 occur in the remaining blocks. Equation 1.3 counts, in two different ways, the number of times each pair of varieties in block b_0 occur in the remaining blocks.

Solving Equations 1.1–1.3, Hamada and Kobayashi found nine non-negative solutions. They were able to show the infeasibility of five of these solutions. The remaining four are known as block intersection patterns and are given in the following theorem:

Theorem 1.8. *The following are the only possible block intersection patterns for a BIBD*

with parameters $(22, 33, 12, 8, 4)$*:*

Type	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
1	0	0	12	16	4	0	0	0	0
2	0	1	9	19	3	0	0	0	0
3	0	2	6	22	2	0	0	0	0
4	1	0	6	24	1	0	0	0	0

Using extensive CPU time, McKay and Radziszowski [16, 17] have shown that neither the type 4 nor the type 3 block intersection patterns can occur. This gives us the following result:

Theorem 1.9. *The following are the only possible block intersection patterns for a BIBD with parameters* $(22, 33, 12, 8, 4)$*:*

Type	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
1	0	0	12	16	4	0	0	0	0
2	0	1	9	19	3	0	0	0	0

Let us now review the work that has been done on the point code of the design. Hall, Roth, van Rees, and Vanstone [10] were the first to consider the point code of a $(22, 33, 12, 8, 4)$-BIBD. Suppose a $(22, 33, 12, 8, 4)$-BIBD B exists. Let C be the point code of B. Since B contains 33 blocks, C is a linear code with length 33. Using the facts that the rows in the incidence matrix of B have weight $\equiv 0 \bmod 4$ and any pair of rows in the incidence matrix intersect in an even number of positions, Hall et al. showed that any codeword in C has weight $\equiv 0 \bmod 4$ and every pair of codewords in C intersects in an even number of positions. This gives us the following result:

Theorem 1.10. *Let C be the point code of a $(22, 33, 12, 8, 4)$-BIBD. Then C is a $(33, k)$ doubly-even self-orthogonal code.*

From [10, page 164], we know that $A_{12} = 13 \cdot 2^{k-6} - 10 - 6A_4 - 3A_8 - A_{24} - 3A_{28}$, where k is the dimension of the point code and A_i is the number of weight i words in the point code. Since the point code must contain at least 22 weight 12 words, this leads us to the inequality $13 \cdot 2^{k-6} - 10 \geq 22$, which implies $k \geq 8$. Therefore, the dimension k of the point code C is such that $8 \leq k \leq 16$. Van Rees [19] has also shown that C cannot be a composed code.

Let A be the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD B. Let C be the point code of B. Using the MacWilliams' identities [15], Hall et al. have found that the orthogonal complement $C^{\perp}$ of C contains at least 6 weight 5 words. Let $\vec{w}$ be a weight 5 word in $C^{\perp}$. Let A_5 denote the 22×5 matrix consisting of the 5 columns in A that are supported by $\vec{w}$. Then the rows of A_5 have even weight and the columns of A_5 have weight 8. Using these facts, Hall et al. proved there are 108 possibilities for A_5, unique up to row and column rearrangements. After much work they were able to reduce these possibilities to 13. Greig [8] has since reduced this number to 7. Greig also considered the cases of $C^{\perp}$ containing weight 3 words and $C^{\perp}$ containing weight 4 words.

Suppose $C^{\perp}$ contains a weight i word $\vec{w}$. Let A_i denote the $22 \times i$ matrix consisting of the i columns in A that are supported by $\vec{w}$. Let $S_i = A_i^T A_i$. Then entry $s_{i,i}$ of S_i is equal to the number of ones in column i of A_i, and $s_{i,j}$, where $i \neq j$, is equal to the number of rows in A_i in which both columns i and j of A_i have a value of one. For $i = 3, 4, 5$, Greig proved the following:

Theorem 1.11. *The only possibilities, up to row and column rearrangement, for S_i, for $i = 3, 4, 5$, are as given in Figures 1.1, 1.2, and 1.3.*

Using the fact that the rows of A_i have even weight, one finds that each S_i in Figures 1.1–1.3 correspond to a unique A_i. Therefore, if $C^{\perp}$ contains a weight i word $\vec{w}$, where $3 \leq i \leq 5$, then the number of possibilities, up to row and column rearrangement, for the columns in A that are supported by $\vec{w}$ are seven if $i = 5$, three if $i = 4$, and one if $i = 3$.

$$\begin{bmatrix} 8 & 4 & 4 \\ 4 & 8 & 4 \\ 4 & 4 & 8 \end{bmatrix}$$

Figure 1.1: The only possibility for S_3 up to row and column rearrangement.

$$\begin{bmatrix} 8 & 4 & 3 & 1 \\ 4 & 8 & 1 & 3 \\ 3 & 1 & 8 & 4 \\ 1 & 3 & 4 & 8 \end{bmatrix} \quad \begin{bmatrix} 8 & 4 & 2 & 2 \\ 4 & 8 & 2 & 2 \\ 2 & 2 & 8 & 4 \\ 2 & 2 & 4 & 8 \end{bmatrix} \quad \begin{bmatrix} 8 & 3 & 3 & 2 \\ 3 & 8 & 2 & 3 \\ 3 & 2 & 8 & 3 \\ 2 & 3 & 3 & 8 \end{bmatrix}$$

Figure 1.2: The only possibilities for S_4 up to row and column rearrangement.

$$\begin{bmatrix} 8 & 2 & 2 & 2 & 2 \\ 2 & 8 & 2 & 2 & 2 \\ 2 & 2 & 8 & 2 & 2 \\ 2 & 2 & 2 & 8 & 2 \\ 2 & 2 & 2 & 2 & 8 \end{bmatrix} \begin{bmatrix} 8 & 3 & 3 & 2 & 2 \\ 3 & 8 & 2 & 3 & 2 \\ 3 & 2 & 8 & 3 & 2 \\ 2 & 3 & 3 & 8 & 2 \\ 2 & 2 & 2 & 2 & 8 \end{bmatrix} \begin{bmatrix} 8 & 4 & 4 & 2 & 2 \\ 4 & 8 & 2 & 4 & 2 \\ 4 & 2 & 8 & 4 & 2 \\ 2 & 4 & 4 & 8 & 2 \\ 2 & 2 & 2 & 2 & 8 \end{bmatrix} \begin{bmatrix} 8 & 4 & 3 & 3 & 2 \\ 4 & 8 & 3 & 3 & 2 \\ 3 & 3 & 8 & 4 & 2 \\ 3 & 3 & 4 & 8 & 2 \\ 2 & 2 & 2 & 2 & 8 \end{bmatrix}$$

$$\begin{bmatrix} 8 & 3 & 3 & 3 & 3 \\ 3 & 8 & 3 & 3 & 3 \\ 3 & 3 & 8 & 3 & 3 \\ 3 & 3 & 3 & 8 & 1 \\ 3 & 3 & 3 & 1 & 8 \end{bmatrix} \quad \begin{bmatrix} 8 & 3 & 3 & 3 & 3 \\ 3 & 8 & 4 & 3 & 2 \\ 3 & 4 & 8 & 2 & 3 \\ 3 & 3 & 2 & 8 & 2 \\ 3 & 2 & 3 & 2 & 8 \end{bmatrix} \quad \begin{bmatrix} 8 & 2 & 4 & 3 & 3 \\ 2 & 8 & 4 & 3 & 3 \\ 4 & 4 & 8 & 2 & 2 \\ 3 & 3 & 2 & 8 & 2 \\ 3 & 3 & 2 & 2 & 8 \end{bmatrix}$$

Figure 1.3: The only possibilities for S_5 up to row and column rearrangement.

The last result we will mention in this section is that the full automorphism group of a $(22, 33, 12, 8, 4)$ is either a 2-group or the trivial group. This result was found by Landgev and Tonchev [12] and Kapralov [11]. For further results on $(22, 33, 12, 8, 4)$-BIBDs see the survey by van Rees [19].

1.4 Our Approach

Our approach to investigating whether or not a $(22, 33, 12, 8, 4)$-BIBD exists is based on the fact that if it does exist then its point code, which is a doubly-even self-orthogonal code with length 33, also exists.

Suppose a $(22, 33, 12, 8, 4)$-BIBD B exists. Let A be the incidence matrix of B and let C be the point code of B. We know C is a $(33, k)$ doubly-even self-orthogonal code, for some dimension k, where $8 \leq k \leq 16$. As we shall later see, we also know there must exist a $(33, 16)$ doubly-even self-orthogonal code C' that contains C. Therefore, if a $(22, 33, 12, 8, 4)$-BIBD B exists then there must exist a $(33, 16)$ doubly-even self-orthogonal code C' that contains 22 weight 12 words that form the incidence matrix A of B.

Suppose we enumerate a list L of $(33, 16)$ doubly-even self-orthogonal codes that contains one and only one code from each equivalence class of $(33, 16)$ doubly-even self-orthogonal codes. Then we know there must exist a code C'' in L that is equivalent to C'. Furthermore, since C'' differs from C' in only a permutation of its coordinates, C'' must contain 22 weight 12 words that form an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD that is isomorphic to B. Thus, a $(22, 33, 12, 8, 4)$-BIBD exists if and only if there exists a code in L that contains 22 weight 12 words that form the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD.

Our method for examining whether or not a $(22, 33, 12, 8, 4)$-BIBD exists consists of two main stages. The first stage is to enumerate a list L of $(33, 16)$ doubly-even self-

orthogonal codes containing one and only one code from each equivalence class. The second stage is to determine, for each code C in L, whether or not C contains 22 weight 12 words that form the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. If we find a code in L that contains the incidence matrix of such a design then we have found a $(22, 33, 12, 8, 4)$-BIBD, and thus, we know a $(22, 33, 12, 8, 4)$-BIBD exists. If none of the codes in L contain the incidence matrix of such a design then we know a $(22, 33, 12, 8, 4)$-BIBD does not exist.

As we shall later see, we can enumerate a list L of inequivalent $(33, 16)$ doubly-even self-orthogonal codes from the $(34, 17)$ self-dual codes. Using the $(34, 17)$ self-dual code enumerated using the methods of [2], we have enumerated a list L of inequivalent $(33, 16)$ doubly-even self-orthogonal codes. The number of codes in L is 594. Note that since each column of the incidence matrix contains 8 ones, we do not include in our list L the $(33, 16)$ doubly-even self-orthogonal codes that have a coordinate consisting entirely of zeros.

Our problem now is to determine, for each code C in L, whether or not C contains 22 weight 12 words that form the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. That is, for each C in L, we want to determine if C contains a set S of 22 weight 12 words such that: (1) for every pair of words $\vec{u}, \vec{v}$ in S, there are exactly 4 coordinates in which both $\vec{u}$ and $\vec{v}$ have a value of 1, and (2) for each coordinate j, there are exactly 8 words in S that have a value of 1 in coordinate j. Unfortunately, most of the codes contain approximately $10,000$ weight 12 words which makes a simple clique search of each code infeasible. Therefore, much work is required in determining whether or not a code contains the incidence matrix of such a design.

We use two different approaches to determine whether or not a code contains the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. One is a theoretical approach, which works by proving that any code that contains a certain structure cannot contain the incidence matrix of the design. We then remove all codes from L that contain the structure. We

have eliminated 116 codes using this method, leaving us with 478. For the codes we have not been able to theoretically prove cannot contain an incidence matrix, we use an algorithmic approach. That is, we write efficient computer programs for searching the weight 12 words of a code in order to determine whether or not the code contains the incidence matrix. At the time of writing, we have used this method to search 155 of the 478 codes that were not eliminated theoretically, leaving us with 323 codes to search. Thus far, none of the codes we have searched contained the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD.

Unfortunately, due to the size of the search space, we have not been able to search every code. Therefore, the problem of whether or not a $(22, 33, 12, 8, 4)$-BIBD exists is still an open problem. However, we do believe our algorithm for searching the codes demonstrates that it is not infeasible to search every code. Our algorithmic approach may also aid in finding proofs for the non-existence of the incidence matrix in certain classes of the codes we have not eliminated theoretically.

1.5 What's Ahead

In Chapter 2, we will discuss the coding theory we will need in order to produce a list L of $(33, k)$ doubly-even self-orthogonal codes with the property that a $(22, 33, 12, 8, 4)$-BIBD exists if and only if L contains a code that contains the incidence matrix of such a design.

First, it will be shown that any $(33, k)$ doubly-even self-orthogonal code is contained in a $(33, 16)$ doubly-even self-orthogonal code. It will then be shown how we can enumerate a list L of inequivalent $(33, 16)$ doubly-even self-orthogonal codes from any complete list of inequivalent $(34, 17)$ self-dual codes. It will also be shown how we can find the automorphism group of each code in L.

There are two main reasons for finding the automorphism groups. One reason is to aid us in our search for the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. The other reason

is that, as we shall later see, having the automorphism groups allows us to check that we did not make any mistakes in our enumeration. For this check, we will need to know the total number of $(33, 16)$ doubly-even self-orthogonal codes. Therefore, we will also calculate this number.

In Chapters 3 and 4, we consider the problem of determining whether or not a $(33, 16)$ doubly-even self-orthogonal code contains the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. In Chapter 3, we discuss our theoretical approach to this problem, in which we prove any code that contains certain structures cannot contain the incidence matrix of such a design. We will also give a proof, that does not depend on the results of Greig [8] (whose results required much computational time), that there are 3 possibilities, up to row and column rearrangement, for any four columns supported by a weight 4 word, in the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. In Chapter 4, we discuss our algorithmic approach, in which we develop efficient algorithms for searching the weight 12 words of a code for the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. Since we have not yet been able to search every code with our algorithm, we will also discuss some of the ideas we have for improving our algorithmic approach.

Finally, in Chapter 5, we will discuss the contributions we have made in this thesis towards determining whether or not a $(22, 33, 12, 8, 4)$-BIBD exists.

Chapter 2

The Length 33 Doubly-Even Self-Orthogonal Codes

Our approach to investigating whether or not a $(22, 33, 12, 8, 4)$-BIBD exists is based on the fact that if it does exist then so does its point code, which we know must be a length 33 doubly-even self-orthogonal code. Our approach consists of two main stages:

1. First, we find a list L of length 33 doubly-even self-orthogonal codes with the property that if C is the point code of a $(22, 33, 12, 8, 4)$-BIBD then there exists at least one code in L that contains a code that is equivalent to C.

2. For each code C in L, we then determine whether or not C contains the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD.

In this chapter, we will look at how we can find such a list L of length 33 doubly-even self-orthogonal codes.

In Section 2.1, we will discuss some standard coding theory concepts that we will use in some of the proofs given in this chapter. In Section 2.2, we will prove several straightforward coding theory results that we will need to prove the results presented

later in the chapter. In Section 2.3, we will show that any complete list L of inequivalent $(33, 16)$ doubly-even self-orthogonal codes is just the sort of list we are looking for. This will be accomplished by proving that for any $(33, k)$ doubly-even self-orthogonal code C, there exists a $(33, 16)$ doubly-even self-orthogonal code that contains C. The theory in this section, along with Section 2.5, is based on the work of MacWilliams, Sloane, and Thompson in [14].

In Section 2.4, we will discuss how we can enumerate a list L of inequivalent $(33, 16)$ doubly-even self-orthogonal codes, along with their automorphism groups. Our method for enumerating such a list L is dependent on us having a complete list of inequivalent $(34, 17)$ self-dual codes available, along with their automorphism groups. Therefore, we will also discuss how we can use the methods presented in [2] to enumerate a list of inequivalent $(34, 17)$ self-dual codes and their automorphism groups. We produce the automorphism groups of the $(33, 16)$ doubly-even self-orthogonal codes for two reasons: (1) to aid us in our search for an incidence matrix in a code, and (2) to check that we have not made any mistakes in our enumeration. How they are used to check our enumeration is also discussed in Section 2.4.

One piece of theory we will need in order to check our enumeration is the total number of $(33, 16)$ doubly-even self-orthogonal codes. Therefore, in Section 2.5, we will find this total. Finally, in Section 2.6, we will conclude this chapter with a review of the main results presented in the chapter.

2.1 Concepts And Notation

In this section, we will discuss some of the standard coding theory concepts and notation that we will use in several of the proofs presented later in the chapter.

We begin with the concept of a *translate* of a code. Let C be any (n, k) linear code over $GF(2)$. For a given $\vec{v} \in V_n(2)$, the set $\{\vec{c} + \vec{v} \mid \vec{c} \in C\}$, written $C + \vec{v}$, is called a

translate of C. Two well known properties of translates are: (1) $|C + \vec{v}| = |C|$, and (2) for any $\vec{v}_1, \vec{v}_2 \in V_n(2)$, either $C + \vec{v}_1 = C + \vec{v}_2$ or $C + \vec{v}_1 \cap C + \vec{v}_2 = \emptyset$. Together, these two properties tell us that if C and D are linear codes in which $C \subseteq D$ then D can be uniquely written as the union of $|D|/|C|$ distinct translates of C:

$$D = C \cup (C + \vec{v}_1) \cup (C + \vec{v}_2) \cup \cdots \cup (C + \vec{v}_l)$$

where $l = |D|/|C| - 1$ and $\vec{v}_i \in D$, for $i = 1, 2, \ldots, l$.

Let C, D be linear codes such that $C \subseteq D$. We will use the notation $D - C$ to denote the set $\{\vec{v} \mid \vec{v} \in D \text{ and } \vec{v} \notin C\}$. A well known result we will use throughout this chapter is the following:

Lemma 2.1. *Let C, D be linear codes such that $C \subseteq D$. Suppose for all $\vec{u}, \vec{v} \in D - C$ we have $\vec{u} + \vec{v} \in C$. Then, if $D - C \neq \emptyset$, $|D - C| = |C| = \frac{1}{2}|D|$.*

Proof. See [14]. □

We conclude this section by describing a process called *augmenting a code*, in which we create a larger code from a smaller code. Let C be an (n, k) linear code over $GF(2)$. Let $\vec{v} \in V_n(2) - C$. Let $D = C \cup (C + \vec{v})$. It is easy to show that D is an $(n, k+1)$ linear code over $GF(2)$. This process of adding a vector from $V_n(2) - C$ to the code C, creating the larger code D, is called *augmenting* the code C to D.

2.2 Some Standard Coding Theory Results

In this section, we will prove some of the well known and straightforward coding theory results we will use throughout the remainder of this chapter.

We begin with a lemma that tells us, for two vectors $\vec{u}$ and $\vec{v}$ in $V_n(2)$, what the weight of $\vec{u} + \vec{v}$ is modulo 4. This lemma immediately gives us the weight of $\vec{u} + \vec{v}$ modulo 2, which we give in a corollary to the lemma.

Lemma 2.2. *Let $\vec{u}, \vec{v} \in V_n(2)$. If $\vec{u} \bullet \vec{v} = 0$ then $w(\vec{u}+\vec{v}) \equiv w(\vec{u}) + w(\vec{v}) \bmod 4$. If $\vec{u} \bullet \vec{v} = 1$ then $w(\vec{u}+\vec{v}) \equiv w(\vec{u}) + w(\vec{v}) + 2 \bmod 4$.*

Proof. Consider the 3 non-zero possibilities for the values of $\vec{u}$ and $\vec{v}$ in a coordinate j:

u_j	1	1	0
v_j	1	0	1
	x_3	x_2	x_1

Let x_i, $1 \leq i \leq 3$, denote the number of times each of these possibilities occur in $\vec{u}$ and $\vec{v}$. Then $w(\vec{u}) = x_3 + x_2$, $w(\vec{v}) = x_3 + x_1$, and $w(\vec{u}+\vec{v}) = x_2 + x_1$. Therefore:

$$\begin{aligned} w(\vec{u}+\vec{v}) &\equiv x_2 + x_1 && \bmod 4 \\ &\equiv 4x_3 + x_2 + x_1 && \bmod 4 \\ &\equiv (x_3 + x_2) + (x_3 + x_1) + 2x_3 && \bmod 4 \\ &\equiv w(\vec{u}) + w(\vec{v}) + 2x_3 && \bmod 4 \end{aligned}$$

If $\vec{u} \bullet \vec{v} = 0$ then $x_3 \equiv 0 \bmod 2$, which implies $2x_3 \equiv 0 \bmod 4$, and thus, $w(\vec{u}+\vec{v}) \equiv w(\vec{u}) + w(\vec{v}) \bmod 4$. If $\vec{u} \bullet \vec{v} = 1$ then $x_3 \equiv 1 \bmod 2$, which implies $2x_3 \equiv 2 \bmod 4$, and thus, $w(\vec{u}+\vec{v}) \equiv w(\vec{u}) + w(\vec{v}) + 2 \bmod 4$. □

Corollary 2.3. *Let $\vec{u}, \vec{v} \in V_n(2)$. Then $w(\vec{u}+\vec{v}) \equiv w(\vec{u}) + w(\vec{v}) \bmod 2$.*

Next, we will show that for any self-orthogonal code C, the weights of the vectors in any translate of C are congruent modulo 2.

Lemma 2.4. *Let C be an (n,k) self-orthogonal code over $GF(2)$. Let $\vec{v} \in V_n(2)$. Then for any $\vec{u} \in C + \vec{v}$, we have $w(\vec{u}) \equiv w(\vec{v}) \bmod 2$.*

Proof. If $\vec{u} \in C + \vec{v}$ then $\vec{u} = \vec{c} + \vec{v}$ for some $\vec{c} \in C$. By Corollary 2.3, $w(\vec{c}+\vec{v}) \equiv$

$w(\vec{c}) + w(\vec{v}) \bmod 2$. Since C is self-orthogonal, $w(\vec{c}) \equiv 0 \bmod 2$. Therefore:

$$\begin{aligned} w(\vec{u}) &\equiv w(\vec{c}+\vec{v}) && \bmod 2 \\ &\equiv w(\vec{c}) + w(\vec{v}) && \bmod 2 \\ &\equiv 0 + w(\vec{v}) && \bmod 2 \\ &\equiv w(\vec{v}) && \bmod 2 \end{aligned}$$

□

We will now show that if C is a doubly-even self-orthogonal code and if $\vec{v}$ is a vector in $C^{\perp}$, then the weights of the vectors in the translate $C + \vec{v}$ are congruent modulo 4.

Lemma 2.5. *Let C be an (n, k) doubly-even self-orthogonal code over $GF(2)$. Let $\vec{v} \in C^{\perp}$. Then for any $\vec{u} \in C + \vec{v}$, we have $w(\vec{u}) \equiv w(\vec{v}) \bmod 4$.*

Proof. If $\vec{u} \in C + \vec{v}$ then $\vec{u} = \vec{c} + \vec{v}$ for some $\vec{c} \in C$. Since $\vec{v} \in C^{\perp}$, $\vec{c} \bullet \vec{v} = 0$. Therefore, by Lemma 2.2, $w(\vec{c}+\vec{v}) \equiv w(\vec{c}) + w(\vec{v}) \bmod 4$. Since C is doubly-even, $w(\vec{c}) \equiv 0 \bmod 4$. Therefore:

$$\begin{aligned} w(\vec{u}) &\equiv w(\vec{c}+\vec{v}) && \bmod 4 \\ &\equiv w(\vec{c}) + w(\vec{v}) && \bmod 4 \\ &\equiv 0 + w(\vec{v}) && \bmod 4 \\ &\equiv w(\vec{v}) && \bmod 4 \end{aligned}$$

□

Next we will consider when the process of augmenting a self-orthogonal code results in a larger code that is also self-orthogonal. Let C be an (n, k) self-orthogonal code, let $\vec{v}$ be a vector in $V_n(2) - C$, and let D denote the augmented code $C \cup (C + \vec{v})$. Then, as we have seen, D is an $(n, k+1)$ linear code. However, D is not necessarily a self-orthogonal code. We want to determine which vectors $\vec{v} \in V_n(2) - C$ will result in a D that is also self-orthogonal. One requirement we have of $\vec{v}$ is that it must be an element of $C^{\perp}$. The reason for this is that $C \subset D$, and thus $\vec{v}$ must be orthogonal to every codeword in C.

A second requirement is that $\vec{v}$ must have even weight since it must be orthogonal to itself. This leaves us with the even weight vectors $\vec{v} \in C^{\perp} - C$. As the next lemma demonstrates, any such $\vec{v}$ will result in D being a self-orthogonal code.

Lemma 2.6. *Let C be an (n, k) self-orthogonal code over $GF(2)$. Let $\vec{v}$ be a vector in $C^{\perp} - C$ that has even weight. Then the $(n, k+1)$ linear code $D = C \cup (C + \vec{v})$ is a self-orthogonal code over $GF(2)$.*

Proof. To prove that D is self-orthogonal, all we need to show is that for any two (not necessarily distinct) codewords $\vec{u}_1$ and $\vec{u}_2$ in D, we have $\vec{u}_1 \bullet \vec{u}_2 = 0$. We have three cases to consider, based on which of the sets, C and $C + \vec{v}$, the codewords $\vec{u}_1$ and $\vec{u}_2$ are in: (1) If $\vec{u}_1$ and $\vec{u}_2$ are both elements of C then since C is self-orthogonal it immediately follows that $\vec{u}_1 \bullet \vec{u}_2 = 0$. (2) Suppose only one of $\vec{u}_1$ and $\vec{u}_2$ is an element of C, say $\vec{u}_1 \in C$. Then $\vec{u}_2 = \vec{c}_2 + \vec{v}$, for some $\vec{c}_2 \in C$. Therefore, $\vec{u}_1 \bullet \vec{u}_2 = \vec{u}_1 \bullet (\vec{c}_2 + \vec{v}) = (\vec{u}_1 \bullet \vec{c}_2) + (\vec{u}_1 \bullet \vec{v}) = 0 + 0 = 0$ since $\vec{u}_1$ is in C and $\vec{c}_2, \vec{v}$ are both elements of $C^{\perp}$. (3) Suppose neither one of $\vec{u}_1$ and $\vec{u}_2$ are elements of C. Then $\vec{u}_1 = \vec{c}_1 + \vec{v}$ and $\vec{u}_2 = \vec{c}_2 + \vec{v}$ for some codewords $\vec{c}_1, \vec{c}_2 \in C$. Since $\vec{c}_1$ and $\vec{c}_2$ are in C, $\vec{v}$ is in $C^{\perp}$, and $\vec{v}$ has even weight, we have $\vec{u}_1 \bullet \vec{u}_2 = (\vec{c}_1 + \vec{v}) \bullet (\vec{c}_2 + \vec{v}) = (\vec{c}_1 \bullet \vec{c}_2) + (\vec{c}_1 \bullet \vec{v}) + (\vec{v} \bullet \vec{c}_2) + (\vec{v} \bullet \vec{v}) = 0 + 0 + 0 + 0 = 0$. Therefore, for any $\vec{u}_1, \vec{u}_2 \in D$, we have $\vec{u}_1 \bullet \vec{u}_2 = 0$. Thus the $(n, k+1)$ linear code D is a self-orthogonal code. □

Let C be an (n, k) self-orthogonal code. Since $C \subseteq C^{\perp}$, we know $C^{\perp}$ can be uniquely partitioned into $l = |C^{\perp}|/|C|$ distinct translates of C:

$$C^{\perp} = C \cup (C + \vec{v}_2) \cup (C + \vec{v}_3) \cup \cdots \cup (C + \vec{v}_l)$$

where $\vec{v}_i \in C^{\perp}$. If $C + \vec{v}$ is any translate of C, where $\vec{v} \in C^{\perp} - C$, then $C + \vec{v}$ must be equal to one of the $l - 1$ translates $C + \vec{v}_i$. Therefore, the only augmented codes of C that are self-orthogonal are the codes $C + \vec{v}_i$ in which $\vec{v}_i$ has even weight. Since, by Lemma 2.4, the weights of the vectors in any translate of C are congruent modulo 2, we

can determine how many of the codes $C + \vec{v}_i$ are self-orthogonal by simply finding the total number of even weight vectors in $C^{\perp}$.

Our next lemma tells us how many even weight vectors are in the orthogonal complement of a self-orthogonal code.

Lemma 2.7. *Let C be an (n, k) self-orthogonal code over $GF(2)$. If $\vec{1} \in C$ then every vector in $C^{\perp}$ has even weight. If $\vec{1} \notin C$ then exactly half the vectors in $C^{\perp}$ have even weight.*

Proof. Suppose $\vec{1} \in C$. Then every vector in $C^{\perp}$ must be orthogonal to $\vec{1}$. Therefore, every vector in $C^{\perp}$ must have even weight.

Suppose $\vec{1} \notin C$. Let $C_0^{\perp}$ denote the set of even weight vectors in $C^{\perp}$ and let $C_1^{\perp}$ denote the set of odd weight vectors in $C^{\perp}$. If $\vec{u}, \vec{v} \in C_0^{\perp}$ then, by Corollary 2.3, we have $w(\vec{u} + \vec{v}) \equiv w(\vec{u}) + w(\vec{v}) \equiv 0 + 0 \equiv 0 \bmod 2$, which implies $\vec{u} + \vec{v} \in C_0^{\perp}$, which in turn implies $C_0^{\perp}$ is a linear code. If $\vec{u}, \vec{v} \in C_1^{\perp}$ then, by Corollary 2.3, we have $w(\vec{u} + \vec{v}) \equiv w(\vec{u}) + w(\vec{v}) \equiv 1 + 1 \equiv 0 \bmod 2$, which implies $\vec{u} + \vec{v} \in C_0^{\perp}$. Therefore, if $C_1^{\perp} \neq \emptyset$ then, by Lemma 2.1, we must have $|C_0^{\perp}| = |C_1^{\perp}| = \frac{1}{2}|C^{\perp}|$. Thus, all we have to do is show that if $\vec{1} \notin C$ then $C_1^{\perp} \neq \emptyset$.

Suppose $n \equiv 1 \bmod 2$. Since the codewords in C have even weight, $\vec{1} \in C^{\perp}$. Furthermore, since n is odd, $\vec{1}$ has odd weight. Therefore, $\vec{1} \in C_1^{\perp}$. Thus, if $n \equiv 1 \bmod 2$ then $C_1^{\perp} \neq \emptyset$.

Suppose $n \equiv 0 \bmod 2$. Assume $C_1^{\perp} = \emptyset$. That is, assume every vector in $C^{\perp}$ has even weight. Consider the code $D = C \cup (C + \vec{1})$. Since $\vec{1} \in C^{\perp} - C$ and has even weight, by Lemma 2.6, D is an $(n, k+1)$ self-orthogonal code. Since the vectors in $C^{\perp}$ have even weight, for any $\vec{v} \in C^{\perp}$ we have $\vec{1} \bullet \vec{v} = 0$. Therefore, for any $\vec{u} \in C$ and $\vec{v} \in C^{\perp}$ we have $(\vec{u} + \vec{1}) \bullet \vec{v} = (\vec{u} \bullet \vec{v}) + (\vec{1} \bullet \vec{v}) = 0 + 0 = 0$, which implies every vector in $C^{\perp}$ is orthogonal to every vector in $C + \vec{1}$. Thus, every vector in $C^{\perp}$ is orthogonal to every vector in D, which implies $C^{\perp} \subseteq D^{\perp}$. However, $|D^{\perp}| = 2^{n-(k+1)} = 2^{n-k-1} < 2^{n-k} = |C^{\perp}|$, which

contradicts $C^{\perp} \subseteq D^{\perp}$. Therefore, our assumption that every vector in $C^{\perp}$ has even weight was wrong. Thus, if $n \equiv 0 \bmod 2$ then $C_{\vec{1}}^{\perp} \neq \emptyset$.

Therefore, if $\vec{1} \notin C$ then exactly half the vectors in $C^{\perp}$ have even weight. □

In our next lemma, we will show that either all or half of the codewords in a self-orthogonal code C have weight $\equiv 0 \bmod 4$. Of course, since the codewords in a self-orthogonal code all have even weight, if half of the codewords in C have weight $\equiv 0 \bmod 4$, then the remaining half have weight $\equiv 2 \bmod 4$.

Lemma 2.8. *Let C be an (n, k) self-orthogonal code over $GF(2)$. Then C contains either 2^k or 2^{k-1} codewords with weight $\equiv 0 \bmod 4$.*

Proof. Let C_0 denote the set of codewords in C with weight $\equiv 0 \bmod 4$, and let C_2 denote the set of codewords in C with weight $\equiv 2 \bmod 4$. If $C_2 = \emptyset$ then $C_0 = C$, which implies $|C_0| = 2^k$.

Assume $C_2 \neq \emptyset$. If $\vec{u}, \vec{v} \in C$ then, since C is self-orthogonal, we have $\vec{u} \bullet \vec{v} = 0$. Therefore, by Lemma 2.2, for any $\vec{u}, \vec{v} \in C$, we have $w(\vec{u} + \vec{v}) \equiv w(\vec{u}) + w(\vec{v}) \bmod 4$. If $\vec{u}, \vec{v} \in C_0$ then $w(\vec{u}+\vec{v}) \equiv w(\vec{u}) + w(\vec{v}) \equiv 0+0 \equiv 0 \bmod 4$. Therefore, $\vec{u}+\vec{v} \in C_0$, which implies C_0 is a linear code. If $\vec{u}, \vec{v} \in C_2$ then $w(\vec{u}+\vec{v}) \equiv w(\vec{u}) + w(\vec{v}) \equiv 2+2 \equiv 0 \bmod 4$, which implies $\vec{u}+\vec{v} \in C_0$. Therefore, by Lemma 2.1, since $C_2 \neq \emptyset$, we have $|C_2| = |C_0| = \frac{1}{2}|C| = 2^{k-1}$. □

Let C be a doubly-even self-orthogonal code and let $\vec{v}$ be any even weight vector in $V_n(2)$ that is not an element of $C^{\perp}$. In our next lemma, we will count the number of codewords $\vec{c} \in C$ in which the weight of $\vec{v} + \vec{c}$ is congruent to 0 modulo 4.

Lemma 2.9. *Let C be an (n, k) doubly-even self-orthogonal code over $GF(2)$. Let $\vec{v} \in V_n(2) - C^{\perp}$ such that $w(\vec{v}) \equiv 0 \bmod 2$. Then $w(\vec{v} + \vec{c}) \equiv 0 \bmod 4$ for exactly 2^{k-1} codewords $\vec{c} \in C$.*

Proof. Let $C_0 = \{\vec{c} \in C \mid \vec{v} \bullet \vec{c} = 0\}$ and let $C_1 = \{\vec{c} \in C \mid \vec{v} \bullet \vec{c} = 1\}$. If $\vec{u}_0, \vec{u}_1 \in C_0$ then $\vec{v} \bullet (\vec{u}_0 + \vec{u}_1) = \vec{v} \bullet \vec{u}_0 + \vec{v} \bullet \vec{u}_1 = 0 + 0 = 0$, which implies $\vec{u}_0 + \vec{u}_1 \in C_0$. Therefore, C_0 is a linear code. If $\vec{u}_0, \vec{u}_1 \in C_1$ then $\vec{v} \bullet (\vec{u}_0 + \vec{u}_1) = \vec{v} \bullet \vec{u}_0 + \vec{v} \bullet \vec{u}_1 = 1 + 1 = 0$, which implies $\vec{u}_0 + \vec{u}_1 \in C_0$. Since $\vec{v} \notin C^{\perp}$, we know there must exist a $\vec{c} \in C$ such that $\vec{v} \bullet \vec{c} = 1$, which implies $C_1 \neq \emptyset$. Therefore, by Lemma 2.1, we have $|C_1| = |C_0| = \frac{1}{2}|C| = 2^{k-1}$. Thus, $\vec{v} \bullet \vec{c} = 0$ for exactly 2^{k-1} codewords $\vec{c} \in C$, and $\vec{v} \bullet \vec{c} = 1$ for exactly 2^{k-1} codewords $\vec{c} \in C$.

Let $\vec{c} \in C$. Then, since C is doubly-even, $w(\vec{c}) \equiv 0 \bmod 4$. Therefore, by Lemma 2.2, if $\vec{v} \bullet \vec{c} = 0$ then $w(\vec{v} + \vec{c}) \equiv w(\vec{v}) + w(\vec{c}) \equiv w(\vec{v}) \bmod 4$, and if $\vec{v} \bullet \vec{c} = 1$ then $w(\vec{v} + \vec{c}) \equiv w(\vec{v}) + w(\vec{c}) + 2 \equiv w(\vec{v}) + 2 \bmod 4$. Therefore, $w(\vec{v} + \vec{c}) \equiv w(\vec{v}) \bmod 4$ for exactly 2^{k-1} codewords $\vec{c} \in C$, and $w(\vec{v} + \vec{c}) \equiv w(\vec{v}) + 2 \bmod 4$ for exactly 2^{k-1} codewords $\vec{c} \in C$. Thus, since $w(\vec{v}) \equiv 0 \bmod 2$, we have $w(\vec{v} + \vec{c}) \equiv 0 \bmod 4$ for exactly 2^{k-1} codewords $\vec{c} \in C$. □

We will now count the number of vectors in $V_n(2)$ that have weight congruent to 0 modulo 4, for n odd.

Lemma 2.10. *Let $n \equiv 1 \bmod 2$. The number of $\vec{v} \in V_n(2)$ in which $w(\vec{v}) \equiv 0 \bmod 4$ is:*

$$\begin{aligned} &2^{n-2} + 2^{(n-3)/2}, && \text{if } n \equiv \pm 1 \bmod 8, \\ &2^{n-2} - 2^{(n-3)/2}, && \text{if } n \equiv \pm 3 \bmod 8. \end{aligned}$$

Proof. The number of binary n-vectors with weight $\equiv 0 \bmod 4$ is:

$$\sum_{j=0}^{\lfloor n/4 \rfloor} \binom{n}{4j}.$$

In order to evaluate this equation, consider the equation $(1^j + (-1)^j + i^j + (-i)^j) = x$, where $i^2 = -1$ and $j \geq 0$. If $j \equiv 0 \bmod 4$ then $x = 4$, otherwise $x = 0$. Therefore,

$\binom{n}{j}(i^j + (-1)^j + i^j + (-i)^j)$ is equal to $4\binom{n}{j}$ if $j \equiv 0 \bmod 4$, and 0 if $j \not\equiv 0 \bmod 4$. Thus, the number of binary n-vectors with weight $\equiv 0 \bmod 4$ is:

$$\begin{aligned}
\sum_{j=0}^{\lfloor n/4 \rfloor} \binom{n}{4j} &= \frac{1}{4}\sum_{j=0}^{n} \binom{n}{j}(1^j + (-1)^j + i^j + (-i)^j) \\
&= \frac{1}{4}(\sum_{j=0}^{n} \binom{n}{j}1^j + \sum_{j=0}^{n} \binom{n}{j}(-1)^j + \sum_{j=0}^{n} \binom{n}{j}i^j + \sum_{j=0}^{n} \binom{n}{j}(-i)^j) \\
&= \frac{1}{4}(2^n + 0 + (1+i)^n + (1-i)^n) \\
&= 2^{n-2} + \frac{1}{4}((1+i)^n + (1-i)^n).
\end{aligned}$$

If $n \equiv \pm 1 \bmod 8$ then $(1+i)^n + (1-i)^n = 2^{(n+1)/2}$, and if $n \equiv \pm 3 \bmod 8$ then $(1+i)^n + (1-i)^n = -2^{(n+1)/2}$. This leads us to our result. □

Our final lemma is given without a proof. For a given (n, s) self-orthogonal code C, the lemma counts the number of (n, k) self-orthogonal codes that contain C. Such a count for the case of $n \equiv 0 \bmod 2$ and $\vec{1} \in C$ was first given, with proof, in [14]. A count for the case of $n \equiv 1 \bmod 2$ and the case of $n \equiv 0 \bmod 2$ and $\vec{1} \notin C$ were first given in [20]. Both can be proved using the methods of [14].

Lemma 2.11. *Let C be an (n, s) self-orthogonal code over $GF(2)$, where $0 \le s \le \lfloor n/2 \rfloor$. The number of (n, k) self-orthogonal codes over $GF(2)$, where $s \le k \le \lfloor n/2 \rfloor$, that contain C is:*

$$\begin{cases}
\displaystyle\prod_{i=1}^{k-s} \frac{2^{n-2(k-i)} - 1}{2^i - 1}, & \text{if } n \equiv 0 \bmod 2 \text{ and } \vec{1} \in C, \\
\displaystyle\frac{2^{n-k-s} - 1}{2^{n-2s} - 1} \prod_{i=1}^{k-s} \frac{2^{n-2(k-i)} - 1}{2^i - 1}, & \text{if } n \equiv 0 \bmod 2 \text{ and } \vec{1} \notin C, \text{ and} \\
\displaystyle\prod_{i=1}^{k-s} \frac{2^{n-2(k-i)-1} - 1}{2^i - 1}, & \text{if } n \equiv 1 \bmod 2.
\end{cases}$$

Note that an empty product equals 1.

Corollary 2.12. *Let C be an (n, s) self-orthogonal code over $GF(2)$, where $0 \leq s \leq \lfloor n/2 \rfloor$. Then there exists an (n, k) self-orthogonal code over $GF(2)$ that contains C, where $s \leq k \leq \lfloor n/2 \rfloor$.*

This completes our look at some of the known coding theory results we will use throughout this chapter. In the next section, we will encounter most of the results presented in this section.

2.3 Containment of the $(33, k)$ Doubly-Even Codes

In this section, it will be shown that if C is an (n, k) doubly-even self-orthogonal code with length $n \equiv 0, \pm 1 \bmod 8$, then C can be augmented to an $(n, \lfloor n/2 \rfloor)$ doubly-even self-orthogonal code. This, in turn, tells us that for $n \equiv 0, \pm 1 \bmod 8$, any (n, k) doubly-even self-orthogonal code is contained in an $(n, \lfloor n/2 \rfloor)$ doubly-even self-orthogonal code.

We begin by showing that for any (n, k) doubly-even self-orthogonal code C with dimension $k \leq \lfloor n/2 \rfloor - 1$, there always exists an $(n, \lfloor n/2 \rfloor - 1)$ doubly-even self-orthogonal code that contains C, no matter what the length n is. We will prove this by showing that any (n, k) doubly-even self-orthogonal code, where $k \leq \lfloor n/2 \rfloor - 2$, can always be augmented to an $(n, k+1)$ doubly-even self-orthogonal code. This, in turn, tells us that if we are given any (n, k) doubly-even self-orthogonal code C, where $k \leq \lfloor n/2 \rfloor - 2$, then we can repeatedly add vectors to C until we produce an $(n, \lfloor n/2 \rfloor - 1)$ doubly-even self-orthogonal code that contains C. (Of course, we do not have to consider the case of $k = \lfloor n/2 \rfloor - 1$ since this implies C is an $(n, \lfloor n/2 \rfloor - 1)$ doubly-even self-orthogonal code that contains itself.) Our proof is similar to the one given in [14].

Theorem 2.13. *Let C be an (n, k) doubly-even self-orthogonal code over $GF(2)$, where $0 \leq k \leq \lfloor n/2 \rfloor - 2$. Then C can be augmented to an $(n, k+1)$ doubly-even self-orthogonal code over $GF(2)$.*

Proof. By Corollary 2.12, we know there exists an $(n, \lfloor n/2 \rfloor)$ self-orthogonal code D over $GF(2)$ that contains C. By Lemma 2.8, D contains at least $2^{\lfloor n/2 \rfloor - 1}$ words with weight $\equiv 0 \bmod 4$. Since C has dimension $k \leq \lfloor n/2 \rfloor - 2$, C contains at most $2^{\lfloor n/2 \rfloor - 2}$ words with weight $\equiv 0 \bmod 4$. Therefore, there exists a $\vec{v} \in D$ such that $\vec{v} \notin C$ and $w(\vec{v}) \equiv 0 \bmod 4$. Furthermore, since $C \subset D$ and D is self-orthogonal, $D \subset C^{\perp}$, which implies $\vec{v} \in C^{\perp}$. Thus, by Lemma 2.6, the augmented code $C \cup (C + \vec{v})$ is an $(n, k+1)$ self-orthogonal code over $GF(2)$. Furthermore, since C is doubly-even and $\vec{v} \in C^{\perp}$, by Lemma 2.5, we know that for all $\vec{u} \in C + \vec{v}$ we have $w(\vec{u}) \equiv w(\vec{v}) \equiv 0 \bmod 4$, which implies $C \cup (C + \vec{v})$ is doubly-even. Thus, C can be augmented to an $(n, k+1)$ doubly-even self-orthogonal code over $GF(2)$. □

Corollary 2.14. *Let C be an (n, k) doubly-even self-orthogonal code over $GF(2)$, where $0 \leq k \leq \lfloor n/2 \rfloor - 1$. Then there exists an $(n, \lfloor n/2 \rfloor - 1)$ doubly-even self-orthogonal code that contains C.*

All we have left to do is to determine, for a given length n and doubly-even code C of dimension $k \leq \lfloor n/2 \rfloor - 1$, whether or not there exists an $(n, \lfloor n/2 \rfloor)$ doubly-even self-orthogonal code that contains C. The case for $n \equiv 0 \bmod 2$ can be found in [14], where it is shown there exists an $(n, n/2)$ doubly-even self-orthogonal code that contains C if and only if $n \equiv 0 \bmod 8$. Therefore, in our proof, we will only consider the case $n \equiv 1 \bmod 2$, which mirrors the proof of the even case. Of course, since by Corollary 2.14 we already know that any (n, k) doubly-even self-orthogonal code C, where $k \leq \lfloor n/2 \rfloor - 1$, is contained in an $(n, \lfloor n/2 \rfloor - 1)$ doubly-even self-orthogonal code, all we need to prove is that any $(n, \lfloor n/2 \rfloor - 1)$ doubly-even self-orthogonal code can be augmented to an $(n, \lfloor n/2 \rfloor)$ doubly-even self-orthogonal code.

Theorem 2.15. *Let C be an $(n, \lfloor n/2 \rfloor - 1)$ doubly-even self-orthogonal code over $GF(2)$. Then C can be augmented to an $(n, \lfloor n/2 \rfloor)$ doubly-even self-orthogonal code over $GF(2)$ if and only if $n \equiv 0, \pm 1 \bmod 8$.*

Proof. If $n \equiv 0 \bmod 2$, then it is shown in [14] that C can be augmented to an $(n, n/2)$ doubly-even code if and only if $n \equiv 0 \bmod 8$.

Suppose $n \equiv 1 \bmod 2$. Then $|C| = 2^{((n-1)/2)-1}$ and $|C^{\perp}| = 2^n/|C| = 2^{((n-1)/2)+2}$. Therefore, $C^{\perp}$ can be written as the union of $|C^{\perp}|/|C| = 8$ translates of C:

$$C^{\perp} = C \cup (C + \vec{g}_1) \cup (C + \vec{g}_2) \cup \cdots \cup (C + \vec{g}_7).$$

Since $n \equiv 1 \bmod 2$, $\vec{1} \notin C$. Therefore, by Lemma 2.7, we know that exactly half the words in $C^{\perp}$ have even weight. Since C is a doubly-even self-orthogonal code and $\vec{g}_i \in C^{\perp}$, by Lemma 2.5, for each $\vec{v} \in C + \vec{g}_i$, we have $w(\vec{v}) \equiv w(\vec{g}_i) \bmod 4$. This implies the weight of the words in each translate have the same parity. Therefore, exactly 4 of the 8 translates that make up $C^{\perp}$ consist of words with even weight. Since C is self-orthogonal, we know C must be one of them. Therefore, exactly $4 - 1 = 3$ of the translates $C + \vec{g}_i$ consist of even weight words. Without loss of generality, assume they are $C + \vec{g}_1$, $C + \vec{g}_2$, and $C + \vec{g}_3$. Then, by Lemma 2.6, $C \cup (C + \vec{g}_i)$, where $1 \leq i \leq 3$, are $(n, (n-1)/2)$ self-orthogonal codes. These are the only $(n, (n-1)/2)$ self-orthogonal codes to which C can be augmented. We want to determine whether or not at least one of these self-orthogonal codes is doubly-even.

Consider the $(n, n-1)$ linear code F consisting of all even weight vectors of length n. Write F as the union of $|F|/|C| = 2^{n-1}/2^{((n-1)/2)-1} = 2^{(n+1)/2}$ translates of C:

$$F = C \cup (C + \vec{g}_1) \cup (C + \vec{g}_2) \cup (C + \vec{g}_3) \cup (C + \vec{f}_1) \cup \cdots \cup (C + \vec{f}_l).$$

The first four translates of F consist of the even weight vectors in $C^{\perp}$, while the remaining $l = 2^{(n+1)/2} - 4$ translates consist of the even weight vectors not in $C^{\perp}$. Let us count the number of vectors with weight $\equiv 0 \bmod 4$ in each translate. Since C is doubly-even, C contains $|C| = 2^{((n-1)/2)-1}$ words with weight $\equiv 0 \bmod 4$. Since each vector $\vec{f}_i$ is in $V_n(2) - C^{\perp}$ and has even weight, by Lemma 2.9, we know that each translate $C + \vec{f}_i$ contains exactly $|C|/2 = 2^{((n-1)/2)-2}$ words with weight $\equiv 0 \bmod 4$. Finally, since for

any $\vec{v} \in C + \vec{g}_i$ we know $w(\vec{v}) \equiv w(\vec{g}_i) \bmod 4$, each translate $C + \vec{g}_i$ contains either 0 or $|C| = 2^{((n-1)/2)-1}$ words with weight $\equiv 0 \bmod 4$.

Let e denote the number of translates $C + \vec{g}_i$ that consist of words with weight $\equiv 0 \bmod 4$. Let K denote the number of vectors in $V_n(2)$ with weight $\equiv 0 \bmod 4$. Then K is equal to the total number of words with weight $\equiv 0 \bmod 4$ in each of the translates that make up F. Therefore:

$$\begin{aligned} K &= |C| + l(|C|/2) + e|C| \\ &= (|C|/2)l + |C|(e+1) \\ &= 2^{((n-1)/2)-2}(2^{(n+1)/2} - 4) + 2^{((n-1)/2)-1}(e+1) \\ &= 2^{n-2} - 2^{(n-1)/2} + 2^{((n-1)/2)-1}e + 2^{((n-1)/2)-1} \\ &= 2^{n-2} - 2^{((n-1)/2)-1}(2-1) + 2^{((n-1)/2)-1}e \\ &= 2^{n-2} - 2^{(n-3)/2} + 2^{(n-3)/2}e \\ &= 2^{n-2} + 2^{(n-3)/2}(e-1). \end{aligned}$$

Lemma 2.10 tells us the value of K for $n \equiv 1 \bmod 2$. If $n \equiv \pm 1 \bmod 8$ then $K = 2^{n-2} + 2^{(n-3)/2}$, which implies $e = 2$. If $n \equiv \pm 3 \bmod 8$ then $K = 2^{n-2} - 2^{(n-3)/2}$, which implies $e = 0$. Therefore, if $n \equiv \pm 1 \bmod 8$ there exists a $\vec{g}_i$ with weight $\equiv 0 \bmod 4$, and thus, C can be augmented to a doubly-even code. If $n \equiv \pm 3 \bmod 8$ then there does not exist a $\vec{g}_i$ with weight $\equiv 0 \bmod 4$, and thus C cannot be augmented to a doubly-even code. Thus, if n is odd then C can be augmented to an $(n, \lfloor n/2 \rfloor)$ doubly-even self-orthogonal code over $GF(2)$ if and only if $n \equiv \pm 1 \bmod 8$. □

Corollary 2.16. *Let C be an (n, k) doubly-even self-orthogonal code over $GF(2)$. Then there exists an $(n, \lfloor n/2 \rfloor)$ doubly-even self-orthogonal code over $GF(2)$ that contains C if and only if $n \equiv 0, \pm 1 \bmod 8$.*

Since $33 \equiv 1 \bmod 8$, Corollary 2.16 tells us that if C is a $(33, k)$ doubly-even self-orthogonal code over $GF(2)$, then there must exist at least one $(33, 16)$ doubly-even

self-orthogonal code over $GF(2)$ that contains C. Therefore, if C is the code generated by the incidence matrix A of a $(22, 33, 12, 8, 4)$-BIBD then there must exist a $(33, 16)$ doubly-even self-orthogonal code that contains A. Thus, we need only search for the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD in the $(33, 16)$ doubly-even self-orthogonal codes over $GF(2)$.

2.4 Enumeration of the $(33, 16)$ Doubly-Even Codes

We now know that if a $(22, 33, 12, 8, 4)$-BIBD exists, then there must exist a $(33, 16)$ doubly-even self-orthogonal code C over $GF(2)$ that contains 22 weight 12 words that form the incidence matrix of such a design. Therefore, if we enumerate a list L of all such codes then we can determine whether or not a $(22, 33, 12, 8, 4)$-BIBD exists by searching each code in L for the incidence matrix of such a design.

In this section, we will consider the problem of enumerating a list of inequivalent $(33, 16)$ doubly-even self-orthogonal codes over $GF(2)$. That is, a list L of generator matrices with that property that for any equivalence class $\mathcal{C}$ of $(33, 16)$ doubly-even self-orthogonal codes, there exists one and only one matrix G in L that generates a code in $\mathcal{C}$. The reason we only produce generator matrices for inequivalent codes is that if C is any $(33, 16)$ doubly-even self-orthogonal code that contains the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD B, then if C' is any code that is equivalent to C, then C' will contain the incidence matrix for a design that is isomorphic to B. Therefore, a $(22, 33, 12, 8, 4)$-BIBD exists if and only if our complete list L of inequivalent $(33, 16)$ double-even self-orthogonal codes contains the incidence matrix of such a design.

We will also consider, in this section, the problem of finding generators for the automorphism group of each code that is generated by a matrix in our list L. Producing the automorphism groups will prove useful in both checking that our list of codes is complete and in searching for an incidence matrix in a given code.

As we shall see, we can both enumerate our list of inequivalent $(33, 16)$ doubly-even self-orthogonal codes and find their automorphism groups if we have available any complete list of inequivalent $(34, 17, d \geq 4)$ self-dual codes along with their automorphism groups. Such a list of $(34, 17)$ self-dual codes and automorphism groups has been produced using the methods of [2].

2.4.1 Enumerating the $(33, 16)$ Codes From the $(34, 17)$ Codes

Let C be a $(33, 16)$ self-orthogonal code over $GF(2)$. Suppose we insert a coordinate consisting entirely of zeros after the $(j-1)^{th}$ coordinate of C, producing a $(34, 16)$ self-orthogonal code C_j. Suppose we then add the all ones vector to C_j, producing the code $D = C_j \cup (C_j + \vec{1})$. Since $\vec{1} \in C_j^{\perp}$ and $w(\vec{1})$ is even, we know D is a self-orthogonal code. Since the j^{th} coordinate of C_j consists entirely of zeros, we know $\vec{1} \notin C_j$, which implies that adding $\vec{1}$ to C_j increases its dimension by 1. Therefore, D has dimension 17, which implies it is a self-dual code.

What we have just shown is that, given the $(33, 16)$ self-orthogonal code C over $GF(2)$, we can *lengthen* C to the $(34, 17)$ self-dual code D over $GF(2)$. The converse of this is also true. That is, given the $(34, 17)$ self-dual code D we can *shorten* D to the $(33, 16)$ self-orthogonal code C by first removing all words from D that have a value of 1 in the j^{th} coordinate, producing the code C_j, and then deleting the j^{th} coordinate from C_j, producing C. We refer to this process as shortening the code D by taking a *cross-section* of D at coordinate j.

We will now show that if we take a cross-section of any $(34, 17)$ self-dual code D at any coordinate j then the resulting code is always a $(33, 16)$ self-orthogonal code. Let C be the cross-section of D at coordinate j. Since D is a self-dual code and since each codeword in C is just a codeword in D with a zero component removed, we know C must be a self-orthogonal code with length 33. The dimension of C depends on the number of codewords in D that have a value of 0 in coordinate j. As the following lemma

demonstrates, the number of such codewords is always 2^{16} for any coordinate j.

Lemma 2.17. *Let D be a $(2k, k)$ self-dual code over $GF(2)$. Then exactly half of the codewords in D have a value of 0 in coordinate j, where $1 \leq j \leq 2k$.*

Proof. Let D_0 denote the set of codewords in D that have a value of 0 in coordinate j, and let D_1 denote the set of codewords in D that have a value of 1 in coordinate j. If $\vec{u}, \vec{v} \in D_0$ then $\vec{u}+\vec{v} \in D_0$, which implies D_0 is a linear code. If $\vec{u}, \vec{v} \in D_1$ then $\vec{u}+\vec{v} \in D_0$. Since D is self-dual, we know $\vec{1} \in D$, which implies D_1 is not the empty set. Therefore, by Lemma 2.1, we have $|D_1| = |D_0| = \frac{1}{2}|D|$. □

Thus, for any $(34, 17)$ self-dual code D and coordinate j, the code C produced by taking a cross-section of D at coordinate j is a $(33, 16)$ self-orthogonal code.

Let us now determine whether or not taking the cross-section of a $(34, 17)$ self-dual code D results in a doubly-even code. Let C be the $(33, 16)$ self-orthogonal code produced by taking the cross-section of D at coordinate j. Then the weight distribution of the codewords in C is equal to the weight distribution of the codewords in D that have a value of 0 in coordinate j. Therefore, C is doubly-even if and only if the 2^{16} codewords in D that have a value of 0 in coordinate j all have weight $\equiv 0 \bmod 4$. However, since $\vec{1} \in D$ and $w(\vec{1}) \equiv 2 \bmod 4$, we know D is not doubly-even. Therefore, it is not necessarily true that the codewords in D that have a value of 0 in coordinate j all have weight $\equiv 0 \bmod 4$. In fact, since D is not doubly-even, we know, by Lemma 2.8, that exactly 2^{16} codewords in D have weight $\equiv 0 \bmod 4$. Thus, C is doubly-even if and only if the 2^{16} codewords in D with weight $\equiv 0 \bmod 4$ all have a value of 0 in the coordinate j. This leads us to the concept of a *doubly-even coordinate*:

Definition 2.18. Let D be a $(2k, k)$ self-dual code over $GF(2)$. A *doubly-even coordinate* of D is any coordinate j in which all the codewords in D with weight $\equiv 0 \bmod 4$ have a value of 0.

We summarize the properties of a doubly-even coordinate in the following theorem:

Theorem 2.19. *Let D be a $(2k, k)$ self-dual code over $GF(2)$. Suppose D has a doubly-even coordinate j. Then D contains 2^{k-1} codewords with weight $\equiv 0 \bmod 4$ and 2^{k-1} codewords with weight $\equiv 2 \bmod 4$. Furthermore, the codewords with weight $\equiv 0 \bmod 4$ all have a value of 0 in coordinate j, and the codewords with weight $\equiv 2 \bmod 4$ all have a value of 1 in coordinate j.*

Proof. Let D_0 denote the set of codewords in D that have weight $\equiv 0 \bmod 4$, and let D_1 denote the set of codewords in D that have weight $\equiv 2 \bmod 4$. By our definition of a doubly-even coordinate, the codewords in D_0 all have a value of 0 in coordinate j. By Lemma 2.17, the number of codewords in D with a value of 0 in coordinate j is 2^{k-1}, which implies $|D_0| \leq 2^{k-1}$. Since, by Lemma 2.8, D contains either 2^k or 2^{k-1} codewords with weight $\equiv 0 \bmod 4$, the fact that $|D_0| \leq 2^{k-1}$ implies $|D_0| = 2^{k-1}$. Therefore, $|D_1| = |D| - |D_0| = 2^{k-1}$. Furthermore, since D contains exactly 2^{k-1} codewords with a value of 0 in coordinate j, and since the 2^{k-1} codewords in D_0 all have a value of 0 in coordinate j, the codewords in D_1 must all have a value of 1 in coordinate j. □

As we have just seen, taking a cross-section of a $(34, 17)$ self-dual code D at coordinate j results in a code C that is doubly-even if and only if coordinate j is a doubly-even coordinate. Therefore, if D does not have any doubly-even coordinates, then D cannot be shortened to a $(33, 16)$ doubly-even self-orthogonal code. Furthermore, as we shall soon see, if D has more than one doubly-even coordinate then each of the $(33, 16)$ doubly-even self-orthogonal codes to which D can be shortened will have a coordinate of zeros. However, we are only interested in producing $(33, 16)$ doubly-even self-orthogonal codes that may contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. Since the incidence matrix of such a design contains 8 ones in every column, if a $(33, 16)$ doubly-even self-orthogonal code C has a coordinate of zeros then C cannot contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. Therefore, we are only interested in the $(34, 17)$ self-dual codes D that have one and only one doubly-even coordinate.

We will now show that if D has more than one doubly-even coordinate then taking the cross section of D at one of its doubly-even coordinates will always result in a code that has a coordinate of zeros. Suppose D has at least two doubly-even coordinates, say j_0 and j_1. Then, by Theorem 2.19, coordinates j_0 and j_1 of D must be identical. Consider C, the cross-section of D at coordinate j_0. Since coordinate j_1 of D is identical to coordinate j_0, every codeword in C must have a value of 0 in the coordinate of C that corresponds to coordinate j_1 of D. Therefore, C has a coordinate of zeros. Thus, if D has more than one doubly-even coordinate then if we take a cross-section of D at one of its doubly-even coordinates, the resulting code C will have a coordinate that consists entirely of zeros.

Since we are only interested in the $(33, 16)$ doubly-even self-orthogonal codes that may contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD, we will only enumerate a list L_C of inequivalent $(33, 16)$ doubly-even self-orthogonal codes that do not have a coordinate of zeros. Since the $(34, 17)$ self-dual codes that have more than one doubly-even coordinate cannot be shortened to such codes, we do not have to consider them in our enumeration. So, let us now determine which of the $(34, 17)$ self-dual codes have more than one doubly-even coordinate.

As we have already seen, if a $(34, 17)$ self-dual code D has more than one doubly-even coordinate then D must have identical coordinates. The next lemma shows us that D has identical coordinates if and only if D has distance 2.

Lemma 2.20. *Let D be a $(2k, k)$ self-dual code over $GF(2)$. Then D has identical coordinates if and only if D contains a weight 2 word. Furthermore, if j_0 and j_1 are identical coordinates in D then D contains a weight 2 word whose ones are located in coordinates j_0 and j_1, and vice-versa.*

Proof. Suppose D contains a weight 2 word $\vec{c}_2$. Let j_0 and j_1 be the coordinates of D in which $\vec{c}_2$ has a value of 1. Consider any codeword $\vec{c} \in D$. Since D is self-dual, $\vec{c}$ must be

orthogonal to $\vec{c}_2$. Therefore, $\vec{c}$ must have values of either 0 0 or 1 1 in coordinates j_0 and j_1. Thus, j_0 and j_1 are identical coordinates in D.

Suppose D has two identical coordinates j_0 and j_1. Let $\vec{c}_2$ be the weight 2 vector whose ones occur in coordinates j_0 and j_1. Consider any codeword $\vec{c} \in D$. Since coordinates j_0 and j_1 of $\vec{c}$ are identical, $\vec{c}_2$ must be orthogonal to $\vec{c}$. Therefore, $\vec{c}_2 \in D^{\perp}$. Since D is self-dual, we know $D = D^{\perp}$, which implies $\vec{c}_2 \in D$. Thus, $\vec{c}_2$ is a weight 2 word in D. □

Therefore, if D is a $(34, 17)$ self-dual code with more than one doubly-even coordinate, then D must have distance 2. As the next lemma demonstrates, if D is a $(34, 17, 2)$ self-dual code that does not have two or more doubly-even coordinates, then D does not have any doubly-even coordinates.

Lemma 2.21. *Let D be a $(2k, k, 2)$ self-dual code over $GF(2)$. Then D either has 0 or 2 or more doubly-even coordinates.*

Proof. Assume D has one and only one doubly-even coordinate, say coordinate j. Consider any weight 2 word $\vec{c}_2$ in D. Let j_0 and j_1 be the coordinates in D in which $\vec{c}_2$ has a value of 1. Then, by Lemma 2.20, coordinates j_0 and j_1 are identical, which implies neither coordinate can be coordinate j since coordinate j is distinct. Therefore, $\vec{c}_2$ must have a value of 0 in coordinate j. However, coordinate j is a doubly-even coordinate and $w(\vec{c}_2) \equiv 2 \bmod 4$, which, by Theorem 2.19, implies $\vec{c}_2$ has a value of 1 in coordinate j, which is a contradiction. Therefore, our assumption that D has one and only one doubly-even coordinate was incorrect. Thus, D either has 0 or 2 or more doubly-even coordinates. □

Together, Lemmas 2.20 and 2.21 tell us that we do not have to consider the $(34, 17, 2)$ self-dual codes. That is, if D is a $(34, 17)$ self-dual code that has more than one doubly-even coordinate then D must have distance 2. Furthermore, if D is a $(34, 17, 2)$ self-dual code then D cannot have one and only one doubly-even coordinate.

Therefore, if D is a $(34, 17)$ self-dual code with one and only one doubly-even coordinate then D must have distance ≥ 4. Furthermore, if D is a $(34, 17)$ self-dual code with distance ≥ 4 then, by Lemma 2.20, D cannot have identical coordinates, which implies D has either 0 or 1 doubly-even coordinate. Thus, the only $(34, 17)$ self-dual codes that can be shortened to a $(33, 16)$ doubly-even self-orthogonal code, that does not have a coordinate of zeros, are the codes with distance ≥ 4 that have a doubly-even coordinate. Let us now consider how we can use this fact to enumerate a list L_C of *inequivalent* $(33, 16)$ doubly-even self-orthogonal codes that do not have a coordinate of zeros.

Let D_0 and D_1 be any two equivalent $(34, 17, d \geq 4)$ self-dual codes. Suppose D_0 has a doubly-even coordinate, say coordinate j_0. Then, since the coordinates of D_1 are simply a permutation of the coordinates of D_0, there must exist a coordinate j_1 in D_1 that is a doubly-even coordinate. Let C_0 and C_1 be the $(33, 16)$ doubly-even self-orthogonal codes that are obtained by taking cross-sections of D_0 and D_1 at coordinates j_0 and j_1, respectively. Then, since all we are doing when we take cross-sections of these codes is deleting the words with weight $\equiv 2 \bmod 4$ and then dropping a coordinate of zeros, the codes C_0 and C_1 must be equivalent.

Let us now consider the converse. Let C_0 and C_1 be any two equivalent $(33, 16)$ self-orthogonal codes that do not have a coordinate of zeros. Suppose we lengthen C_0 and C_1 to the $(34, 17, d \geq 4)$ self-dual codes D_0 and D_1, respectively. Then, since all we are doing when we lengthen a code is inserting a coordinate of zeros and then adding the all ones vector, the codes D_0 and D_1 must be equivalent.

Thus, we have just proven the following:

Theorem 2.22. *Let D_0 and D_1 be $(34, 17, d \geq 4)$ self-dual codes over $GF(2)$ such that D_0 and D_1 have doubly-even coordinates j_0 and j_1, respectively. Let C_0 and C_1 be the $(33, 16)$ doubly-even self-orthogonal codes that are obtained by taking cross-sections of D_0 and D_1 at coordinates j_0 and j_1, respectively. Then C_0 and C_1 are equivalent if and only if D_0 and D_1 are equivalent.*

Therefore, we can enumerate a list L_C of inequivalent $(33, 16)$ doubly-even self-orthogonal codes, that do not have a coordinate of zeros, using any complete list L_D of inequivalent $(34, 17, d \geq 4)$ self-dual codes. This is accomplished by examining each code D in L_D to see if it has a doubly-even coordinate j. For each code D with such a coordinate j, the cross-section of D at coordinate j is taken, producing a code C which is added to L_C. By Theorem 2.22, we know each code added to L_C will be inequivalent to every other code in L_C. Furthermore, the list L_C will be complete.

Let us now consider how we can use the automorphism groups of the codes in L_D to produce the automorphism groups of the codes in L_C.

Let D be any $(34, 17, d \geq 4)$ self-dual code that has a doubly-even coordinate j. Let C be the cross-section of D at coordinate j. Since D has distance ≥ 4, we know there exists one and only one such coordinate j in D. Therefore, if π_D is any automorphism of D then, since coordinate j is the only coordinate in D in which all codewords with weight $\equiv 0 \bmod 4$ have a value of 0, π_D cannot move coordinate j. Furthermore, since π_D does not move coordinate j, π_D must map the set of codewords in D that have a value of 0 in coordinate j to itself. Thus, if we rewrite π_D with the corresponding coordinates in C then the resulting permutation will be an automorphism of C.

The converse is also true. To see this, let π_C be any automorphism of C and let π_D be π_C rewritten in terms of the coordinates of D. Consider the sets D_0 and D_1, where D_i is the set of all codewords in D that have a value of i in coordinate j. Since $\pi_C C = C$ and since C is just D_0 with coordinate j deleted, we have $\pi_D D_0 = D_0$. Furthermore, since D is self-dual, we know $\vec{1} \in D$, which implies $D_1 = D_0 + \vec{1}$, and thus, $\pi_D D_1 = \pi_D(D_0 + \vec{1}) = \pi_D D_0 + \pi_D \vec{1} = D_0 + \vec{1} = D_1$. Therefore $\pi_D D = \pi_D(D_0 \cup D_1) = D$, which implies π_D is an automorphism of D.

Thus, we have just proven the following:

Theorem 2.23. *Let D be a $(34, 17, d \geq 4)$ self-dual code over $GF(2)$ that has a doubly-even coordinate j. Let C be the cross-section of D at coordinate j. Let $\mathcal{AUT}(D)$ denote the*

automorphism group of D. Then none of the permutations in $\mathcal{AUT}(D)$ move coordinate j. Furthermore, if we rewrite the permutations in $\mathcal{AUT}(D)$ in terms of the coordinates of C then the resulting set of permutations is the automorphism group of C.

Together, Theorems 2.22 and 2.23 give us our algorithm for enumerating a list of inequivalent $(33, 16)$ doubly-even self-orthogonal codes that do not have a coordinate of zeros, and finding their automorphism groups, given any complete list of inequivalent $(34, 17, d \geq 4)$ self-dual codes and their automorphism groups.

2.4.2 The Enumeration Algorithm

We now have the logic behind an algorithm for enumerating a list of inequivalent $(33, 16)$ doubly-even self-orthogonal codes, that do not contain a coordinate of zeros, along with their automorphism groups. Let us now have a look at the details of the implementation of this algorithm.

Since each $(34, 17, d \geq 4)$ self-dual code contains 2^{17} codewords, we store our codes as generator matrices. That is, each $(34, 17, d \geq 4)$ code D is stored as a 34×17 matrix G_D whose rows generate the codewords in D. We also store the automorphism group $\mathcal{AUT}(D)$ of each code D as a set of permutations that generate the group. Let us now consider how we can use these data structures to accomplish the operations we need to perform in our algorithm.

Let us first consider how we can use a generator matrix for a code D to determine whether or not a coordinate in D is a doubly-even coordinate. In order to do this, we first require the following lemma:

Lemma 2.24. *Let G be a generator matrix for an (n, k) self-orthogonal code C over $GF(2)$. Let $\vec{c}_1, \vec{c}_2, \ldots, \vec{c}_k$ denote the codewords in C that are also rows in G. Let $\vec{c}$ be any non-zero codeword in C. Suppose $\vec{c}$ can be written as the sum of m rows in G. Then the*

weight of $\vec{c}$ is congruent modulo 4 to the sum of the weights of these m rows. That is, if $\vec{c} = \vec{c}_{i_1} + \vec{c}_{i_2} + \cdots + \vec{c}_{i_m}$, then $w(\vec{c}) \equiv \sum_{j=1}^{m} w(\vec{c}_{i_j}) \bmod 4$.

Proof. By induction on m.

If $\vec{c}$ is a row in G, say $\vec{c} = \vec{c}_i$, then $w(\vec{c}) \equiv w(\vec{c}_i) \bmod 4$.

Assume that the weight of any codeword that can be written as the sum of r rows in G is congruent modulo 4 to the sum of the weights of the r rows. Let $\vec{c}$ be any codeword that can be written as the sum of $r+1$ rows in G, say $\vec{c} = \sum_{j=1}^{r+1} \vec{c}_{i_j}$. Then $\vec{c} = \vec{c}_{i_1} + \vec{u}$, where $\vec{u} = \sum_{j=2}^{r+1} \vec{c}_{i_j}$. Since C is a self-orthogonal code, by Lemma 2.2, we have $w(\vec{c}) \equiv w(\vec{c}_{i_1}) + w(\vec{u}) \bmod 4$. By our induction hypothesis, since $\vec{u}$ is the sum of r rows in G, $w(\vec{u}) \equiv \sum_{j=2}^{r+1} w(\vec{c}_{i_j})$. Therefore, $w(\vec{c}) \equiv w(\vec{c}_{i_1}) + \sum_{j=2}^{r+1} w(\vec{c}_{i_j}) \bmod 4$, and thus, the weight of $\vec{c}$ is congruent modulo 4 to the sum of the weights of the $r+1$ rows that sum to $\vec{c}$.

Thus, by the principle of mathematical induction our proposition is true. □

Lemma 2.24 leads us to the following theorem which, given a generator matrix G_D for a $(34, 17, d \geq 4)$ self-dual code D, shows us how we can determine whether or not a coordinate j in D is a doubly-even coordinate by simply examining the weights of the rows in G_D and column j of G_D.

Theorem 2.25. *Let G_D be a generator matrix for a $(2k, k, d \geq 4)$ self-dual code D over $GF(2)$. Then coordinate j of D is a doubly-even coordinate if and only if the rows in G_D that have weight $\equiv 0 \bmod 4$ all have a value of 0 in column j and the rows in G_D that have weight $\equiv 2 \bmod 4$ all have a value of 1 in column j.*

Proof. Suppose coordinate j of D is a doubly-even coordinate. Then, by Theorem 2.19, the rows in G_D that have weight $\equiv 0 \bmod 4$ must all have a value of 0 in column j and the rows in G_D that have weight $\equiv 2 \bmod 4$ must all have a value of 1 in column j.

Suppose each row in G_D that has weight $\equiv 0 \bmod 4$ has a value of 0 in column j, and suppose each row in G_D that has weight $\equiv 2 \bmod 4$ has a value of 1 in column j. Let G_0 denote the rows in G_D that have weight $\equiv 0 \bmod 4$, and let G_2 denote the rows in G_D that have weight $\equiv 2 \bmod 4$. Let $\vec{c}$ be any codeword in C. Suppose $\vec{c}$ can be written as the sum of m_0 rows in G_0 and m_2 rows in G_2. Then, by Lemma 2.24, the weight of $\vec{c}$ is congruent modulo 4 to the sum of the weights of the m_0 rows in G_0 and the m_2 rows in G_2. That is, $w(\vec{c}) \equiv 0m_0 + 2m_2 \equiv 2m_2 \bmod 4$. Therefore, if $m_2 \equiv 0 \bmod 2$ then $w(\vec{c}) \equiv 0 \bmod 4$, and if $m_2 \equiv 1 \bmod 2$ then $w(\vec{c}) \equiv 2 \bmod 4$. Furthermore, since the rows in G_0 all have a value of 0 in column j and the rows of G_2 all have a value of 1 in column j, if $m_2 \equiv 0 \bmod 2$ then $\vec{c}$ has a value of 0 in coordinate j and if $m_2 \equiv 1 \bmod 2$ then $\vec{c}$ has a value of 1 in coordinate j. Therefore, coordinate j is a doubly-even coordinate. □

Thus, given a generator matrix G_D for a $(34, 17, d \geq 4)$ self-dual code D, we can determine whether or not coordinate j of D is a doubly-even coordinate by a simple examination of the matrix G_D.

Let G_D be a generator matrix for a $(34, 17, d \geq 4)$ self-dual code D that has a doubly-even coordinate j. Let us now consider how we can use G_D and j to find a generator matrix G_C for the $(33, 16)$ doubly-even self-orthogonal code C that results when we take a cross-section of D at coordinate j. Let r be any row in G_D that has a value of 1 in column j. Suppose we add row r to every other row in G_D that has a value of 1 in column j, producing the matrix G'_D. Then G'_D also generates D. Furthermore, every row in G'_D, other than row r, has a value of 0 in column j, and thus generates the codewords in D that have a value of 0 in coordinate j. Therefore, if we delete row r from G'_D and then delete column j, the resulting 33×16 matrix is a generator matrix G_C for the $(33, 16)$ doubly-even self-orthogonal code C.

Let us now turn our attention to the automorphism group of C. That is, given any set of permutations Π_D that generates the automorphism group $\mathcal{AUT}(D)$ of D, let us now consider how we can find a set of permutations Π_C that generates the automorphism

group $\mathcal{AUT}(C)$ of C. Let π_D be any permutation in Π_D. Then, by Theorem 2.23, π_D does not move coordinate j. That is:

$$\pi_D = \begin{pmatrix} 1 & 2 & \cdots & j-1 & j & j+1 & \cdots & 34 \\ c_1 & c_2 & \cdots & c_{j-1} & j & c_{j+1} & \cdots & c_{34} \end{pmatrix}.$$

If we delete column j in π_D and replace every integer s in π_D, in which $s > j$, with the integer $s-1$ then, by Theorem 2.23, the resulting permutation π_C is an automorphism of C. Furthermore, if we do this to every permutation in Π_D then the resulting set of permutations Π_C generates the automorphism group of C.

This completes our look at how we can use our data structures to accomplish the operations we need to perform in our algorithm. We will now combine these operations to give us a formal description of our algorithm for enumerating a list of inequivalent $(33, 16)$ doubly-even self-orthogonal codes, that do not have a coordinate of zeros, and finding each code's automorphism group.

Algorithm 2.26. *EnumerationAlgorithm*(L_D, L_C):

- Input: A list L_D of pairs (G_D, Π_D). The matrix G_D generates a $(34, 17, d \geq 4)$ self-dual code D over $GF(2)$. The set of permutations Π_D generates $\mathcal{AUT}(D)$. The list L_D must have the property that for each equivalence class $\mathcal{D}$ of $(34, 17, d \geq 4)$ self-dual codes over $GF(2)$, there exists one and only one pair (G_D, Π_D) in L_D in which G_D generates a code in $\mathcal{D}$.

- Output: A list L_C of pairs (G_C, Π_C). The matrix G_C generates a $(33, 16)$ doubly-even self-orthogonal code C over $GF(2)$ that does not have a coordinate of zeros. The set of permutations Π_C generates $\mathcal{AUT}(C)$. The list L_C has the property that for each equivalence class $\mathcal{C}$ of $(33, 16)$ doubly-even self-orthogonal codes over $GF(2)$ that do not have a coordinate of zeros, there exists one and only one pair (G_C, Π_C) in L_C in which that G_C generates a code in $\mathcal{C}$.

begin
  Clear the list L_C.
  for each pair (G_D, Π_D) in L_D **do:**
    Find (if it exists) the one column j in G_D with the property that a row in G_D has a value of 0 in column j if and only if the row has weight $\equiv 0 \bmod 4$.
    if such a column j exists **then:**
      Let i be any row in G_D that has a value of 1 in column j.
      Add row i to every other row in G_D that has a value of 1 in column j, producing the matrix G'_D.
      Delete row r and column j from G'_D, producing the matrix G_C.
      Set $\Pi_C = \emptyset$.
      for each $\pi_D \in \Pi_D$ **do**
        Delete column j from π_D and then replace all integers s in π_D, in which $s \geq j$, with the integer $s - 1$, producing the permutation π_C.
        Insert π_C into Π_C.
      end for
      Insert (G_C, Π_C) into L_C.
    end if
  end for
end

2.4.3 Recursively Enumerating the $(34, 17)$ Codes

Algorithm 2.26 gives us our method for enumerating an inequivalent list of $(33, 16)$ doubly-even self-orthogonal codes, that do not have a coordinate of zeros, and finding the automorphism group of each code. Let us now consider how we can produce the input to Algorithm 2.26. That is, let us consider how we can enumerate a list of inequivalent $(34, 17)$ self-dual codes along with their automorphism groups.

As was mentioned earlier, we can use the methods of [2] to enumerate the $(34, 17)$ self-dual codes. In [2], we describe a recursive method for enumerating the $(2k, k)$ self-dual codes over $GF(2)$. The algorithm works in two stages: the Enumeration Stage and the Unique Representative Stage.

For a given length $2k$, the Enumeration Stage produces a list L of generator matrices for the $(2k, k)$ self-dual codes. The list L has the property that for each equivalence class $\mathcal{C}$ of $(2k, k)$ self-dual codes, there exists at least one matrix in L that generates a code in $\mathcal{C}$. The algorithm uses the $(2j, j, d \geq 4)$ self-dual codes, where $j \leq 2k - 2$, to enumerate the $(2k, k, 2)$ self-dual codes. The $(2j, j, d \geq 4)$ self-dual codes, where $j \leq 2k - 4$, are used to enumerate the $(2k, k, 4)$ self-dual codes. The $(2k, k, 4)$ self-dual codes that do not contain any intersecting weight 4 words are used to enumerate the $(2k, k, d \geq 6)$ self-dual codes.

For each matrix G in the list L produced by the Enumeration Stage, the Unique Representative Stage permutes the columns (and performs elementary row-operations) of G in such a way that all matrices that generate equivalent codes are permuted to the same matrix. This leaves us with a list containing one and only one generator matrix for each equivalence class of $(2k, k)$ self-dual codes. The Unique Representative Stage also computes the size of the automorphism group of each code found.

We used our algorithm in [2] to enumerate lists of the inequivalent $(2k, k)$ self-dual codes of length $2k \leq 32$. The sizes of the automorphism groups of each code were also computed. We have since made several improvements to the Unique Representative Stage of the algorithm. The algorithm now generally runs faster for most codes. The algorithm also now finds the automorphism group of each code, instead of only its size. We store each automorphism group as a set of permutations that generates the group. The number of generators required to store each group is less than $\binom{n}{2}$, where n is the length of the code.

We have used our improved algorithm to enumerate a list of inequivalent $(34, 17)$ self-

dual codes over $GF(2)$. We have also used the algorithm to produce the automorphism groups for each of the $(2k, k)$ self-dual codes enumerated, for $2k \leq 34$.

2.4.4 Results of the Enumerations

We now have a complete list L_D of pairs (G_D, Π_D), where G_D generates a $(34, 17, d \geq 4)$ self-dual code D (that is inequivalent to all other codes generated by a matrix in the list) and Π_D generates the automorphism group of D. The number of generator matrices in the list L_D is 20852, and thus there are 20852 equivalence classes of $(34, 17, d \geq 4)$ self-dual codes.

Using our list L_D as input, we have used Algorithm 2.26 to enumerate our complete list L_C of pairs (G_C, Π_C), where G_C generates a $(33, 16)$ doubly-even self-orthogonal code C that does not have a coordinate of zeros, and Π_C generates the automorphism group of C. The number of generator matrices in the list L_C is 594, and thus, there are 594 equivalence classes of $(33, 16)$ doubly-even self-orthogonal codes that do not have a coordinate of zeros. Generators for each code in L_C, and their automorphism groups, are available on the world wide web at: *www.cs.umanitoba.ca/~umbilou1/DoublyEvenCodes/*

2.4.5 Checking Our Results

We will now conclude this section by explaining how we can use the sizes of the automorphism groups to check that our enumerations of both the inequivalent $(2k, k)$ self-dual codes and the inequivalent $(33, 16)$ doubly-even self-orthogonal codes, that do not have a coordinate of zeros, were correct.

Let L be any complete list of inequivalent $(2k, k)$ self-dual codes. Our method for checking that we did not make any mistakes in our enumeration of L works as follows: First, we use the size of the automorphism group of each code C in L to compute the total number of codes that are equivalent to C. That is, if $|\mathcal{AUT}(C)| = x$ then there

are $(2k)!/x$ codes equivalent to C. We then compute the total number of $(2k, k)$ self-dual codes by summing up the number of codes that are equivalent to each code in L. That is, we compute $\sum_{C \in L}(2k)!/|\mathcal{AUT}(C)|$. We then check this total against the known theoretical number of $(2k, k)$ self-dual codes which we can compute using Lemma 2.11.

We have performed this check on each of our lists of $(2k, k)$ self-dual codes, for $2k \leq 34$. For each list, the number of codes computed using the automorphism group sizes matched up with the theoretical number of codes.

We can also perform such a check on our list L_C of inequivalent $(33, 16)$ doubly-even self-orthogonal codes that do not have a coordinate of zeros. That is, we can first find the total number of $(33, 16)$ doubly-even self-orthogonal codes that do not have a coordinate of zeros by computing $\sum_{C \in L_C} 33!/|\mathcal{AUT}(C)|$. We can then check this total against the theoretical number of such codes. However, in order to do so, we must first find the theoretical number of $(33, 16)$ doubly-even self-orthogonal codes that do not have a coordinate of zeros.

2.5 The Number of $(33, 16)$ Doubly-Even Codes

In this section, we will count the total number of $(n, (n-1)/2)$ doubly-even self-orthogonal codes over $GF(2)$, that do not have any coordinates of zeros, for $n \equiv 1 \bmod 8$.

Throughout this section, we will use the following notation:

- $\mathcal{A}(n, k)$ will denote the set of (n, k) self-orthogonal codes over $GF(2)$.
- $\mathcal{B}(n, k)$ will denote the subset of codes in $\mathcal{A}(n, k)$ that are doubly-even.

Recalling our definition of set difference, we also have $\mathcal{A}(n, k) - \mathcal{B}(n, k)$, the subset of codes in $\mathcal{A}(n, k)$ that are not doubly-even.

In [14], MacWilliams found, for a given $C \in \mathcal{B}(n, s)$, the number of codes in $\mathcal{B}(n, n/2)$ that contain C, where $n \equiv 0 \bmod 8$. Using a similar proof, we will first find, for a given

$C \in \mathcal{B}(n,s)$, the number of codes in $\mathcal{B}(n,k)$ that contain C, where $0 \leq s < k \leq \lfloor n/2 \rfloor$ and $n \equiv \pm 1 \bmod 8$. In order to do this, we will first need a couple of lemmas.

Lemma 2.27. *Let $C \in \mathcal{B}(n,s)$, where $n \equiv \pm 1 \bmod 8$. Then the number of vectors $\vec{v} \in C^{\perp}$ in which $w(\vec{v}) \equiv 2 \bmod 4$ is $2^{n-s-2} - 2^{((n-1)/2)-1}$.*

Proof. By Corollary 2.16, since $n \equiv \pm 1 \bmod 8$, we know there exists a $D \in \mathcal{B}(n,(n-1)/2)$ such that $C \subseteq D \subset D^{\perp} \subseteq C^{\perp}$. Write $C^{\perp}$ as the union of $|C^{\perp}|/|D| = 2^{n-s}/2^{(n-1)/2} = 2^{((n+1)/2)-s}$ translates of D:

$$C^{\perp} = D \cup (D + \vec{g}_1) \cup (D + \vec{f}_1) \cup \cdots \cup (D + \vec{f}_l),$$

where the first $|D^{\perp}|/|D| = 2$ translates form $D^{\perp}$, and thus, $l = 2^{((n+1)/2)-s} - 2$. We will find the number of vectors with weight $\equiv 2 \bmod 4$ in $C^{\perp}$ by counting the number of such vectors in each translate of D.

First, consider $D^{\perp} = D \cup (D + \vec{g}_1)$. Since $n \equiv 1 \bmod 2$, we know $\vec{1} \notin D$, and thus, by Lemma 2.7, exactly half the vectors in $D^{\perp}$ have even weight. Therefore, since the vectors in D have even weight, the vectors in $D + \vec{g}_1$ must all have odd weight. Furthermore, since D is doubly-even, the vectors in D have weight $\equiv 0 \bmod 4$. Thus, none of the vectors in $D^{\perp}$ have weight $\equiv 2 \bmod 4$.

Let us now consider the remaining l translates $D + \vec{f}_i$. Since $n \equiv 1 \bmod 2$, we know $\vec{1} \notin C$, and thus, by Lemma 2.7, exactly half the vectors in $C^{\perp}$ have even weight. Since exactly half the vectors in $D^{\perp} = D \cup (D + \vec{g}_1)$ have even weight, we know exactly half the vectors in the remaining l translates $D + \vec{f}_i$ have even weight. Since, by Lemma 2.4, the parity of the weight of the vectors in each translate $D + \vec{f}_i$ is equal to the parity of the weight of $\vec{f}_i$, we know $l/2$ of the translates consist of even weight vectors and $l/2$ of the translates consist of odd weight vectors. Since each $\vec{f}_i$ is not an element of $D^{\perp}$, if $\vec{f}_i$ has even weight then, by Lemma 2.9, exactly $|D|/2$ of the vectors in $D + \vec{f}_i$ have weight $\equiv 2 \bmod 4$. Therefore, the total number of vectors with weight $\equiv 2 \bmod 4$ in the l translates $D + \vec{f}_i$ is $(l/2)(|D|/2)$.

Thus, the number of vectors with weight $\equiv 2 \bmod 4$ in $C^{\perp}$ is:

$$\begin{aligned}(l/2)(|D|/2) &= ((2^{((n+1)/2)-s}-2)/2)(2^{(n-1)/2}/2) \\ &= (2^{((n+1)/2)-s-1}-1)(2^{((n-1)/2)-1}) \\ &= 2^{n-s-2}-2^{((n-1)/2)-1}.\end{aligned}$$

□

Lemma 2.28. *Let $C \in \mathcal{B}(n,s)$, where $n \equiv \pm 1 \bmod 8$ and $0 \leq s < (n-1)/2$. The number of codes in $\mathcal{A}(n,k)-\mathcal{B}(n,k)$, where $s < k \leq (n-1)/2$, that contain C is:*

$$2^{\frac{n-1}{2}-k}(2^{\frac{n-1}{2}-s}-1)\prod_{i=1}^{k-s-1}\frac{2^{n-2(k-i)-1}-1}{2^i-1}.$$

Proof. Let H denote the set of vectors $\vec{v} \in C^{\perp}$ with weight $\equiv 2 \bmod 4$. Let S denote the set of codes in $\mathcal{A}(n,k)-\mathcal{B}(n,k)$ that contain C. Let T denote the set of codes $D \in \mathcal{A}(n,k)-\mathcal{B}(n,k)$ for which there exists a $\vec{v} \in H$ such that $C \cup (C+\vec{v}) \subseteq D$.

We will first show that $S = T$. Suppose $A \in S$. Since A is not doubly-even there must exist a $\vec{v} \in A$ such that $w(\vec{v}) \equiv 2 \bmod 4$. Since $C \subset A$ and $\vec{v} \in A$, we know $C \cup (C+\vec{v}) \subseteq A$. Furthermore, since $C \subset A \subset C^{\perp}$, we know $\vec{v} \in H$. Therefore, there exists a $\vec{v} \in H$ such that $C \cup (C+\vec{v}) \subseteq A$, which implies $A \in T$. Thus, $S \subseteq T$. Conversely, if $A \in T$, then there exists a $\vec{v} \in H$ such that $C \cup (C+\vec{v}) \subseteq A$, which implies $C \subset A$. Thus, $T \subseteq S$. Therefore, $S = T$. Thus, we can find the number of codes in $\mathcal{A}(n,k)-\mathcal{B}(n,k)$ that contain C by finding $|T|$.

For each $\vec{v} \in H$, let $x(\vec{v})$ denote the number of codes in $\mathcal{A}(n,k)-\mathcal{B}(n,k)$ that contain $C \cup (C+\vec{v})$. For each $D \in T$, let $y(D)$ denote the number of vectors $\vec{v} \in H$ such that $C \cup (C+\vec{v}) \subseteq D$. Then:

$$\sum_{D \in T} y(D) = \sum_{\vec{v} \in H} x(\vec{v})$$

We will use this equality to find $|T|$.

Let us first consider $\sum_{\vec{v}\in H} x(\vec{v})$. Since C is doubly-even and $w(\vec{v}) \equiv 2 \bmod 4$, we know $\vec{v} \notin C$, and thus, $C \cup (C+\vec{v}) \in \mathcal{A}(n, s+1) - \mathcal{B}(n, s+1)$. Since $C \cup (C+\vec{v})$ is not doubly-even, the number of codes in $\mathcal{B}(n,k)$ that contain $C \cup (C+\vec{v})$ is zero. Therefore, the number of codes in $\mathcal{A}(n,k) - \mathcal{B}(n,k)$ that contain $C \cup (C+\vec{v})$ is equal to the number of codes in $\mathcal{A}(n,k)$ that contain $C \cup (C+\vec{v})$. Let f denote the number of codes in $\mathcal{A}(n,k)$ that contain the $(n, s+1)$ self-orthogonal code $C \cup (C+\vec{v})$. By Lemma 2.11, we know f is the same for any $\vec{v} \in H$. Thus, $\sum_{\vec{v}\in H} x(\vec{v}) = |H|f$.

Let us now consider $\sum_{D\in T} y(D)$. Let $\vec{v}$ be any vector in D in which $w(\vec{v}) \equiv 2 \bmod 4$. Then, since $C \subset D \subset C^{\perp}$, we know $\vec{v} \in C^{\perp}$, and thus $\vec{v} \in H$. Furthermore, since $\vec{v} \in D$ and $C \subset D$, we know $C \cup (C+\vec{v}) \subseteq D$. Therefore, the number of vectors $\vec{v} \in H$ in which $C \cup (C+\vec{v}) \subseteq D$ is equal to the number of vectors in D with weight $\equiv 2 \bmod 4$. By Lemma 2.8, since D is not doubly-even, the number of such vectors in D is 2^{k-1}. Thus, $\sum_{D\in T} y(D) = |T|2^{k-1}$.

Therefore, $|T|2^{k-1} = |H|f$, which implies:

$$|T| = \frac{|H|f}{2^{k-1}}$$

Lemma 2.11 gives us $f = \sum_{i=1}^{k-(s+1)} (2^{n-2(k-i)-1} - 1)/(2^i - 1)$. Lemma 2.27 gives us $|H| = 2^{n-s-2} - 2^{((n-1)/2)-1}$. Thus:

$$\begin{aligned}
|T| &= \frac{(2^{n-s-2} - 2^{\frac{n-1}{2}-1}) \sum_{i=1}^{k-(s+1)} (2^{n-2(k-i)-1} - 1)/(2^i - 1)}{2^{k-1}} \\
&= \frac{2^{n-s-2} - 2^{\frac{n-1}{2}-1}}{2^{k-1}} \sum_{i=1}^{k-s-1} \frac{2^{n-2(k-i)-1} - 1}{2^i - 1} \\
&= (2^{n-s-k-1} - 2^{\frac{n-1}{2}-k}) \sum_{i=1}^{k-s-1} \frac{2^{n-2(k-i)-1} - 1}{2^i - 1} \\
&= 2^{\frac{n-1}{2}-k} (2^{\frac{n-1}{2}-s} - 1) \sum_{i=1}^{k-s-1} \frac{2^{n-2(k-i)-1} - 1}{2^i - 1}
\end{aligned}$$

□

We are now ready to count, for a given (n, s) doubly-even code C, the number of (n, k) doubly-even self-orthogonal codes that contain C:

Theorem 2.29. *Let $C \in \mathcal{B}(n, s)$, where $n \equiv \pm 1 \bmod 8$ and $0 \leq s < (n-1)/2$. Then the number of codes in $\mathcal{B}(n, k)$, $s < k \leq (n-1)/2$, that contain C is:*

$$\left(\frac{2^{n-2s-1}-1}{2^{k-s}-1} - 2^{\frac{n-1}{2}-k}(2^{\frac{n-1}{2}-s}-1)\right)\prod_{i=1}^{k-s-1}\frac{2^{n-2(k-i)-1}-1}{2^i-1}.$$

Proof. The number of codes in $\mathcal{B}(n, k)$ that contain C is equal to the number of codes in $\mathcal{A}(n, k)$ that contain C minus the number of codes in $\mathcal{A}(n, k) - \mathcal{B}(n, k)$ that contain C. Lemma 2.11 gives us the number of codes in $\mathcal{A}(n, k)$ that contain C. Lemma 2.28 gives us the number of codes in $\mathcal{A}(n, k) - \mathcal{B}(n, k)$ that contain C. This and some algebraic manipulation gives us our result. □

We have now found, for a given code C in $\mathcal{B}(n, s)$, the number of codes in $\mathcal{B}(n, k)$ that contain C. We can use this result to count the number of (n, k) doubly-even self-orthogonal codes:

Lemma 2.30. *Let $n \equiv \pm 1 \bmod 8$. The number of (n, k) doubly-even self-orthogonal codes is:*

$$\left(\frac{(2^{(n-1)/2}-1)(2^{((n-1)/2)-k}+1)}{2^k-1}\right)\prod_{i=1}^{k-1}\frac{2^{n-2(k-i)-1}-1}{2^i-1}.$$

Proof. Every (n, k) doubly-even self-orthogonal code contains the *empty code* of dimension 0. Setting s to 0 in Theorem 2.29 and some algebraic manipulation gives us our result. □

This result can be used to count the number of $(n, \lfloor n/2 \rfloor)$ doubly-even self-orthogonal codes:

Lemma 2.31. *Let $n \equiv 0, \pm 1 \bmod 8$. Then the number of $(n, \lfloor n/2 \rfloor)$ doubly-even self-orthogonal codes over $GF(2)$ is:*

$$2 \prod_{i=1}^{\lceil n/2 \rceil - 2} (2^i + 1).$$

Proof. For the case of $n \equiv 0 \bmod 8$, see [14]. Setting k to $(n-1)/2$ in Lemma 2.30 and some algebraic manipulation gives us our result for the case of $n \equiv \pm 1 \bmod 8$. □

We are now ready to count the number of $(n, (n-1)/2)$ doubly-even self-orthogonal codes over $GF(2)$ that do not have a coordinate of zeros.

Theorem 2.32. *Let $n \equiv 1 \bmod 8$. The number of $(n, (n-1)/2)$ doubly-even self-orthogonal codes over $GF(2)$ that do not have a coordinate of zeros is:*

$$(2^{\frac{n-1}{2}} - 2(n-1)) \prod_{i=1}^{\frac{n-1}{2}-2} (2^i + 1).$$

Proof. Let X denote the number of $(n, (n-1)/2)$ doubly-even self-orthogonal codes over $GF(2)$. Let Y denote the number of such codes that have a coordinate of zeros. Then the number of $(n, (n-1)/2)$ doubly-even self-orthogonal codes that do not have a coordinate of zeros is equal to $X - Y$. Lemma 2.31 gives us X. Thus, all we need to find is Y.

Let C be an $(n, (n-1)/2)$ doubly-even self-orthogonal code over $GF(2)$ that has x coordinates of zeros. Let C' be the $(n-x, (n-1)/2)$ code that is obtained by deleting the x coordinates of zeros from C. Then C' is a doubly-even self-orthogonal code. Since C' is self-orthogonal, we must have $((n-1)/2) \leq (n-x) - ((n-1)/2)$, which implies $n - 1 \leq n - x$, which implies $x = 1$. Therefore, C cannot have more that one coordinate consisting entirely of zeros, and thus, C' must be an $(n-1, (n-1)/2)$ doubly-even *self-dual* code.

Thus, if C is an $(n,(n-1)/2)$ doubly-even self-orthogonal code that has a coordinate of zeros then if we remove the coordinate of zeros from C, the resulting code is an $(n-1,(n-1)/2)$ doubly-even self-dual code C'. Conversely, if C' is an $(n-1,(n-1)/2)$ doubly-even self-dual code then if we insert a coordinate of zeros anywhere in C', the resulting code C is an $(n,(n-1)/2)$ doubly-even self-orthogonal code. Thus, Y is equal to the number of $(n-1,(n-1)/2)$ doubly-even self-dual codes multiplied by the number of ways in which we can insert a coordinate of zeros into such a code.

Since for any particular $(n-1,(n-1)/2)$ doubly-even self-dual code C', there are n locations in C' in which we can insert a coordinate of zeros, we have $Y = nZ$, where Z is the number of $(n-1,(n-1)/2)$ doubly-even self-dual codes over $GF(2)$. Substituting $n-1$ for n in Lemma 2.31 gives us Z.

Thus, the number of $(n,(n-1)/2)$ doubly-even self-orthogonal codes that do not have a coordinate of zeros is:

$$\begin{aligned}
X - nZ &= 2\prod_{i=1}^{\frac{n+1}{2}-2}(2^i+1) - 2n\prod_{i=1}^{\frac{n-1}{2}-2}(2^i+1) \\
&= 2(2^{\frac{n+1}{2}-2}+1)\prod_{i=1}^{\frac{n-1}{2}-1}(2^i+1) - 2n\prod_{i=1}^{\frac{n-1}{2}-2}(2^i+1) \\
&= 2(2^{\frac{n+1}{2}-2}+1-n)\prod_{i=1}^{\frac{n-1}{2}-2}(2^i+1) \\
&= (2^{\frac{n-1}{2}}-2(n-1))\prod_{i=1}^{\frac{n-1}{2}-2}(2^i+1)
\end{aligned}$$

□

Using Theorem 2.32, we have computed the theoretical number of $(33,16)$ doubly-even self-orthogonal codes that do not have a coordinate of zeros. The total computed matched the number of such codes we computed using the sizes of the automorphism groups of the codes in our list L_C, that we enumerated using Algorithm 2.26, thus assuring us our enumeration was correct.

2.6 Concluding Remarks

We will now conclude this chapter by reviewing the main results presented in the chapter.

Using the methods of [2], we have enumerated a list L_D of inequivalent $(34, 17, d \geq 4)$ self-orthogonal codes over $GF(2)$. Using the list L_D, we have enumerated a list L_C of inequivalent $(33, 16)$ doubly-even self-orthogonal codes over $GF(2)$ that do not have a coordinate of zeros. The list L_D contains 20852 codes. The list L_C contains 594 codes. We have also found the automorphism groups of the codes in L_D and L_C. Generators for the codes in L_C, along with their automorphism groups, are available on the world wide web at: *www.cs.umanitoba.ca/~umbilou1/DoublyEvenCodes/*

Using the methods of MacWilliams, Sloane, and Thompson in [14], we have counted the theoretical number of $(33, 16)$ doubly-even self-orthogonal codes over $GF(2)$ that do not have a coordinate of zeros. Together with the automorphism groups, we have used this result to check that we did not make any mistakes in our enumeration of the list L_C.

Using the methods in [14], we have also proved that any $(33, k)$ doubly-even self-orthogonal code over $GF(2)$ is contained in a $(33, 16)$ doubly-even self-orthogonal code over $GF(2)$. Since the point code of a $(22, 33, 12, 8, 4)$-BIBD is a $(33, k)$ doubly-even self-orthogonal code over $GF(2)$, that does not have a coordinates of zeros, this tells us that if a $(22, 33, 12, 8, 4)$-BIBD exists, then our list L_C of 594 inequivalent $(33, 16)$ doubly-even self-orthogonal codes must contain a code that contains the incidence matrix of such a design.

Chapter 3

BIBD Search: Eliminating Codes Theoretically

We now have a list L_C of 594 $(33, 16)$ doubly-even self-orthogonal codes with the property that if a $(22, 33, 12, 8, 4)$-BIBD exists then there exists a code in L_C that contains the incidence matrix of such a design. In the next two chapters, we will consider the problem of determining whether or not a given code in L_C contains such an incidence matrix. In this chapter, we will look at how we can prove that certain classes of codes in L_C cannot contain the incidence matrix of such a design. For each code C in L_C that we cannot eliminate theoretically, we must search the weight 12 words of C for the incidence matrix. How we perform this search is the subject of Chapter 4.

Since our approach to determining whether or not a $(22, 33, 12, 8, 4)$-BIBD exists is based on the fact that if it does then its incidence matrix generates a $(33, 16)$ doubly-even self-orthogonal code, we will begin this chapter by giving a formal proof of this fact in Section 3.1. In Section 3.2, we will look at the weight distribution of the $(33, 16)$ doubly-even self-orthogonal codes. In Section 3.3, we will introduce the concept of a "weight 4 block." Simply put, a weight 4 block is a set of weight 4 words, with certain properties, that may occur in a self-orthogonal code. In Section 3.4, we will classify the $(33, 16)$

doubly-even self-orthogonal codes based on the weight 4 blocks they contain. We will use the weight 4 blocks they contain to prove that certain classes of codes in L_C cannot contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD.

In Section 3.5, we will give a general outline of the method we use to prove that a code cannot contain an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD. In Sections 3.6 and 3.7, we will describe some of the *tools* we will need for our proofs. In Section 3.6, we will have a look at a standard method we use to prove that any matrix that contains a certain submatrix cannot be the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. In Section 3.7, we will have a look at the different possibilities for the four columns in an incidence matrix that are supported by a weight 4 word.

In Sections 3.8–3.10, we will prove that three different classes of codes in L_C cannot contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. In Section 3.8, we will prove that the class of codes we refer to as the e_3-codes cannot contain an incidence matrix. In Section 3.9, we will prove that the class of codes we refer to as the e_{2i}-codes, for $i \geq 2$, cannot contain an incidence matrix. Finally, in Section 3.10, we will prove that the class of codes we refer to as the d_i-codes, for $i \geq 5$, cannot contain an incidence matrix. The number of codes in L_C that fall into one of these three classes is 116. Therefore, these proofs eliminate 116 of the codes in L_C, leaving us with $594 - 116 = 478$ codes to search.

In Section 3.11, we will briefly discuss the classes of codes in L_C that we have not been able to eliminate theoretically. Finally, in Section 3.12, we will conclude this chapter with a brief review of the main results presented in the chapter.

3.1 The Point Code of a $(22, 33, 12, 8, 4)$-BIBD

Our approach for investigating whether or not a $(22, 33, 12, 8, 4)$-BIBD exists is based on the fact that its point code (i.e. the code generated by the rows of its incidence matrix) is a doubly-even self-orthogonal code with length 33. Therefore, in this section, we will

give a formal proof of this fact.

Theorem 3.1. *Let A be the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD B. Let C be the point code of B. Let C_m denote the set of all codewords in C that are the sum modulo 2 of at most m rows in A, where $0 \leq m \leq 22$. Then the codewords in C_m all have weight $\equiv 0 \bmod 4$. Furthermore, if $\vec{a}$ is a row in A and $\vec{c} \in C_m$ then $\vec{a}$ and $\vec{c}$ intersect in an even number of positions.*

Proof. By induction on m.

If $m = 0$ then C_m consists of the all zeros vector $\vec{0}$. Since $w(\vec{0}) = 0$, $\vec{0}$ has weight $\equiv 0 \bmod 4$. Furthermore, since $\vec{0}$ intersects each row in A in zero positions, $\vec{0}$ intersects each row in A in an even number of positions.

Assume our proposition is true for $k = m$, where $0 \leq m < 22$. Consider the set C_{k+1}. Let $\vec{c} \in C_{k+1}$. If $\vec{c}$ is also in C_k then, by our induction hypothesis, $\vec{c}$ has weight $\equiv 0 \bmod 4$ and intersects every row in A in an even number of positions. Suppose $\vec{c} \notin C_k$. Then $\vec{c} = \vec{a} + \vec{c}_k$, where $\vec{a}$ is a row in A and $\vec{c}_k$ is the sum of k other rows in A. Consider the three different non-zero possibilities for a component in $\vec{a}$ and $\vec{c}_k$:

$\vec{a}$	1	1	0
$\vec{c}_k$	1	0	1
	x_3	x_2	x_1

Let x_i, where $1 \leq i \leq 3$, denote the number of times each of these possibilities occur. Then $w(\vec{a}) = x_3 + x_2$, $w(\vec{c}_k) = x_3 + x_1$, and $w(\vec{c}) = x_2 + x_1$. Since both $\vec{a}$ and $\vec{c}_k$ are the sum of $\leq k$ rows of A, we know $\vec{a}, \vec{c}_k \in C_k$. Therefore, by our induction hypothesis, $x_3 + x_2 \equiv 0 \bmod 4$ and $x_3 + x_1 \equiv 0 \bmod 4$. Furthermore, since $\vec{a}$ is a row in A, by our induction hypothesis, $\vec{a}$ and $\vec{c}_k$ intersect in an even number of positions. Therefore,

$x_3 \equiv 0 \bmod 2$, which implies $2x_3 \equiv 0 \bmod 4$. Thus:

$$\begin{aligned} w(\vec{c}) &\equiv x_2 + x_1 && \bmod 4 \\ &\equiv 4x_3 + x_2 + x_1 && \bmod 4 \\ &\equiv 2x_3 + (x_3 + x_2) + (x_3 + x_1) && \bmod 4 \\ &\equiv 0 + 0 + 0 && \bmod 4 \\ &\equiv 0 && \bmod 4. \end{aligned}$$

Furthermore, if $\vec{r}$ is any row in A then, by our induction hypothesis, $\vec{r}$ intersects both $\vec{c}_k$ and $\vec{a}$ in an even number of positions, which implies:

$$\begin{aligned} \vec{c} \bullet \vec{r} &= (\vec{c}_k + \vec{a}) \bullet \vec{r} \\ &= (\vec{c}_k \bullet \vec{r}) + (\vec{a} \bullet \vec{r}) \\ &= 0 + 0 = 0. \end{aligned}$$

Therefore, the codewords in C_{k+1} have weight $\equiv 0 \bmod 4$, and, each codeword in C_{k+1} intersect each row in A in an even number of positions.

Thus, by the principle of mathematical induction our proposition is true. □

Corollary 3.2. *The codewords in C have weight $\equiv 0 \bmod 4$.*

Corollary 3.3. *Every pair of codewords in C intersect in an even number of positions.*

Proof. Let $\vec{u}, \vec{v} \in C$. Consider the three different non-zero possibilities for the components of $\vec{u}$ and $\vec{v}$:

$\vec{u}$	1	1	0
$\vec{v}$	1	0	1
	x_3	x_2	x_1

Since $\vec{u}$, $\vec{v}$, and $\vec{u} + \vec{v}$ all have weight $\equiv 0 \bmod 4$, we have $x_3 + x_2 \equiv x_3 + x_1 \equiv x_2 + x_1 \equiv 0 \bmod 4$. Therefore, $(x_3 + x_2) + (x_3 + x_1) \equiv 0 \bmod 4$, which implies $2x_3 + (x_2 + x_1) \equiv 0 \bmod 4$, which implies $2x_3 \equiv 0 \bmod 4$, and thus $x_3 \equiv 0 \bmod 2$. Therefore, $\vec{u}$ and $\vec{v}$ intersect in an even number of positions. □

Corollary 3.4. *C is a doubly-even self-orthogonal code.*

Thus, the point code C of a $(22, 33, 12, 8, 4)$-BIBD is a doubly-even self-orthogonal code with length 33.

3.2 The Weight Distribution of the $(33, 16)$ Codes

In this section, we will have a look at the weight distribution of both the $(33, 16)$ doubly-even self-orthogonal codes that do not have a coordinate of zeros, and the orthogonal complement of such codes.

Throughout this section, we will use the following notation: We will use D to denote a $(34, 17, d \geq 4)$ self-dual code that has a *doubly-even coordinate*, as defined in Section 2.4. We will use j to denote the doubly-even coordinate in D. Finally, we will use C to denote the $(33, 16)$ doubly-even self-orthogonal code that is formed by taking the cross-section of D at coordinate j. Of course, as we saw in Section 2.4, since D has distance ≥ 4, C does not have a coordinate of zeros.

In order to determine the general properties of the weight distributions of both C and $C^{\perp}$, we will first give a theorem that relates the codewords of $C^{\perp}$ to the codewords of D.

Theorem 3.5. *The orthogonal complement $C^{\perp}$ is equal to D with coordinate j deleted.*

Proof. Let C' denote the $(33, 17)$ code that consists of the codewords in D with component j deleted. Let $\vec{u}$ be any codeword in C' and let $\vec{v}$ be the corresponding codeword in D. If $\vec{v}$ has a value of 0 in component j then $\vec{u} \in C$ which implies $\vec{u} \in C^{\perp}$. If $\vec{v}$ has a value of 1 in component j then $\vec{u}$ must be orthogonal to every codeword in C since the codewords in D that correspond to the codewords in C all have a value of 0 in component j. Therefore, if $\vec{v}$ has a value of 1 in component j then $\vec{u} \in C^{\perp}$. Thus, $C' \subseteq C^{\perp}$. Since both C' and $C^{\perp}$ have dimension 17, this implies $C' = C^{\perp}$. □

Let us now apply what we know about the codewords in D to the codewords in $C^{\perp}$ and C. Since D is a self-dual code and j is a doubly-even coordinate, we know, by Theorem 2.19, that D contains 2^{16} codewords with a value of 0 in component j and 2^{16} codewords with a value of 1 in component j. Furthermore, the 2^{16} codewords with a value of 0 in component j have weight $\equiv 0 \bmod 4$, and the 2^{16} codewords with a value of 1 in component j have weight $\equiv 2 \bmod 4$. Therefore, by Theorem 3.5, we know the following about the codewords in $C^{\perp}$:

- $C^{\perp}$ contains 2^{16} codewords with weight $\equiv 0 \bmod 4$,
- $C^{\perp}$ contains 2^{16} codewords with weight $\equiv 1 \bmod 4$ (i.e. the weight $\equiv 2 \bmod 4$ codewords in D with component j deleted), and thus
- $C^{\perp}$ does not contain any codewords with weight $\equiv 2, 3 \bmod 4$.

Since the codewords in C are the codewords in D that have a value of 0 in component j, with component j deleted, we also know the 2^{16} codewords in $C^{\perp}$ with weight $\equiv 0 \bmod 4$ is C. In other words, if $\vec{c} \in C^{\perp}$ then $\vec{c} \in C$ if and only if $w(\vec{c}) \equiv 0 \bmod 4$.

We will now use D to show that the number of weight $4i$ words in $C^{\perp}$ is equal to the number of $33 - 4i$ words in $C^{\perp}$. Let S_i denote the number of weight i words in D. Then, since D contains the all ones vector, $|S_{4i}| = |S_{34-4i}|$. Furthermore, since $34 - 4i \equiv 2 \bmod 4$, the words in S_{34-4i} correspond to the weight $33 - 4i$ words in $C^{\perp}$. Therefore, the number of weight $4i$ words in $C^{\perp}$ are equal to the number of weight $33 - 4i$ words in $C^{\perp}$.

Since D has distance ≥ 4, we know D does not contain any weight 2 words, which implies D also does not contain any weight $34 - 2 = 32$ words. Therefore, $C^{\perp}$ does not contain any weight $2 - 1 = 1$ words, nor any weight 32 words.

This concludes our look at the general properties of the weight distribution of C and $C^{\perp}$. In Appendix A, we give tables listing all the different weight distributions for the

594 equivalence classes of $(33, 16)$ doubly-even self-orthogonal codes that do not have a coordinate of zeros. The weight distribution of the orthogonal complement of the codes can be easily obtained from the tables using the fact that the number of weight $33 - 4i$ words in $C^{\perp}$ is equal to the number of weight $4i$ words in C.

3.3 The Weight 4 Blocks of a Code

In this section, we will have a look at the different ways in which the weight 4 words may *interact* in the $(33, 16)$ doubly-even self-orthogonal codes.

We begin by defining a structure we call a *weight 4 block*, which we use to describe the weight 4 words that may occur in a given code.

Definition 3.6. Let C be a self-orthogonal code with distance 4. A *weight 4 block* W in C is a set of weight 4 words in C with the following two properties: (1) if $\vec{c}$ is a weight 4 word in C that is *not* in W then $\vec{c}$ does not intersect any of the weight 4 words in W, and (2) W cannot be partioned into two sets, W_0 and W_1, in which the words in W_0 do not intersect any of the words in W_1.

Example 3.7. Let C be a self-orthogonal code with distance 4. Suppose C contains the following weight 4 words:

$$
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0^* \\
1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0^* \\
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0^* \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0^* \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0^*
\end{array}
$$

Then the weight 4 blocks in C are:

$$W_1 = \left\{ \begin{array}{cccccccccccccccc} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0^* \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0^* \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0^* \end{array} \right\}$$

$$W_2 = \left\{ \begin{array}{cccccccccccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0^* \end{array} \right\}$$

$$W_3 = \left\{ \begin{array}{cccccccccccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0^* \end{array} \right\}$$

We typically represent our weight 4 blocks with a *generator block*, which we define next.

Definition 3.8. Let C be a self-orthogonal code with distance 4. Let W be a weight 4 block in C. A *generator block* for W is any set S of codewords in C with the property that a weight 4 word $\vec{w}$ can be generated by the words in S if and only if $\vec{w}$ is an element of W.

Example 3.9. Consider the weight 4 block W_1 in Example 3.7. The set:

$$S_1 = \left\{ \begin{array}{cccccccccccccccc} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0^* \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0^* \end{array} \right\}$$

is a generator block for W_1.

Any weight 4 block can be generated by a generator block S that consists entirely of weight 4 words. Furthermore, any such generator block S also generates a self-orthogonal code. Therefore, we can find generators for all the inequivalent weight 4 blocks by finding all the inequivalent self-orthogonal codes that are generated by weight 4 words only. Pless and Sloane [20] have found all the inequivalent self-orthogonal codes that can be generated by weight 4 words only. They are the e_7, E_8, and d_{2n}-codes, for $n \geq 2$. We

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

the e_3-block

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

the e_4-block

$$\begin{matrix} & & \overbrace{\quad\quad\quad\quad\quad\quad} & & & & & & \\ 1 & 1 & 1 & 1 & 0 & 0 & \cdots & 0 & 0 & 0^* \\ 1 & 1 & 0 & 0 & 1 & 1 & \cdots & 0 & 0 & 0^* \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 1 & 1 & 0 & 0 & 0 & 0 & \cdots & 1 & 1 & 0^* \end{matrix} \Bigg\} i$$

the d_i-block, $i \geq 1$

Figure 3.1: The weight 4 blocks.

represent our weight 4 blocks with these codes. However, whereas Pless and Sloane subscript their codes with their length, we use the dimension of the code to subscript our weight 4 blocks. Thus, we refer to their e_7, E_8, and d_{2n}-codes as e_3, e_4, and d_{n-1}-blocks, respectively. Generator blocks for the e_3, e_4, and d_i-blocks, for $i \geq 1$, are depicted in Figure 3.1. Note that we will use the notation e_3, e_4, and d_i to refer to *any* weight 4 block that is equivalent to the corresponding weight 4 block in Figure 3.1. For example, any weight 4 block that is equivalent to an e_3-block will also be referred to as an e_3-block.

We distinguish a special case of the d_{2i-1}-blocks, for $i \geq 2$, in which we attach the vector 1 0 1 0...1 0 to the generator block for the d_i-block depicted in Figure 3.1. We refer to such blocks as e_{2i}-blocks. A generator block for the e_{2i}-blocks, $i \geq 2$, is given in Figure 3.2. The reason we distinguish these blocks is that when we remove the coordinates of zeros from the blocks, we get a block that generates a $(4i, 2i)$ self-dual code. Note that for the case of $i = 2$, the generator block given in Figure 3.2 generates a weight 4 block

$$
\begin{array}{cccccccccc}
 & & \overbrace{\qquad\qquad\qquad\qquad\qquad\qquad}^{4i-2} & & & & & & & \\
1 & 1 & 1 & 1 & 0 & 0 & \cdots & 0 & 0 & 0^* \\
1 & 1 & 0 & 0 & 1 & 1 & \cdots & 0 & 0 & 0^* \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\
1 & 1 & 0 & 0 & 0 & 0 & \cdots & 1 & 1 & 0^* \\
1 & 0 & 1 & 0 & 1 & 0 & \cdots & 1 & 0 & 0^*
\end{array}
\left.\vphantom{\begin{array}{c}a\\a\\a\\a\end{array}}\right\} 2i-1
$$

Figure 3.2: the e_{2i}-blocks, $i \geq 2$.

that is equivalent to the e_4-block given in Figure 3.1. This can be seen by adding the last row of the generator block for the e_4-block in Figure 3.1 to the second last row and then performing an appropriate coordinate permutation, producing the generator block for the e_4-block in Figure 3.2.

3.4 Classifying the Self-Orthogonal Codes

We will now classify the self-orthogonal codes with distance 4 based on the weight 4 blocks they contain.

Since any self-orthogonal code with distance 4 may contain more than one weight 4 block, we will first define an order of precedence for the different weight 4 blocks. In descending order, this precedence is:

$$e_{2i}, \ldots, e_8, e_6, e_4, e_3, \ldots, d_i, \ldots, d_3, d_2, d_1.$$

We classify our self-orthogonal codes based on the highest precedence block they contain.

Example 3.10. Suppose C is a self-orthogonal code that contains an e_4-block, two e_3-blocks, and a d_5-block. Then, since the e_4-block is the highest precedence block in C, we classify C as an e_4-code.

code class	number of codes in class	code class	number of codes in class
e_8	1	d_5	14
e_4	13	d_4	37
e_3	76	d_3	93
d_{10}	1	d_2	205
d_8	2	d_1	132
d_7	2	distance 8	11
d_6	7		

Table 3.1: The number of inequivalent $(33, 16)$ doubly-even self-orthogonal codes in each class of codes.

In Table 3.1, we list the number of inequivalent $(33, 16)$ doubly-even self-orthogonal codes, that do not have a coordinate of zeros, that are in each class of codes. Also listed in the table is the number of codes with distance > 4, all of which have distance 8.

3.5 Our Theoretical Approach to Eliminating Codes

In this section, we will give a general outline of the method we use to prove that any code that contains a certain structure, such as a particular weight 4 block, cannot contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD.

Let W be a particular set of codewords that may occur in the orthogonal complement of a $(33, 16)$ doubly-even self-orthogonal code. Let C be any $(33, 16)$ doubly-even self-orthogonal code in which $C^{\perp}$ contains W. Our method for proving that any such C cannot contain the incidence matrix A of a $(22, 33, 12, 8, 4)$-BIBD is to first assume that C does contain such an A, and then, using the fact that $C^{\perp}$ contains W, prove that our

assumption is wrong.

Our method for disproving our assumption works as follows: We begin by assuming C contains 22 weight 12 words that form the incidence matrix A of a $(22, 33, 12, 8, 4)$-BIBD. Let S denote the set of columns in A that are supported by W. Without loss of generality, we assume S occurs in the first $|S|$ columns of A. We then find a list L of $i \times |S|$ matrices Y, where $i \leq 22$, with the property that the first $|S|$ columns of any such A must contain, up to row and column rearrangement, one of the matrices in Y. We find the list L using facts such as C is a doubly-even self-orthogonal code, $C^{\perp}$ contains the set of codewords W, and, under our assumption, A is the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. For each Y in L, we then show that Y cannot be a submatrix in the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. This tells us that our assumption that C does contain an incidence matrix A was wrong, and thus, any C in which $C^{\perp}$ contains W cannot contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD.

Once we are able to prove that any code C, in which $C^{\perp}$ contains a certain structure W, cannot contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD, we can then eliminate all such codes from our list of 594 inequivalent $(33, 16)$ doubly-even self-orthogonal codes.

3.6 The Column Weight Solutions Algorithm

One common method we use to show that a matrix Y cannot be a submatrix in the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD is a generalization of the method used by Hamada and Kobayashi [9] to produce the block intersection patterns given in Theorem 1.8.

Suppose we are given an $n \times m$ matrix Y, where $1 \leq n \leq 22$ and $1 \leq m < 33$. We assume Y is a submatrix of a 22×33 matrix A that is the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. Without loss of generality, we assume Y is located in the upper left corner of A, and we let Z denote the $n \times (33 - m)$ submatrix of A that is adjacent to Y, as in Figure 3.3.

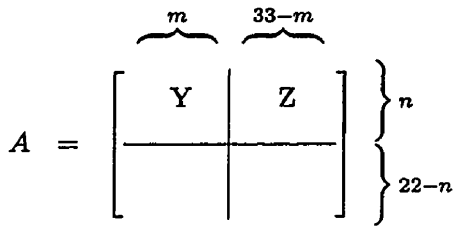

Figure 3.3: The incidence matrix A, of a $(22, 33, 12, 8, 4)$-BIBD, that contains the $n \times m$ submatrix Y in its upper left corner.

Let z_j denote the number of columns in Z with weight j. Since the columns of A have weight 8, the columns of Z have weight in the range $[j_{min}, j_{max}]$, where $j_{min} = \max(0, 8-(22-n))$ and $j_{max} = \min(8, n)$. Using the known matrix Y and our assumption that the matrix A is the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD, we will now find three equations that the unknowns $z_{j_{min}}, z_{j_{min}+1}, \ldots, z_{j_{max}}$ must satisfy.

Since the matrix Z consists of $33 - m$ columns, we must have:

$$\sum_{j=j_{min}}^{j_{max}} z_j = 33 - m. \tag{3.1}$$

Let r_i denote the weight of row i in Y, for $1 \leq i \leq n$. Then, since the rows of A have weight 12, row i of Z must have weight $12 - r_i$. Therefore, the number of ones in the matrix Z is equal to $\sum_{i=1}^{n}(12 - r_i)$. Since $\sum_{j=j_{min}}^{j_{max}} j z_j$ must also sum to the number of ones in Z, we must have:

$$\sum_{j=j_{min}}^{j_{max}} j z_j = \sum_{i=1}^{n}(12 - r_i). \tag{3.2}$$

Let s_{i_1,i_2} denote the number of columns in rows i_1 and i_2 of Y in which both rows have a value of 1, for $1 \leq i_1 < i_2 \leq n$. Then, since rows i_1 and i_2 of A intersect in 4 positions, the number of columns in rows i_1 and i_2 of Z in which both rows have a value of 1 is $4 - s_{i_1,i_2}$. Therefore, the sum, over all pairs of rows in Z, of the number of columns

in which a pair of rows both have a value of 1 is equal to $\sum_{i_1=1}^{n-1}\sum_{i_2=i_1+1}^{n}(4-s_{i_1,i_2})$. Since, for a given column c in Z with weight j, there are $\binom{j}{2}$ pairs of rows in Z that both have a value of 1 in column c, $\sum_{j=j_{min}}^{j_{max}}\binom{j}{2}z_j$ must also be equal to the sum, over all pairs of rows in Z, of the number of columns in which a pair of rows both have a value of 1. Thus, we must have:

$$\sum_{j=j_{min}}^{j_{max}}\binom{j}{2}z_j=\sum_{i_1=1}^{n-1}\sum_{i_2=i_1+1}^{n}(4-s_{i_1,i_2}). \tag{3.3}$$

Since we are given the matrix Y, and thus, the integers m, n, r_i, and s_{i_1,i_2}, the only unknowns in Equations 3.1– 3.3 are the variables z_j. Using integer linear programming, we can find all non-negative integer solutions for the unknowns z_j. If we find there are no non-negative integer solutions, we then know the matrix Z cannot exist, and thus, we know that any matrix A that contains the submatrix Y cannot be the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD.

We summarize this procedure in Algorithm 3.11. The algorithm takes as input the matrix Y and returns the set of all non-negative integer solutions to Equations 3.1–3.3. We will refer to these solutions as *column weight solutions* for the matrix Z adjacent to Y. If no column weight solutions exists, the algorithm returns the empty set.

Algorithm 3.11. *ColumnWeightSolutions*(Y, S):

- Input: an $n \times m$ matrix Y, where $1 \leq n \leq 22$ and $1 \leq m < 33$.
- Output: the (possibly empty) set S of non-negative integer solutions to Equations 3.1–3.3 for the unknowns $z_{j_{min}}, z_{j_{min}+1}, \ldots, z_{j_{max}}$, where $j_{min} = \max(0, n-14)$ and $j_{max} = \min(8, n)$. If there exists a matrix A that contains Y, in which A is the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD, then the column weight distribution of the matrix Z, adjacent to Y in A, is an element of the set S of column weight solutions returned by the algorithm.

begin

for $i = 1$ to n **do**

Let r_i = the weight of row i in Y.

for $i_1 = 1$ to $n - 1$ **do**

for $i_2 = i_1 + 1$ to n **do**

Let s_{i_1,i_2} = the number of columns of Y in which both rows i_1 and i_2 have a value of 1.

Using r_i, s_{i_1,i_2}, m, and n, evaluate the right hand sides of Equations 3.1–3.3.

Using an integer linear program, find all non-negative integer solutions for the unknowns in Equations 3.1–3.3.

Return S, the set of all solutions found by the linear program.

end

3.7 The Columns Supported by a Weight 4 Word

In the remainder of this chapter, we will prove that any $(33, k, 4)$ doubly-even self-orthogonal code that contains either an e_3-block, an e_{2i}-block, $i \geq 2$, or a d_i-block, $i \geq 5$, cannot contain an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD. Towards this goal, we will first determine all the different possibilities for the columns of the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD that are supported by a weight 4 word.

Theorem 3.12. *Let C be a $(33, k, 4)$ doubly-even self-orthogonal code that contains an incidence matrix A for a $(22, 33, 12, 8, 4)$-BIBD B. Let $\vec{w}$ be a weight 4 word in $C^{\perp}$. Let Y denote the 22×4 matrix consisting of the 4 columns in A that are supported by $\vec{w}$. Then the rows and columns of Y are a permutation of the rows and columns of one of the three matrices given in Figure 3.4.*

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

Figure 3.4: The three different possibilities, up to row and column rearrangement, for the 4 columns in the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD that are supported by a weight 4 word.

Proof. Let $S_4 = YY^T$. Then, as Greig shows us in Theorem 1.11, the only unique possibilities for S_4, up to row and column rearrangement of Y, are:

$$\begin{bmatrix} 8 & 4 & 3 & 1 \\ 4 & 8 & 1 & 3 \\ 3 & 1 & 8 & 4 \\ 1 & 3 & 4 & 8 \end{bmatrix} \quad \begin{bmatrix} 8 & 4 & 2 & 2 \\ 4 & 8 & 2 & 2 \\ 2 & 2 & 8 & 4 \\ 2 & 2 & 4 & 8 \end{bmatrix} \quad \begin{bmatrix} 8 & 3 & 3 & 2 \\ 3 & 8 & 2 & 3 \\ 3 & 2 & 8 & 3 \\ 2 & 3 & 3 & 8 \end{bmatrix}$$

Using the fact that the rows of Y must intersect the weight 4 word $\vec{w}$ in an even number of positions, and thus, must have weight 0, 2, or 4, it can easily be shown that for each of the three S_4's given there exists one and only one Y such that $YY^T = S_4$. This leads us to the three matrices given in Figure 3.4. □

Corollary 3.13. *Let C be a $(33, k, 4)$ doubly-even self-orthogonal code that contains an incidence matrix A for a $(22, 33, 12, 8, 4)$-BIBD. Let $\vec{w}$ be a weight 4 word in $C^{\perp}$. Then the rows of A cannot intersect $\vec{w}$ in 4 positions.*

Proof. Let Y denote the 22×4 matrix consisting of the columns supported by $\vec{w}$. By Theorem 3.12, the only possibilities for Y, up to row and column rearrangement, are the three given in Figure 3.4, none of which have rows with weight 4. □

Note that in the proof of Theorem 3.12, we used Theorem 1.11 to find the different possibilities for the columns of A that are supported by a weight 4 word $\vec{w}$ in $C^{\perp}$. However, Theorem 1.11 is stated in terms of a weight 4 word that is in the orthogonal complement of the *point code* D of B. This, however, is not a problem since C must contain D, which implies $C^{\perp} \subseteq D^{\perp}$, and thus $\vec{w}$ must also be a weight 4 word in $D^{\perp}$.

For future reference purposes, we will now give a theorem (without proof) that tells us there are seven different possibilities, up to row and column rearrangement, for the columns in an incidence matrix that are supported by a weight 5 word. The theorem can be proved in the same manner as Theorem 3.12, using the results of Theorem 1.11.

Theorem 3.14. *Let C be a $(33, k)$ doubly-even self-orthogonal code that contains an incidence matrix A for a $(22, 33, 12, 8, 4)$-BIBD. Let $\vec{w}$ be a weight 5 word in $C^{\perp}$. Let Y denote the 22×5 matrix consisting of the 5 columns in A that are supported by $\vec{w}$. Then there are seven different possibilities, up to row and column rearrangement, for the matrix Y.*

We will now conclude this section by giving an alternate proof to Theorem 3.12 that does not depend on the results of Theorem 1.11 (which required much computational time). Instead, the proof makes use of Algorithm 3.11.

Let C be a $(33, k, 4)$ doubly-even self-orthogonal code that contains an incidence matrix A for a $(22, 33, 12, 8, 4)$-BIBD. Let $\vec{w}$ be a weight 4 word in $C^{\perp}$. Without loss of generality, assume the four ones of $\vec{w}$ occur in the first four coordinates of $C^{\perp}$. Let Y be the 22×4 submatrix in the first four columns of A. Since every row in Y is orthogonal to $\vec{w}$, we have the following eight possibilities for a row in Y:

$$\begin{array}{cccc} 1 & 1 & 1 & 1 \quad x_7 \\ 1 & 1 & 0 & 0 \quad x_6 \\ 1 & 0 & 1 & 0 \quad x_5 \\ 1 & 0 & 0 & 1 \quad x_4 \\ 0 & 1 & 1 & 0 \quad x_3 \\ 0 & 1 & 0 & 1 \quad x_2 \\ 0 & 0 & 1 & 1 \quad x_1 \\ 0 & 0 & 0 & 0 \quad x_0 \end{array}$$

Let x_i, $0 \leq i \leq 7$, denote the number of times the corresponding row occurs in Y. Without loss of generality, we will assume $x_6 \geq x_5 \geq x_4$. (That is, given any Y, we will assume the columns of Y are sorted in such a way that $x_6 \geq x_5 \geq x_4$.) Using the facts that there are 22 rows in Y and each column in Y has weight 8, we get the following set

of equations:

$$\begin{aligned} x_7 + x_6 + x_5 + x_4 + x_3 + x_2 + x_1 + x_0 &= 22 \\ x_7 + x_6 + x_5 + x_4 &= 8 \\ x_7 + x_6 + x_3 + x_2 &= 8 \\ x_7 + x_5 + x_3 + x_1 &= 8 \\ x_7 + x_5 + x_2 + x_1 &= 8 \end{aligned}$$

This system of linear equations reduces to:

$$x_0 = 14 - (x_6 + x_5 + x_4) \tag{3.4}$$

$$x_1 = x_6 \tag{3.5}$$

$$x_2 = x_5 \tag{3.6}$$

$$x_3 = x_4 \tag{3.7}$$

$$x_7 = 8 - (x_6 + x_5 + x_4) \tag{3.8}$$

Using Equations 3.4 and 3.8, along with Algorithm 3.11, we will now find the range of x_7. If we run Algorithm 3.11 with an 8×4 matrix of zeros as input, then the output we get is $S = \emptyset$. This implies the incidence matrix A, and thus Y, cannot contain an 8×4 matrix of zeros. Therefore, we must have $x_0 \leq 7$. This, together with Equation 3.4, implies $x_6 + x_5 + x_4 \geq 7$. This in turn, together with Equation 3.8, implies $x_7 \leq 1$. Therefore, $x_7 = 0$ or $x_7 = 1$.

Suppose $x_7 = 1$. Then, by Equation 3.8, we have $x_6 + x_5 + x_4 = 7$, which, by Equation 3.4, implies $x_0 = 7$. Thus, the matrix A must contain the 8 rows in the matrix given in Figure 3.5.

1	1	1	1	1	1	1	1	1	1	1	0^{21}
0	0	0	0								
0	0	0	0								
0	0	0	0								
0	0	0	0				Z_0				Z_1
0	0	0	0								
0	0	0	0								
0	0	0	0								

Figure 3.5: A subset of the rows of an incidence matrix that contains a row that intersects a weight 4 word in 4 positions. The notation 0^{21} represents a string of 21 zeros. Z_0 is a 7×8 matrix and Z_1 is a 7×21 matrix.

Consider the 7×4 matrix of zeros in the lower left corner of the matrix in Figure 3.5. If we run Algorithm 3.11 on this matrix, then we get the following five solutions as output:

z_7	z_6	z_5	z_4	z_3	z_2	z_1	z_0
0	0	1	0	23	5	0	0
0	0	0	3	20	6	0	0
0	0	0	2	23	3	1	0
0	0	0	1	26	0	2	0
0	0	0	0	28	0	0	1

If the matrix in Figure 3.5 is a submatrix of the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD then one of these five solutions must be the column weight distribution for the matrices Z_0 and Z_1. However, if the columns of Z_0 and Z_1 have one of these five weight distributions, then Z_0 cannot contain more than 27 ones. For example, if the columns of Z_0 and Z_1 have the weights given in the first solution (i.e. 1 column with weight 5, 23 columns with weight 3, and 5 columns with weight 2), then the maximum number of ones that Z_0 may contain is $1 \times 5 + 7 \times 3 = 26$. Similarly, for each of the remaining four

solutions, we also find the maximum number of ones in Z_0 does not exceed 27. Therefore, Z_0 contains at most 27 ones. However, since each of the last 7 rows of the matrix in Figure 3.5 must intersect the first row in 4 positions, each row in Z_0 must have weight 4. This implies the number of ones in Z_0 is $7 \times 4 = 28$, which is a contradiction. Thus, x_7 cannot be 1.

We now know x_7 must be 0, which, together with Equation 3.8, implies $x_6+x_5+x_4 = 8$, which in turn, together with Equation 3.4, implies $x_0 = 6$.

Let us now find all the different possibilities for x_6, x_5, and x_4. If we run Algorithm 3.11 with the following 5×4 matrix as input:

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

the output we get is $S = \emptyset$. This implies A, and thus Y, cannot contain this matrix. Therefore, we must have $x_6 \leq 4$. Together with the fact that $x_6 + x_5 + x_4 = 8$, and our assumption that $x_6 \geq x_5 \geq x_4$, this gives us the following four possibilities for the values of x_6, x_5, and x_4:

x_6	x_5	x_4
4	4	0
4	3	1
4	2	2
3	3	2

Now, if we run Algorithm 3.11 on the following 8×4 matrix:

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

the output we get is $S = \emptyset$. This implies the matrix A, and thus Y, cannot contain this matrix. This eliminates the case of $x_6 = 4$, $x_5 = 4$, and $x_4 = 0$.

Together with Equations 3.5– 3.7 and the facts that $x_7 = 0$ and $x_0 = 6$, this leaves us with the following three solutions for the number of times each of the 8 possible rows occur in Y:

x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
0	4	3	1	1	3	4	6
0	4	2	2	2	2	4	6
0	3	3	2	2	3	3	6

These three solutions lead us to the three matrices given in Theorem 3.12. This completes our proof.

3.8 Eliminating the e_3-Codes

In this section, we will prove that any $(33, k, 4)$ doubly-even self-orthogonal code that contains an e_3-block cannot contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD.

Let C be a $(33, k, 4)$ doubly-even self-orthogonal code that contains an e_3-block. Without loss of generality, assume the e_3-block occurs in the first 7 coordinates of C. In

Figure 3.6, we list all the different possibilities for the values of the components of a codeword in C that are supported by the e_3-block. They are determined by the fact that any codeword in C must be orthogonal to every codeword in the e_3-block.

Suppose C contains the incidence matrix A of a $(22, 33, 12, 8, 4)$-BIBD. Let A_7 denote the 22×7 submatrix in the first 7 columns of A (i.e. the columns of A supported by the e_3-block). Since every column in A has weight 8, the number of ones in A_7 is $8 \times 7 = 56$.

Let us now count the number of ones in A_7 using the fact that the first 7 columns of any row in A must be one of the 16 possibilities given in Figure 3.6. By examining Figure 3.6, one finds that for any codeword in C, the number of ones in the components of the codeword that are supported by the e_3-block is either 0, 3, 4, or 7. However, for each codeword in Figure 3.6 with weight 4 or 7 in the components supported by the e_3-block, there exists at least one weight 4 word in the e_3-block that intersects the codeword in 4 positions. For example, $\vec{c}_2$ intersects $\vec{g}_1$ in 4 positions, $\vec{c}_8$ intersects $\vec{g}_1 \oplus \vec{g}_2 \oplus \vec{g}_3$ in 4 positions, and $\vec{c}_{10}$ intersects $\vec{g}_2 \oplus \vec{g}_3$ in 4 positions. By Corollary 3.13, none of the weight 12 words in the incidence matrix A can intersect any of the weight 4 words in the e_3-block in 4 positions. Therefore, the codewords in Figure 3.6 that have weight 4 or 7 in the components supported by the e_3-block cannot be rows in A. Thus, the rows of A_7 have weight 0 or 3. Let x_3 denote the number of weight 3 rows in A_7. Since the rows of A_7 have weight 0 or 3, the number of ones in A_7 is $3x_3$. However, we already know A_7 contains 56 ones and 3 does not divide 56, which implies A_7 cannot contain $3x_3$ ones. Therefore, C cannot contain the incidence matrix A of a $(22, 33, 12, 8, 4)$-BIBD.

Thus, any $(33, 16)$ doubly-even self-orthogonal code that contains an e_3-block cannot contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. This eliminates the 76 e_3-codes from our list L_C of inequivalent $(33, 16)$ doubly-even self-orthogonal codes, leaving us with the $594 - 76 = 518$ e_{2i}-codes, d_i-codes, and distance 8 codes to check.

$\vec{g}_1$	1	1	1	0	1	0	0	0*
$\vec{g}_2$	1	1	0	1	0	1	0	0*
$\vec{g}_3$	1	0	1	1	0	0	1	0*
$\vec{c}_1$	1	1	1	1	1	1	1	$\vec{v}_1$
$\vec{c}_2$	1	1	1	0	1	0	0	$\vec{v}_2$
$\vec{c}_3$	1	1	0	1	0	1	0	$\vec{v}_3$
$\vec{c}_4$	1	0	1	1	0	0	1	$\vec{v}_4$
$\vec{c}_5$	1	1	0	0	0	0	1	$\vec{v}_5$
$\vec{c}_6$	1	0	1	0	0	1	0	$\vec{v}_6$
$\vec{c}_7$	1	0	0	1	1	0	0	$\vec{v}_7$
$\vec{c}_8$	1	0	0	0	1	1	1	$\vec{v}_8$
$\vec{c}_9$	0	1	1	1	0	0	0	$\vec{v}_9$
$\vec{c}_{10}$	0	1	1	0	0	1	1	$\vec{v}_{10}$
$\vec{c}_{11}$	0	1	0	1	1	0	1	$\vec{v}_{11}$
$\vec{c}_{12}$	0	0	1	1	1	1	0	$\vec{v}_{12}$
$\vec{c}_{13}$	0	1	0	0	1	1	0	$\vec{v}_{13}$
$\vec{c}_{14}$	0	0	1	0	1	0	1	$\vec{v}_{14}$
$\vec{c}_{15}$	0	0	0	1	0	1	1	$\vec{v}_{15}$
$\vec{c}_{16}$	0	0	0	0	0	0	0	$\vec{v}_{16}$

Figure 3.6: The codewords of a code that contains an e_3-block. Above the horizontal line are generators for the e_3-block. Below the horizontal line and to the left of the vertical line are all the different possibilities for the components of a codeword in C that are supported by the e_3-block.

3.9 Eliminating the e_{2i}-Codes

In this section, we will prove that any $(33, k, 4)$ doubly-even self-orthogonal code that contains an e_{2i}-block, where $i \geq 2$, cannot contain an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD.

Let C be a $(33, k, 4)$ doubly-even self-orthogonal code that contains an e_{2i}-block, where $i \geq 2$. Without loss of generality, assume the e_{2i}-block occurs in the first $4i$ coordinates of C. In Figure 3.7, we give generators for our e_{2i}-block.

Let us first consider the different possibilities for the first $4i$ components of a codeword in C (i.e. the components that are supported by the e_{2i}-block). Let $C_{e_{2i}}$ denote the $(4i, 2i)$ self-dual code that is generated by the e_{2i}-block with the trailing coordinates of zeros removed. Since $C_{e_{2i}}$ is self-dual, we have $C_{e_{2i}}^{\perp} = C_{e_{2i}}$. Since any codeword $\vec{c}$ in C must be orthogonal to the e_{2i}-block, the first $4i$ components of $\vec{c}$ must be an element of $C_{e_{2i}}^{\perp}$, and thus must be an element of $C_{e_{2i}}$. Therefore, the only possibilities for the first $4i$ components of any codeword in C are all the linear combinations of the first $4i$ components of our generators for the e_{2i}-block.

Suppose C contains the incidence matrix A of a $(22, 33, 12, 8, 4)$-BIBD. Let us now find all the different possibilities for the first $4i$ components of any codeword $\vec{c}$ that is a row in A, keeping in mind that the first $4i$ components of $\vec{c}$ must be a linear combination

$$
\begin{array}{c}
 \\
\vec{g}_1 \\
\vec{g}_2 \\
\vdots \\
\vec{g}_{2i-1} \\
\vec{g}_{2i}
\end{array}
\overbrace{\begin{array}{cccccccccc}
 & & 1 & 1 & 0 & 0 & \cdots & 0 & 0 & 0^* \\
 & & 1 & 1 & 0 & 0 & \cdots & 0 & 0 & 0^* \\
 & & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\
 & & 1 & 1 & 0 & 0 & \cdots & 1 & 1 & 0^* \\
 & & 1 & 0 & 1 & 0 & \cdots & 1 & 0 & 0^*
\end{array}}^{4i-2}
\left.\begin{array}{c} \\ \\ \\ \\ \end{array}\right\} 2i-1
$$

Figure 3.7: Generators for an e_{2i}-block, $i \geq 2$.

of the first $4i$ components of our generators for the e_{2i}-block. By Corollary 3.13, none of the rows in A can intersect any of the weight 4 words in the e_{2i}-block in 4 positions. Therefore, the first $4i$ components of any codeword $\vec{c}$ that is a row in A cannot be a linear combination of the first $2i-1$ generators in Figure 3.7 since:

- if $\vec{c} = \vec{g}_j$, where $1 \leq j \leq 2i-1$, then $\vec{g}_j$ is a weight 4 word in the e_{2i}-block that intersects $\vec{c}$ in 4 positions, and
- if $\vec{c}$ is the sum of two or more generators in the first $2i-1$ rows, say $\vec{c} = \vec{g}_{j_1} \oplus \vec{g}_{j_2} \oplus \cdots$, then $\vec{g}_{j_1} \oplus \vec{g}_{j_2}$ is a weight 4 word in the e_{2i}-block that intersects $\vec{c}$ in 4 positions.

Therefore, if $\vec{c}$ is a codeword that is a row in A then the first $4i$ components of $\vec{c}$ are either all zero, or, a linear combination of the first $2i-1$ generators plus the last generator in Figure 3.7. In the later case, we have a codeword $\vec{c}$ whose first $4i$ components can be partitioned into consecutive pairs with each pair having a value of either 1 0 or 0 1, which implies $\vec{c}$ has weight $2i$ in its first $4i$ components. Thus, the first $4i$ columns of any row in A either has weight 0 or weight $2i$.

Let A_{4i} denote the $22 \times 4i$ submatrix in the first $4i$ columns of A. Let x_0 denote the number of rows in A_{4i} that have weight 0 and let x_1 denote the number of rows in A_{4i} that have weight $2i$. Then, since A_{4i} only has rows with weight 0 or $2i$, we have $x_0 + x_1 = 22$. Furthermore, since the columns of A_{4i} all have weight 8, we have $2ix_1 = 8 \times 4i = 32i$, which implies $x_1 = 16$. Therefore, $x_0 = 22 - x_1 = 6$. Thus, A_{4i} contains 6 rows with weight 0, which implies A contains a $6 \times 4i$ matrix of zeros. Since $i \geq 2$, and thus $4i \geq 8$, this implies that for any $i \geq 2$, the matrix A_{4i}, and thus the matrix A, contains a 6×8 matrix of zeros.

Let Y be a 6×8 matrix of zeros. If we run Algorithm 3.11 with the matrix Y as input then the output returned is $S = \emptyset$. This implies that any A that contains the matrix Y cannot be the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. Therefore, C cannot contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD.

Thus, any $(33, 16)$ doubly-even self-orthogonal code that contains an e_{2i}-block, $i \geq 2$, cannot contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. This eliminates the 14 e_{2i}-codes, $i \geq 2$, from our list L_C of inequivalent $(33, 16)$ doubly-even self-orthogonal codes, leaving us with the $518 - 14 = 504$ d_i-codes and distance 8 codes to check.

3.10 Eliminating the d_i-Codes, $i \geq 5$

In this section, we will prove that any $(33, k, 4)$ doubly-even self-orthogonal code that contains a d_i-block, where $i \geq 5$, cannot contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD.

Let C be a $(33, k, 4)$ doubly-even self-orthogonal code that contains a d_i-block. Without loss of generality, assume the d_i-block occurs in the first $2i + 2$ coordinates of C. In Figure 3.8, we give generators for the d_i-block. Unless specified otherwise, we will assume $i \geq 1$ in our proof. This reason for this is that it will show us where things go wrong for the case of $i \leq 4$.

Let us first consider all the different possibilities for the first $2i + 2$ components of a codeword $\vec{c}$ in C (i.e. all the different possibilities for the components of $\vec{c}$ that are supported by the d_i-block). Suppose $\vec{c}$ has a value of either 0 0 or 1 1 in the pair

$$
\begin{array}{c}
\overbrace{}^{2i} \\
\begin{array}{cccccccccccc}
\vec{g}_1 & 1 & 1 & 1 & 1 & 0 & 0 & \cdots & 0 & 0 & 0^* \\
\vec{g}_2 & 1 & 1 & 0 & 0 & 1 & 1 & \cdots & 0 & 0 & 0^* \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\
\vec{g}_i & 1 & 1 & 0 & 0 & 0 & 0 & \cdots & 1 & 1 & 0^*
\end{array} \Big\} i \\
\begin{array}{cccccc}
p_0 & & p_1 & p_2 & \cdots & p_i
\end{array}
\end{array}
$$

Figure 3.8: Generators for the d_i-blocks.

of coordinates denoted by p_0 in Figure 3.8 (i.e. the first two coordinates). Then, for $j = 1, 2, \ldots, i$, since $\vec{c}$ must be orthogonal to the generator $\vec{g}_j$, the codeword $\vec{c}$ either has a value of 0 0 or 1 1 in the pair of coordinates p_j. Similarly, if $\vec{c}$ has a value of either 0 1 or 1 0 in the pair of coordinates p_0 then for $j = 1, 2, \ldots, i$, the codeword $\vec{c}$ either has a value of 0 1 or 1 0 in the pair of coordinates p_j.

We will refer to the codewords $\vec{c}$ that have a value of either 0 0 or 1 1 in the pair of coordinates p_0 as Type 0 codewords. We will refer to the codewords $\vec{c}$ that have a value of either 0 1 or 1 0 in the pair of coordinates p_0 as Type 1 codewords.

Suppose C contains the incidence matrix A of a $(22, 33, 12, 8, 4)$-BIBD. Let us now find all the different possibilities for the first $2i+2$ components of a codeword that is also a row in A. Now, by Corollary 3.13, we know that none of the rows in A can intersect any of the weight 4 words in the d_i-block in 4 positions. We will use this fact to find all the different possibilities for the first $2i + 2$ columns of a row in A by determining, for each of the Type 0 and Type 1 codewords, whether or not there exists a weight 4 word in the d_i-block that intersects the codeword in 4 positions.

Now, for each pair of coordinate pairs (p_{j_1}, p_{j_2}) in Figure 3.8 there exists a weight 4 word in the d_i-block whose ones are located in the pair (p_{j_1}, p_{j_2}):

- for the pair (p_0, p_j), where $1 \leq j \leq i$, the generator $\vec{g}_j$ is a weight 4 word whose ones are located in the pair of coordinate pairs (p_0, p_j), and
- for the pair (p_{j_1}, p_{j_2}), where $1 \leq j_1 < j_2 \leq i$, the codeword $\vec{g}_{j_1} \oplus \vec{g}_{j_2}$ is a weight 4 word whose ones are located in the pair of coordinate pairs (p_{j_1}, p_{j_2}).

These are the only weight 4 words in the d_i-block.

Now, the Type 0 codewords have a value of either 0 0 or 1 1 in each pair p_j, which implies a Type 0 codeword will intersect a weight 4 word in the d_i-block if and only if it has weight ≥ 4 in its first $2i+2$ components (i.e. if and only if it contains at least 2 pairs

p_{j_1} and p_{j_2} that have a value of 1 1). Therefore, a Type 0 codeword may be a row in the incidence matrix A if and only if it has weight ≤ 2 in its first $2i+2$ components. Since the Type 1 codewords have a value of either 1 0 or 0 1 in each pair p_j, none of the Type 1 codewords intersect any of the weight 4 words in the d_i-block in 4 positions. Therefore, every Type 1 codeword may be a row in A. Thus, the only possibilities for the first $2i+2$ components of a codeword that is a row in A are those given in Figure 3.10.

Let us now determine the number of times that each of the rows in Figure 3.10 that begin with either 0 0 or 1 1 may occur in the first $2i+2$ columns of A. Towards this goal, let us first count the number of rows that have a value of either 0 1 or 1 0 in a pair p_j. Since every column in A has weight 8, we know that for each pair of columns p_j, the number of rows in A that have a value of 0 1 in p_j is equal to the number of rows in A that have a value of 1 0 in p_j. Therefore, if A contains x rows that have a value of 1 1 in the pair of columns p_0 then A contains $8-x$ rows that have a value of 0 1 in the pair p_0

	0	0	0	0	0	0	$\cdots$	0	0
r_0	1	1	0	0	0	0	$\cdots$	0	0
r_1	0	0	1	1	0	0	$\cdots$	0	0
r_2	0	0	0	0	1	1	$\cdots$	0	0
$\vdots$	$\vdots$		$\vdots$		$\vdots$		$\vdots$	$\vdots$	
r_i	0	0	0	0	0	0	$\cdots$	1	1
	x_0	$\overline{x}_0$	x_1	$\overline{x}_1$	x_2	$\overline{x}_2$	$\cdots$	x_i	$\overline{x}_i$
	p_0		p_1		p_2		$\cdots$	p_i	

Figure 3.9: The possibilities for the first $2i+2$ components of a codeword in C that is a row in the incidence matrix A. The last row of $2i+2$ components represents the first $2i+2$ components of a Type 1 codeword. Each x_j has a value of either 0 or 1. The notation $\overline{x}_j$ means $1-x_j$.

and $8-x$ rows that have a value of 1 0 in the pair p_0. Furthermore, since a row of A has a value of either 0 1 or 1 0 in pair p_0 if and only if it has a value of either 0 1 or 1 0 in every pair p_j, for each pair p_j, A must contain $8-x$ rows that have a value of 0 1 in the pair p_j and $8-x$ rows that a value of 1 0 in the pair p_j. Thus, for each pair of columns p_j, A must contain exactly x rows that have a value of 1 1 in the pair p_j.

Since the only rows in A that have a value of 1 1 in the pair of columns p_j are those that begin with the row r_j depicted in Figure 3.10, A must contain exactly x rows that begin with the row r_j. Thus, A contains $(i+1)x$ rows that begin with one of the $i+1$ rows r_j, where $0 \geq j \leq i$, and $2(8-x)$ rows that begin with either 0 1 or 1 0. The only remaining possibilities for a row in A are those that begin with $2i+2$ zeros. The number of such rows must be $22-((i+1)x+2(8-x)) = 6-x(i-1)$. Thus, the first $2i+2$ columns of A must contain the $x(i+1)+(6-x(i-1)) = 6+2x$ rows depicted in Figure 3.10.

Without loss of generality, we will assume the rows depicted in Figure 3.10 occur in the first $6+2x$ rows of the first $2i+2$ columns of A. Let $Y_{i,x}$ denote this $(6+2x)\times(2i+2)$ submatrix in the upper left corner of A. Since each of the rows in Figure 3.10 cannot occur a negative number of times, we must have $x \geq 0$ and $6-x(i-1) \geq 0$, which implies $x \leq 6/(i-1)$, if $i \geq 2$. (Note that for the cases of $i=1$ and $i=2$, Theorem 3.12 tells us

$$
\begin{array}{rcccccccc}
6-x(i-1) & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & \\
x & 1 & 1 & 0 & 0 & \cdots & 0 & 0 & \Big\} \\
x & 0 & 0 & 1 & 1 & \cdots & 0 & 0 & \Big\} \; i+1 \\
 & \vdots & & \vdots & & & \vdots & & \Big\} \\
x & 0 & 0 & 0 & 0 & \cdots & 1 & 1 & \Big\}
\end{array}
$$

Figure 3.10: If C contains a d_i-block in its first $2i+2$ components, then the first $2i+2$ columns of A must contain the above rows. To the left of each row is the number of times that row must occur in A.

we must have $x \leq 3$.) Therefore, if $i \geq 8$ then we must have $x = 0$ and if $i = 5$, $i = 6$, or $i = 7$ then $x = 0$ or $x = 1$. Thus, if $i \geq 8$ then $Y_{i,0}$ is a submatrix of A, and if $i = 5$, $i = 6$, or $i = 7$ then either $Y_{i,0}$ or $Y_{i,1}$ is a submatrix of A.

If $i \geq 5$ then the matrix $Y_{5,0}$ is a submatrix of $Y_{i,0}$, which implies that if A contains $Y_{i,0}$ then A contains $Y_{5,0}$. If we run Algorithm 3.11 with $Y_{5,0}$ as input then the output returned is $S = \emptyset$. This implies there are no solutions to the column weights of the matrix adjacent to $Y_{5,0}$, and thus A cannot contain $Y_{5,0}$. Thus, if $i \geq 8$ then C cannot contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD, and if $i = 5$, $i = 6$, or $i = 7$ then A must contain the matrix $Y_{i,1}$.

If $i = 5$, $i = 6$, or $i = 7$ then the matrix $Y_{5,1}$ is a submatrix of $Y_{i,1}$, which implies A contains $Y_{5,1}$. If we run Algorithm 3.11 with $Y_{5,1}$ as input then the output returned is $S = \emptyset$. This implies A cannot contain $Y_{5,1}$. Thus, if $i = 5$, $i = 6$, or $i = 7$ then C cannot contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD.

Thus, any $(33, 16)$ doubly-even self-orthogonal code that contains a d_i-block, where $i \geq 5$, cannot contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. This eliminates the 26 d_i-codes, where $i \geq 5$, from our list L_C of inequivalent $(33, 16)$ doubly-even self-orthogonal codes, leaving us with the $504 - 26 = 478$ d_i-codes, $i \leq 4$, and distance 8 codes to check.

For the case of $i \leq 4$, we find there exist matrices $Y_{i,x}$ in which Algorithm 3.11 does return one or more solutions. Therefore, the method we just used to eliminate the d_i-codes, where $i \geq 5$, cannot be used to eliminate the d_i-codes when $i \leq 4$.

3.11 The Remaining Codes

We are now left with a list L of 478 inequivalent $(33, 16)$ doubly-even self-orthogonal codes with the property that if a $(22, 33, 12, 8, 4)$-BIBD exists then there must exist a code in L that contains the incidence matrix of such a design. Each code in the list L

is either a d_4-code, a d_3-code, a d_2-code, a d_1-code, or a distance 8 code. Ideally, we would either like to find the incidence matrix of such a design in one of these codes, or find a theoretical proof that each of these classes of codes cannot contain an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD. However, thus far we have not been able to do either. Therefore, we must use an algorithmic approach to search the weight 12 words of each code in L for the incidence matrix of such a design. This is the topic of our next chapter.

3.12 Concluding Remarks

We will now conclude this chapter with a review of the main results we have presented.

First, we developed a simple algorithm for determining when certain matrices cannot be submatrices in the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. The algorithm is a generalization of the methods used by Hamada and Kobayashi to find the block intersection patterns of a $(22, 33, 12, 8, 4)$-BIBD. We used our algorithm to aid us in proving the two main results presented in this chapter.

The first result was our alternate proof that there are only three possibilities, up to row and column rearrangement, for the columns supported by a weight 4 word in the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. Our proof does not depend on the results of Greig's work (i.e. Theorem 1.11), whose results required much computational time.

The second result was that any $(33, k, 4)$ doubly-even self-orthogonal code that contains either an e_3-block, an e_{2i}-block, where $i \geq 2$, or a d_i-block, where $i \geq 5$, cannot contain an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD. This result allows us to eliminate the 116 e_3-codes, e_{2i}-codes, where $i \geq 2$, and d_i-codes, where $i \geq 5$, from our complete list L_C of 594 inequivalent $(33, 16)$ doubly-even self-orthogonal codes. In Appendix B, we give tables that give the characteristics of each of the 116 codes we have theoretically eliminated. The information the tables provide for each code C includes the class of codes

to which C belongs (i.e. is it an e_3-code, an e_{2i}-code, $i \geq 2$, or a d_i-code, $i \geq 5$), the number of weight 4 words in C, and the size of $\mathcal{AUT}(C)$.

We are now left with a list L of 478 codes with the property that a $(22, 33, 12, 8, 4)$-BIBD exists if and only if L contains a code that contains the incidence matrix of such a design.

Chapter 4

BIBD Search: Searching The Remaining Codes

In this chapter, we will describe our algorithm for searching the weight 12 words of a $(33, 16)$ doubly-even self-orthogonal code for the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD.

In Section 4.1, we will give some of the definitions and notation we will need to describe our algorithm. In Section 4.2, we will give a general outline of our algorithm. Our algorithm consists of five basic stages. In Sections 4.3–4.7, we will give a detailed description of each of these five stages. In Section 4.8, we will combine the five basic stages of our algorithm to give us our complete algorithm. In Section 4.9, we will demonstrate the performance of our algorithm with an example.

Thus far, we have been able to search 155 of the 478 codes we have not eliminated theoretically. None of the codes we haved searched contained the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. In Section 4.10, we will discuss which codes we have been able to search. Included in the codes we have searched are all the d_4 and d_3-codes (as defined in Section 3.4). In Section 4.11, we will discuss several ideas that may help in searching

the 323 codes we have not yet been able to search. Finally, in Section 4.12, we will give a brief review of the main results presented in the chapter.

4.1 Definitions and Notation

In this section, we will define some of the concepts and notation we will use throughout this chapter.

4.1.1 Word Blocks

In Chapter 3, we used the weight 4 blocks to prove that certain classes of codes cannot contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. We will also make use of such structures in our algorithm for searching for an incidence matrix in a code. However, instead of limiting ourselves to just the weight 4 blocks, we will use a more general structure that we shall call a *word block*.

Definition 4.1. A *word block* W is any set of codewords that may occur in the orthogonal complement of a $(33, 16)$ doubly-even self-orthogonal code.

We will limit ourselves to word blocks W that consist only of weight 4 and/or weight 5 words. We consider any linear combination $\vec{c}$ of the words in W to be an element of W if and only if $\vec{c}$ has weight 4 or 5. The reason for this convention is that it allows us to classify the weight 4 blocks as word blocks. We typically represent our word blocks W with a set of linearly independent codewords in W. We refer to such a set as *generators* for W.

Let W and W' be any two word blocks. We consider W and W' to be *equivalent* if there exists a permutation π of the coordinates of W such that $\pi W = W'$. If no such permutation exists, we consider W and W' to be *inequivalent*. Generators for the all the inequivalent word blocks we will be working with in this chapter are given in Figure 4.1.

1 1 1 1 0 0 0 0 0 0 0*	1 1 1 1 0 0 0 0 0 0*
1 1 0 0 1 1 0 0 0 0 0*	1 1 0 0 1 1 0 0 0 0*
1 1 0 0 0 0 1 1 0 0 0*	1 1 0 0 0 0 1 1 0 0*
1 1 0 0 0 0 0 0 1 1 0*	1 0 1 0 1 0 1 0 1 0*
the d_4-block	the d_3^*-block

1 1 1 1 0 0 0 0 0*	1 1 1 1 0 0 0 0 0*
1 1 0 0 1 1 0 0 0*	1 1 0 0 1 1 0 0 0*
1 1 0 0 0 0 1 1 0*	1 0 1 0 1 0 1 1 0*
the d_3-block	the d_2^*-block

1 1 1 1 0 0 0*	1 1 1 1 0 0 0 0*
1 1 0 0 1 1 0*	1 1 0 0 1 1 1 0*
the d_2-block	the d_1^*-block

1 1 1 1 0*	1 1 1 1 1 0*
the d_1-block	the f_1-block

Figure 4.1: Generators for our word blocks. The notation 0* is used to represent a string of zeros that extends each codeword to have 33 components.

The d_4, d_3, d_2, and d_1-blocks are just the weight 4 blocks that we have not been able to prove cannot occur in the point code of a $(22, 33, 12, 8, 4)$-BIBD. The d_3^*, d_2^*, and d_1^*-blocks are the word blocks consisting of the d_3, d_2, and d_1-blocks, respectively, with a particular weight 5 word attached. The f_1-block is a word block consisting of just a weight 5 word.

We know, by Appendix A, that the orthogonal complement of any $(33, 16)$ doubly-even self-orthogonal code must contain at least six weight 5 words. So clearly, the orthogonal complement of each of our remaining codes must contain a word block that is equivalent to one of the word blocks in Figure 4.1. The reason Figure 4.1 contains word blocks other than the f_1-block is that, generally speaking, the "bigger" the word block the easier it is to determine that a code, whose orthogonal complement contains the word block, cannot contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. So, Figure 4.1 contains a list of the *convenient* word blocks we will be working with in this chapter.

4.1.2 Left Word Blocks

Let W be a word block. Let $S(W)$ denote the support of W. That is, $S(W)$ is the set of all coordinates j for which there exists at least one codeword in W that has a value of 1 in coordinate j. Throughout this chapter we will use the notation $n(W)$ to denote $|S(W)|$.

We call W a *left word block* if $S(W) = \{1, 2, \ldots, n(W)\}$. That is, W is a left word block if the non-zero coordinates in W all occur to the left of the zero coordinates in W. The word blocks given in Figure 4.1 are all examples of left word blocks.

Let W be a left word block. Let π be a permutation of the coordinates of W that only moves the integers $1, 2, \ldots, n(W)$. We consider π to be an *automorphism* of W if $\pi W = W$. Note that since W is a set, we do not require that π maps every codeword in W to itself.

Example 4.2. Suppose W is a d_2-block:

$$W = \begin{array}{c} \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & \cdots \end{array} \\ \left\{ \begin{array}{ccccccc} 1 & 1 & 1 & 1 & 0 & 0 & 0^* \\ 1 & 1 & 0 & 0 & 1 & 1 & 0^* \\ 0 & 0 & 1 & 1 & 1 & 1 & 0^* \end{array} \right\} \end{array}.$$

Consider the permutations $\pi_1 = (1\ 3)$ and $\pi_2 = (1\ 3)(2\ 4)$. The sets $\pi_1 W$ and $\pi_2 W$ are:

$$\pi_1 W = \left\{ \begin{array}{ccccccc} 1 & 1 & 1 & 1 & 0 & 0 & 0^* \\ 0 & 1 & 1 & 0 & 1 & 1 & 0^* \\ 1 & 0 & 0 & 1 & 1 & 1 & 0^* \end{array} \right\} \quad \pi_2 W = \left\{ \begin{array}{ccccccc} 1 & 1 & 1 & 1 & 0 & 0 & 0^* \\ 0 & 0 & 1 & 1 & 1 & 1 & 0^* \\ 1 & 1 & 0 & 0 & 1 & 1 & 0^* \end{array} \right\}$$

Since $\pi_1 W \neq W$, π_1 is not an automorphism of W. Since $\pi_2 W = W$, π_2 is an automorphism of W.

We will use the notation $\mathcal{AUT}(W)$ to denote the set of all automorphisms of W. Since for any permutations $\pi_1, \pi_2 \in \mathcal{AUT}(W)$, we have $\pi_1\pi_2 W = \pi_1(\pi_2 W) = \pi_1 W = W$, we know $\mathcal{AUT}(W)$ is a group. Therefore, we will refer to $\mathcal{AUT}(W)$ as the *automorphism group of* W.

For a given left word block W, we will use the notation $\mathcal{C}(W)$ to denote the set of all $(33, 16)$ doubly-even self-orthogonal codes C in which $C^{\perp}$ contains W. We will use the notation $A(W)$ to denote any incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD that may be contained in a code in $\mathcal{C}(W)$. In other words, $A(W)$ will denote any incidence matrix whose rows are orthogonal to every word in the left word block W.

4.1.3 Left Patterns

Since the support of W occurs in the first $n(W)$ coordinates of W, there is a limited number of possibilities for the first $n(W)$ columns of any incidence matrix $A(W)$ that is contained in a code in $\mathcal{C}(W)$. We refer to these possibilities as *left patterns* of the left word block W, which we formally define next.

Definition 4.3. Let W be a left word block. A *left pattern* of W is a $22 \times n(W)$ matrix P that may occur in the first $n(W)$ columns of an incidence matrix $A(W)$. More precisely, P is any matrix we have not been able to theoretically prove cannot occur in the first $n(W)$ columns of an incidence matrix $A(W)$ contained in a code in $\mathcal{C}(W)$.

Example 4.4. Suppose W is a d_3-block. Then:

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

are generators for W (with the trailing zero coordinates eliminated). Consider the $22 \times n(W)$ matrix P in Figure 4.2. We have not been able to theoretically prove there does not exist an incidence matrix $A(W)$ that begins with P. That is, P has all the properties that we know a left pattern of W must posses and that we can check without an exhaustive search (such as our BIBD search algorithm). For example, the rows of P intersect the first $n(W)$ components of each word in W in an even number of positions and the columns of P all have weight 8. Therefore, P is a left pattern of W.

For a given code $C \in \mathcal{C}(W)$, our basic strategy for determining whether or not C contains an incidence matrix $A(W)$ is as follows: First, we produce a list of left patterns $L(W, C)$ with the property that C contains an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD if and only if C contains an incidence matrix that begins with one of the left patterns in $L(W, C)$. For each P in $L(W, C)$, we then search the weight 12 words of C for an incidence matrix $A(W)$ that begins with P. If for each P in $L(W, C)$, we do not find an incidence matrix, we then know C cannot contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD.

In order to produce the list $L(W, C)$, we first find a complete list $L(W)$ of *inequivalent* left patterns for the left word block W. Informally, we consider two left patterns P_1 and P_2 to be *equivalent* if we can permute the rows and columns of P_1 to obtain P_2 without

$$P = \begin{bmatrix}
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

Figure 4.2: A left pattern of a d_3-block.

altering the left word block W. If no such permutations exist, we consider P_1 and P_2 to be *inequivalent.*

Before we can give a formal definition of equivalent and inequivalent left patterns, we must first introduce some permutation notation that we will use throughout this chapter to deal with the fact that any left word block W has 33 coordinates while any left pattern of W has only $n(W)$ columns. Given any permutation π of the integers $1, 2, \ldots, n(W)$, we will use the notation π^{33} to denote the permutation of the integers $1, 2, \ldots, 33$ that moves the integers $1, 2, \ldots, n(W)$ in the same fashion as π but does not move the integers $n(W)+1, n(W)+2, \ldots, 33$. Conversely, given a permutation π of the integers $1, 2, \ldots, 33$ that only moves the integers $1, 2, \ldots, n(W)$, we will use the notation $\pi^{n(W)}$ to denote the permutation of the integers $1, 2, \ldots, n(W)$ that moves the integers $1, 2, \ldots, n(W)$ in the same fashion as π.

Example 4.5. Suppose $n(W) = 10$. Consider the permutations:

$$\pi_1 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & \cdots & 33 \\ 5 & 10 & 1 & 3 & 4 & 6 & 9 & 7 & 8 & 2 & 11 & 12 & \cdots & 33 \end{pmatrix}$$

$$\pi_2 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 5 & 10 & 1 & 3 & 4 & 6 & 9 & 7 & 8 & 2 \end{pmatrix}$$

The permutation π_1 is a permutation of the integers $1, 2, \ldots, 33$, while the permutation π_2 is a permutation of the integers $1, 2, \ldots, n(W)$. The permutation π_1 moves the integers $1, 2, \ldots, n(W)$ in the same fashion as the permutation π_2. Furthermore, π_1 does not move the integers $n(W) + 1, n(W) + 2, \ldots, 33$. If we are given the permutation π_1, and want to use the permutation π_2, then we will use the notation $\pi_1^{n(W)}$ to denote π_2. Similarly, if we are given the permutation π_2, we will use the notation π_2^{33} to denote the permutation π_1.

We are now ready to give a formal description of the concepts of equivalent and inequivalent left patterns of W.

Definition 4.6. Let W be a left word block. Let P_1 and P_2 be any two left patterns of W. Then P_1 and P_2 are said to be *equivalent* if there exists a permutation π_r of the rows of P_1 and a permutation π_c of the columns of P_1, where $\pi_c^{33} \in \mathcal{AUT}(W)$, such that $\pi_r \pi_c P_1 = P_2$. If no such permutations exist, we say P_1 and P_2 are *inequivalent.*

Given a complete list $L(W)$ of inequivalent left patterns of W, we can find a list of all left patterns that may occur in the first $n(W)$ columns of an incidence matrix $A(W)$, up to row rearrangement, by simply applying each permutation π_c, where $\pi_c^{33} \in \mathcal{AUT}(W)$, to the columns of each left pattern in $L(W)$. However, for a given code $C \in \mathcal{C}(W)$, it may not be necessary to include every one of these left patterns in our list $L(W, C)$. For example, suppose the automorphism group of C contains a permutation $\pi_l \pi_r$, where π_l only moves the integers $1, 2, \ldots, n(W)$ and π_r only moves the integers $n(W)+1, n(W)+2, \ldots, 33$, such that π_l is also an element of $\mathcal{AUT}(W)$. Then, since $\pi_l \pi_r$ is an automorphism of C, if C contains an incidence matrix $A(W)$ then C must also contain an incidence matrix $\pi_l \pi_r A(W)$. Furthermore, if P is the left pattern in the first $n(W)$ columns of $A(W)$ then the left pattern in the first $n(W)$ columns of $\pi_l \pi_r A(W)$ is $\pi_l^{n(W)} P$. Thus, for each left pattern P in $L(W)$, the code C contains an incidence matrix that begins with P if and only if C contains an incidence matrix that begins with the left pattern $\pi_l^{n(W)} P$. Therefore, if we search for an incidence matrix in C that begins with P then we do not have to search for an incidence matrix in C that begins with $\pi_l^{n(W)} P$. This leads us to the concept of a *partial left automorphism* of W and C.

Definition 4.7. Let W be a left word block, let C be a code in $\mathcal{C}(W)$, and let π_l be a permutation in $\mathcal{AUT}(W)$. Then π_l is a *partial left automorphism* of W and C if there exists a permutation π_r of the coordinates of C that only moves the integers $n(W)+1, n(W)+2, \ldots, 33$, such that $\pi_l \pi_r$ is an automorphism of C.

Example 4.8. Suppose $n(W) = 10$. Suppose $\mathcal{AUT}(W)$ contains the permutation $\pi_l = (2\ 7\ 5)$ and suppose $\mathcal{AUT}(C)$ contains the permutation $\pi = (2\ 7\ 5)(12\ 15)$. Then the permutation $\pi_r = (12\ 15)$ is a permutation that only moves the integers $n(W)+1, n(W)+$

$2, \ldots, 33$, such that $\pi_l \pi_r = \pi$ is an automorphism of C. Therefore, π_l is a partial left automorphism of W and C.

We will use the notation $\mathcal{AUT}_{PART\_LEFT}(W, C)$ to denote the set of all partial left automorphisms of W and C. The set $\mathcal{AUT}_{PART\_LEFT}(W, C)$ is a subgroup of $\mathcal{AUT}(W)$. We will refer to $\mathcal{AUT}_{PART\_LEFT}(W, C)$ as the *partial left automorphism group* of W and C.

As we shall see later in this chapter, we will use $\mathcal{AUT}(W)$, $\mathcal{AUT}_{PART\_LEFT}(W, C)$, and our complete list $L(W)$ of inequivalent left patterns of W, to produce our list $L(W, C)$ of left patterns with the property that C contains an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD if and only if C contains an incidence matrix that begins with a left pattern in $L(W, C)$.

4.1.4 Left and Right Words

Next we will discuss the concepts of the left and right words of a code in $\mathcal{C}(W)$.

Definition 4.9. Let W be a left word block. A *left word* of W is any binary string of length $n(W)$ that may occur in the first $n(W)$ components of a codeword in a code in $\mathcal{C}(W)$. In other words, a left word is a binary string of length $n(W)$ that intersects each of the words in W in an even number of positions.

We will use the notation $\mathcal{LEFT}(W)$ to denote the set of all left words that do not intersect any of the weight 4 words in W in 4 positions. In other words, $\mathcal{LEFT}(W)$ is the set of all binary strings that may occur in a row of a left pattern P of W. The reason we do not include any of the left words that intersect a weight 4 word in W in 4 positions is that, by Corollary 3.13, none of the rows of P can intersect the weight 4 words of W in 4 positions.

Definition 4.10. Let W be a left word block. Let C be a code in $\mathcal{C}(W)$. A *right word*

of W and C is any binary string of length $33 - n(W)$ that occurs in the last $33 - n(W)$ components of a codeword in C.

For a given $C \in \mathcal{C}(W)$ and $l \in \mathcal{LEFT}(W)$, we will use the notation $\mathcal{RIGHT}(W, C, l)$ to denote the set of all right words r in which lr is a weight 12 word in C that does not intersect any of the weight 4 words in C in 4 positions (where the notation lr is used to denote the concatenation of the strings l and r). In other words, $\mathcal{RIGHT}(W, C, l)$ is the set of all right words r such that lr forms a weight 12 word that may occur in a row of an incidence matrix $A(W)$ of a $(22, 33, 12, 8, 4)$-BIBD that is contained in C. We will refer to $\mathcal{RIGHT}(W, C, l)$ as the *right set* in C for the left word l.

4.1.5 Left and Right Automorphisms

The final concepts we will discuss in this section are the left and right automorphisms of a code in $\mathcal{C}(W)$.

Definition 4.11. Let W be a left word block and let C be a code in $\mathcal{C}(W)$. A *left automorphism* of W and C is an automorphism of C that only moves the first $n(W)$ coordinates of C. A *right automorphism* of W and C is an automorphism of C that only moves the last $33 - n(W)$ coordinates of C.

Example 4.12. Suppose $n(W) = 10$. Suppose $\mathcal{AUT}(C)$ contains the following three permutations:

$$\begin{aligned} \pi_1 &= (4\ 2\ 7)(5\ 6) \\ \pi_2 &= (15\ 18\ 19) \\ \pi_3 &= (8\ 9\ 16) \end{aligned}$$

The permutation π_1 does not move any of the integers $n(W)+1, n(W)+2, \ldots, 33$, and therefore is a left automorphism of W and C. The permutation π_2 does not move any of the integers $1, 2, \ldots, n(W)$, and therefore, is a right automorphism of W and C. The permutation π_3 is neither a left automorphism nor a right automorphism of W and C.

We will use the notation $\mathcal{AUT}_{LEFT}(W, C)$ to denote the set of all left automorphisms of W and C. We will use the notation $\mathcal{AUT}_{RIGHT}(W, C)$ to denote the set of all right automorphisms of W and C. Both $\mathcal{AUT}_{LEFT}(W, C)$ and $\mathcal{AUT}_{RIGHT}(W, C)$ are subgroups of $\mathcal{AUT}(C)$. We will refer to $\mathcal{AUT}_{LEFT}(W, C)$ as the *left automorphism group* of W and C and $\mathcal{AUT}_{RIGHT}(W, C)$ as the *right automorphism group* of W and C. Note that $\mathcal{AUT}_{LEFT}(W, C)$ is not necessarily equal to $\mathcal{AUT}_{PART\_LEFT}(W, C)$. For example, if $\pi_l \in \mathcal{AUT}_{PART\_LEFT}(W, C)$ and $S(\pi_l) = \{\pi_r \mid \pi_r$ is a permutation that only moves the integers $n(W)+1, n(W)+2, \ldots, 33$ in which $\pi_l\pi_r \in \mathcal{AUT}(C)\}$, then it is not necessarily true that the identity permutation is in $S(\pi_l)$, and thus it is not necessarily true that π_l is a left automorphism of W and C.

This completes our look at some of the basic concepts and notation we will use throughout this chapter in describing our algorithm for searching for an incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD in a given code.

4.2 General Outline of Algorithm

Let L_C denote our list of 478 inequivalent $(33, 16)$ doubly-even self-orthogonal codes that we have not been able to eliminate theoretically (i.e. the d_i-codes, where $i \leq 4$, and the distance 8 codes). In this section, we will give a general outline of each of the basic stages in our algorithm for determining whether or not a given code in L_C contains the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD.

We begin by producing, for each left word block W in Figure 4.1, a complete list $L(W)$ of inequivalent left patterns of W. What we mean by a "complete list" is a list $L(W)$ with the property that if $A(W)$ is any incidence matrix contained in a code in $\mathcal{C}(W)$ then $L(W)$ contains a left pattern that is equivalent to the left pattern in the first $n(W)$ columns of $A(W)$. How we produce such a list $L(W)$ is the subject of Section 4.3.

Let C be a code in our list L_C. Then $C^{\perp}$ may contain many word blocks that are

equivalent to one or more of the left word blocks in Figure 4.1. However, our BIBD search algorithm will only work with one of the word blocks in $C^{\perp}$. Therefore, we will need to select the word block in $C^{\perp}$ that our algorithm will work with. Which word block we select is the subject of Section 4.4.

Once we have selected, for each C in L_C, the word block W' in $C^{\perp}$ that we will use in our BIBD search algorithm, we then permute the coordinates of C in such a way that W' is permuted to the left word block W to which W' is equivalent. This leaves us with a list L_C of codes C in which $C^{\perp}$ contains one of the left word blocks in Figure 4.1. We then partition the list of codes L_C into the sets $\mathcal{D}(d_4)$, $\mathcal{D}(d_3^*)$, $\mathcal{D}(d_3)$, $\mathcal{D}(d_2^*)$, $\mathcal{D}(d_2)$, $\mathcal{D}(d_1^*)$, $\mathcal{D}(d_1)$, and $\mathcal{D}(f_1)$, where $\mathcal{D}(W)$ is the set of codes in L_C in which W is the left word block to which our selected word block was permuted.

For a given code $C \in \mathcal{D}(W)$, we then determine whether or not C contains an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD. We accomplish this by first finding a list $L(W, C)$ of left patterns with the property that C contains the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD if and only if C contains an incidence matrix that begins with one of the left patterns in $L(W, C)$. In Section 4.5, we will show how we can produce such a list $L(W, C)$ using the automorphism group of W, the partial left automorphism group of W and C, and our complete list $L(W)$ of inequivalent left patterns of W.

Once we have produced the list $L(W, C)$, our algorithm then searches the weight 12 words of C for an incidence matrix $A(W)$ that begins with the left pattern P, for each left pattern P in $L(W, C)$. If for each P in $L(W, C)$ we find C does not contain an incidence matrix that begins with P, we then know C does not contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD.

For a given left pattern P in $L(W, C)$, we use a backtrack algorithm to search the weight 12 words of C for an incidence matrix that begins with P. We begin the search with a 22×33 matrix A whose first $n(W)$ columns contain P. We then use our backtrack algorithm to fill in, row by row, the last $33 - n(W)$ columns of A, using the fact that if l

is the left word in the first $n(W)$ columns of row i of A then the only choices for the right word in the last $33-n(W)$ columns of row i of A are the right words r in $\mathcal{RIGHT}(W, C, l)$ in which lr intersects each of the first $i-1$ rows of A in 4 positions. The algorithm runs until either an incidence matrix is found or we have exhausted all possibilities for the rows of A. In order to reduce the number of possibilities the algorithm needs to try, our backtrack algorithm also includes pruning of the search space. In Section 4.6, we will discuss the details of our backtrack search algorithm including the methods we use to prune the search space.

Since the last $33 - n(W)$ columns of A are filled row by row, the way in which the rows in the left pattern P are ordered can greatly affect the running time of our backtrack search. Therefore, before we begin our search, we sort the rows of P in an attempt to minimize the running time of our algorithm. In Section 4.7, we will discuss how we sort the rows of our left patterns. Finally, in Section 4.8, we will put all the pieces of our BIBD search algorithm together.

This completes our brief look at the main stages in our algorithm for determining whether or not a code in L_C contains the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. In the sections to follow, we will discuss in detail how we accomplish each of the stages of our algorithm.

4.3 The Left Pattern Algorithm

In this section, we will describe our algorithm for producing a complete list $L(W)$ of inequivalent left patterns for a given left word block W. We will refer to this algorithm as our *Left Pattern Algorithm.*

Let us begin with a brief review of our left pattern terminology. Let W be a left word block. Recall that a left pattern of W is any $22 \times n(W)$ matrix P that we have not been able to theoretically prove cannot occur in the first $n(W)$ columns of an incidence matrix

$A(W)$. The set $\mathcal{LEFT}(W)$ consists of all the left words of W that may be a row in a left pattern of W. We consider two left patterns P_1 and P_2 to be *equivalent* if there exists a permutation π_r of the rows of P_1 and a permutation π_c of the columns of P_1, where π_c^{33} is an automorphism of W, such that $\pi_r\pi_c P_1 = P_2$. If no such permutations exist, we consider P_1 and P_2 to be *inequivalent.*

In this section only, we will use the term *left pattern* to refer to *any* $22 \times n(W)$ matrix P whose rows are elements of $\mathcal{LEFT}(W)$. If we cannot theoretically determine whether or not there exists an incidence matrix $A(W)$ that begins with P, we will refer to P as a *valid left pattern*, otherwise, we will refer to P as an *invalid left pattern.* Of course, if P_1 and P_2 are equivalent left patterns then P_1 is a valid left pattern if and only if P_2 is a valid left pattern.

We will build up our Left Pattern Algorithm in three phases. In the first phase, we will describe the algorithm we use to enumerate a list of left patterns (both valid and invalid). In the second phase, we will discuss how we can alter the algorithm to enumerate a list of *inequivalent* left patterns. Finally, in the third phase, we will discuss how we can alter the algorithm to enumerate a list of inequivalent *valid* left patterns.

4.3.1 Enumerating the Left Patterns of W

We enumerate the left patterns simply by recursively filling in each left pattern, row by row, with left words from the set $\mathcal{LEFT}(W)$. A formal description of our algorithm is given in Algorithm 4.13. The algorithm is recursive and is begun with the call *EnumerateLeftPats*(P_0, L), where L is initially empty.

Algorithm 4.13. *EnumerateLeftPats*(P_{i-1}, L):

- Input: The $(i-1) \times n(W)$ matrix P_{i-1}. The rows of P_{i-1} are elements of the set $\mathcal{LEFT}(W)$.
- Output: The list L of all left patterns of W.

begin
  if $i-1=22$ **then**
    Insert P_{i-1} into L.
  else
    for each left word l in $\mathcal{LEFT}(W)$ **do**
      Append l to the end of P_{i-1}, producing the $i \times n(W)$ matrix P_i.
      Call *EnumerateLeftPats*(P_i, L).
    end for
end

Throughout this section, we will use the term *partial left pattern* of W to refer to the $i \times n(W)$ matrices P_i produced at the i^{th} level of the recursion in Algorithm 4.13. In other words, a partial left pattern is an $i \times n(W)$ matrix whose rows are elements of $\mathcal{LEFT}(W)$.

Let us now consider how we can alter Algorithm 4.13 to only enumerate a list of *inequivalent* left patterns of W. Basically, our method for producing only inequivalent left patterns is to define a canonical form for our left patterns. We will define our canonical form in such a way that it will allow us to determine when a partial left pattern P_i produced by Algorithm 4.13 *cannot* be completed to a left pattern that is in our canonical form. We will refer to the left patterns that are in our canonical form as *minimal increasing left patterns*, which we will define in the next subsection.

4.3.2 Minimal Increasing Left Patterns

Since we are enumerating our left patterns row by row, we will first define a unique ordering for the rows of the left patterns that are in our canonical form.

Definition 4.14. Let P be a left pattern of W. Then P is an *increasing left pattern* if its rows are in increasing order. Recall, from Section 1.1.6, that the rows of a binary

matrix are considered to be in increasing order if, when considered as binary integers, the integer value of row i of P is less than or equal to the integer value of row $i+1$ of P.

Since we can sort the rows of any left pattern into increasing order, any left pattern is equivalent to an increasing left pattern. However, two increasing left patterns are not necessarily inequivalent. The reason for this is that if we apply a permutation π_c, where $\pi_c^{33} \in \mathcal{AUT}(W)$, to the columns of an increasing left pattern P, and then apply a permutation π_r that sorts the rows of $\pi_c P$ into increasing order, then the resulting left pattern $\pi_r \pi_c P$ is an increasing left pattern that is equivalent to P, but not necessarily equal to P. Therefore, in order to distinguish a unique left pattern in the set of all increasing left patterns that are equivalent, we will define a comparison relation between two increasing left patterns:

Definition 4.15. Let P_1 and P_2 be any two distinct increasing left patterns of W. Suppose the first row in which P_1 and P_2 differ is row i. We say P_1 is *less than* P_2 if the integer value of row i of P_1 is less than the integer value of row i of P_2. Otherwise, we say P_1 is *greater than* P_2.

This leads us to our canonical form for our left patterns:

Definition 4.16. Let P be a left pattern of W. We say P is a *minimal increasing left pattern* if P is an increasing left pattern and if P is less than or equal to any increasing left pattern that is equivalent to P.

4.3.3 Enumerating the Minimal Increasing Left Patterns

We are now ready to describe our algorithm for enumerating a list of inequivalent left patterns of W. More specifically, we are ready to make the changes to Algorithm 4.13 that will result in the algorithm enumerating only the minimal increasing left patterns of W.

There are two basic changes that we will make to Algorithm 4.13. First, we will only adjoin to the end of the input partial left pattern P_{i-1}, the left words in $\mathcal{LEFT}(W)$ that are greater than or equal to the last row of P_{i-1}. This results in the algorithm enumerating the increasing left patterns only. The second change we will make is to determine, for each partial left pattern P_i produced by the algorithm, if it is *not* possible to complete P_i to a minimal increasing left pattern. We accomplish this by producing each of the partial left patterns $\pi_r\pi_c P_i$, where $\pi_c^{33} \in \mathcal{AUT}(W)$ and π_r is a permutation that sorts the rows of $\pi_c P_i$ in increasing order. If we find a $\pi_r\pi_c P_i$ that is less than P_i, we then know P_i cannot be completed to a minimal increasing left pattern.

In Algorithm 4.17, we give a formal description of our algorithm for determining whether or not a partial left pattern (whose rows are in increasing order) cannot be completed to a minimal increasing left pattern. The algorithm is implemented as a Boolean function. It returns a value of *false* if we can determine that the input partial left pattern cannot be completed to a minimal increasing left pattern. Of course, if $i = 22$ then the algorithm returns *true* if and only if the input left pattern is a minimal increasing left pattern.

Algorithm 4.17. boolean *MinIncPartialLeftPattern*(P_i):

- Input: the partial left pattern P_i. The rows of P_i must be in increasing order.
- Output: returns a value of *false* if P_i cannot be completed to a minimal increasing left pattern, otherwise a value of *true* is returned.

begin
Let *min_pat* = *true*.
for each $\pi \in \mathcal{AUT}(W)$ **do**
Let $\pi_c = \pi^{n(W)}$.
Let π_r be any permutation that sorts the rows of $\pi_c P_i$ in increasing order.
if $\pi_r\pi_c P_i$ is less than P_i **then**

Set *min_pat* = *false*.
Break from the for loop.
end if
end for
Return *min_pat*.
end

A formal description of our algorithm for enumerating a list of minimal increasing left patterns is given in Algorithm 4.18. The algorithm is recursive and is begun with the call *EnumerateMinIncLeftPatterns*(P_0, L), where L is initially empty.

Algorithm 4.18. *EnumerateMinIncLeftPatterns*(P_{i-1}, L):

- Input: the partial left pattern P_{i-1}. The rows of P_{i-1} are in increasing order. P_{i-1} also has the property that it is less than or equal to each of the partial left patterns $\pi_r \pi_c P_{i-1}$, where $\pi_c^{33} \in \mathcal{AUT}(W)$ and π_r is a permutation that sorts the rows of $\pi_c P_{i-1}$ in increasing order.
- Output: a complete list L of minimal increasing left patterns. In other words, a list L with the property that for any left pattern P of W, there exists one and only one left pattern in L that is equivalent to P.

begin
if $i - 1 = 22$ **then**
Insert P_{i-1} into L.
else
for each left word l in $\mathcal{LEFT}(W)$ **do**
if $i - 1 = 0$ **or** l is greater than or equal to row $i - 1$ of P_{i-1} **then**
Append l to the end of P_{i-1}, producing the partial left pattern P_i.

if *MinIncPartialLeftPattern*(P_i) **then**
Call *EnumerateMinIncLeftPatterns*(P_i, L).
end if
end for
end else
end

We now have our algorithm for enumerating a list L of minimal increasing left patterns of W. Of course, not all of the left patterns produced by the algorithm can occur in the first $n(W)$ columns of an incidence matrix $A(W)$ for a $(22, 33, 12, 8, 4)$-BIBD. How we eliminate such left patterns is the subject of our next subsection.

4.3.4 Eliminating the Invalid Left Patterns

We will now discuss how we determine when a left pattern produced by Algorithm 4.18 is an invalid left pattern. More specifically, we will discuss several ways in which we can determine when a partial left pattern P_i produced by the algorithm can only be completed to left patterns that we know cannot occur in the first $n(W)$ columns of an incidence matrix $A(W)$ for a $(22, 33, 12, 8, 4)$-BIBD. We will refer to such a partial left pattern as an *invalid partial left pattern* and the converse as a *valid partial left pattern.*

In determining whether or not a partial left pattern is an invalid left pattern, we will make use of the following properties that any (valid) left pattern P, that occurs in the first $n(W)$ columns of an incidence matrix $A(W)$, must posses:

- Since every pair of rows in $A(W)$ intersect in 4 positions, every pair of rows in P must intersect in at most 4 positions.
- Since the columns of A all have weight 8, the columns of P all have weight 8.

- By Theorem 1.9, each pair of columns in A, and thus, each pair of columns in P, must intersect in either 1, 2, 3, or 4 positions.
- By Theorem 3.12, for each weight 4 word $\vec{w}$ in the left word block W, there are exactly 3 different possibilities, up to row and column rearrangement, for the four columns in P that are supported by $\vec{w}$.
- By Theorem 3.14, for each weight 5 word $\vec{w}$ in W, there are exactly 7 different possibilities, up to row and column rearrangement, for the five columns in P that are supported by $\vec{w}$.
- Finally, if M is a $z \times n(W)$ matrix whose rows are taken from P, then there must exist a $z \times 33$ matrix N, whose first $n(W)$ columns are M, such that the rows of N have weight 12 and intersect in 4 positions.

We will now discuss how we can use each of these properties to determine whether or not a given P_i is an invalid partial left pattern. More specifically, for each property we will discuss how we can determine when a partial left pattern P_i cannot be completed to a left pattern P that has the given property. At the conclusion of each discussion, we will give a function name that we will later use in our formal description of our algorithm for determining whether or not a given P_i is invalid. Each function will be a Boolean function that returns a value of *false* if it can determine that P_i cannot be completed to a left pattern with the given property.

In our discussions, we will use the notation $\mathcal{R}(i)$ to denote the set of all left words in $\mathcal{LEFT}(W)$ that are greater than or equal to the last row in the partial left pattern P_i (i.e. $\mathcal{R}(i)$ is the set of all left words that the algorithm will use to complete P_i to an increasing left pattern). We will also assume that the input partial left pattern P_{i-1} is valid, and thus, we only need to consider the effect that the i^{th} row of each P_i produced has in determining whether or not P_i is a valid partial left pattern.

Let us first consider the number of positions in which row i of P_i intersects each of its

first $i-1$ rows. Since every pair of rows in a left pattern P intersect in at most 4 positions, row i of P_i must intersect each of its first $i-1$ rows in at most 4 positions. Therefore, if row i of P_i intersects one of the first $i-1$ rows in P_i in more than 4 positions then P_i is an invalid left pattern. We will use the Boolean function *ValidRowIntersections*(P_i) to determine whether or not row i of P_i intersects each of the first $i-1$ rows of P_i in at most 4 positions.

Let us now consider the weights of the columns of P_i. Since every column in a left pattern P has weight 8, we must be able to complete P_i to a left pattern whose columns all have weight 8. Therefore, the columns of P_i must have weight ≤ 8. Furthermore, if column c of P_i has weight w then in order to complete P_i to a left pattern with weight 8 in column c, we must have $22 - i \geq 8 - w$. We can also use the set of left words $\mathcal{R}(i)$ to check whether or not P_i can be completed to a left pattern whose columns all have weight 8. For example, if column c of P_i has weight < 8 then there must exist at least one left word in $\mathcal{R}(i)$ that has a value of 1 in column c. We will use the Boolean function *ValidColumnWeights*$(P_i, \mathcal{R}(i))$ to check whether or not P_i can be completed to a left pattern whose columns all have weight 8.

Next, let us consider the number of positions in which each pair of columns in P_i intersect. Since each pair of columns in P intersect in either 1, 2, 3, or 4 positions, each pair of columns in P_i must intersect in ≤ 4 positions. As with the column weights, we can also use the set of rows $\mathcal{R}(i)$ to check whether or not it is possible to complete P_i to a left pattern whose columns intersect in the required range. For example, if columns c_1 and c_2 of P_i intersect in 0 positions, then since the columns of any left pattern must intersect in at least 1 position, $\mathcal{R}(i)$ must contain a left word that has a value of 1 in both columns c_1 and c_2. We will use the Boolean function *ValidColumnIntersections*$(P_i, \mathcal{R}(i))$ to check whether or not P_i can be completed to a left pattern whose columns intersect in the required range.

Let us next consider how we can use the weight 4 words in the left word block W to

check whether or not P_i can be completed to a valid left pattern. Let $\vec{w}$ be any weight 4 word in W. Let T_i denote the $i \times 4$ matrix that consists of the 4 columns in P_i that are supported by $\vec{w}$. Let P be any left pattern whose first i rows are P_i. Finally, let T denote the 22×4 matrix that consists of the 4 columns in P that are supported by $\vec{w}$. By Theorem 3.12, we know T contains exactly 6 weight 0 rows. Theorem 3.12 also tells us the number of times each of the different weight 2 rows can occur in T. Let r_1, r_2, and r_3 denote the three weight 2 binary strings of length 4 that have a value of 0 in the first position. Let x_i denote the number of times r_i occurs as a row in T. By Theorem 3.12, $\{x_1, x_2, x_3\}$ must be one of the three sets $\{4,3,1\}, \{4,2,2\}, \{3,3,2\}$. Let $\overline{r}_i$ denote the binary string that is the complement of r_i. Let $\overline{x}_i$ denote the number of times $\overline{r}_i$ occurs as a row in T. By Theorem 3.12, we must have $\overline{x}_i = x_i$. We can use each of these facts about the matrix T to eliminate some of the partial left patterns P_i we produce. For example, if the binary strings r_1 and r_2 both occur in T_i four times then T_i cannot be completed to a matrix T in which $\{x_1, x_2, x_3\}$ is one of $\{4,3,1\}, \{4,2,2\}, \{3,3,2\}$, and thus P_i cannot be completed to a valid left pattern. We will use the Boolean function *ValidWeightFourSupports*(P_i) to determine whether or not the partial left pattern P_i can be completed to a left pattern P in which every set of four columns in P that are supported by a weight 4 word in W are valid.

Similarly, for each weight 5 word $\vec{w}$ in W, we can use the fact that, by Theorem 3.14, there are 7 different possibilities for the columns of the left pattern that are supported by $\vec{w}$ to eliminate some of the partial left patterns P_i we produce. We will use the Boolean function *ValidWeightFiveSupports*(P_i) to determine whether or not P_i can be completed to a left pattern P in which every set of five columns in P that are supported by a weight 5 word in W are valid.

Finally, for each partial left patterns P_i, we verify that each $z \times n(W)$ matrix M is "valid," where the rows of M are any combination of z rows in P_i that include row i of P_i. What we mean by M being valid is that if N is a $z \times 33$ matrix whose first $n(W)$ columns are M, then M is valid if we can "fill in" the remaining $33 - n(W)$ columns of N to

produce a set of rows that may occur in an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD (i.e. an N whose rows have weight 12 and intersect in 4 positions). Testing has shown that this check is practical for $z \leq 5$. We will use the Boolean function *ValidRowCombs*(P_i) to determine whether or not each combination of five or fewer rows in P_i, that includes row i, is valid.

This concludes our discussion of how we determine whether or not a partial left pattern P_i produced by our algorithm can be completed to a valid left pattern. We summarize each of the points made in our discussion with Algorithm 4.19.

Algorithm 4.19. boolean *ValidPartialLeftPattern*$(P_i, \mathcal{R}(i))$:

- Input: the partial left pattern P_i and the set $\mathcal{R}(i)$ of left words in $\mathcal{LEFT}(W)$ that are greater than or equal to row i of P_i.
- Output: returns *false* if P_i is an invalid partial left pattern (i.e. if P_i can only be completed to left patterns that we know cannot occur in the first $n(W)$ columns of an incidence matrix $A(W)$ for a $(22, 33, 12, 8, 4)$-BIBD), otherwise *true* is returned.

begin
valid = *ValidRowIntersections*(P_i) **and**
ValidColumnWeights$(P_i, \mathcal{R}(i))$ **and**
ValidColumnIntersections$(P_i, \mathcal{R}(i))$ **and**
ValidWeightFourSupports(P_i) **and**
ValidWeightFiveSupports(P_i) **and**
ValidRowCombs(P_i).
Return *valid*.
end

4.3.5 A Formal Description of our Left Pattern Algorithm

We are now ready for a formal description of our Left Pattern Algorithm. The algorithm is recursive and is begun with the call $EnumerateLeftPatterns(P_0, \mathcal{LEFT}(W), L(W))$, where $L(W)$ is initially empty.

Algorithm 4.20. $EnumerateLeftPatterns(P_{i-1}, \mathcal{R}(i-1), L(W))$:

- Input: the valid partial left pattern P_{i-1} and the set $\mathcal{R}(i-1)$ of all left words in $\mathcal{LEFT}(W)$ that are greater than or equal to row $i-1$ of P_{i-1}.
- Output: the list $L(W)$ of minimal increasing valid left patterns.

begin
  if $i-1 = 22$ **then**
    Insert P_{i-1} into $L(W)$.
  else
    for each left word l in $\mathcal{R}(i-1)$ **do**
      Append l to the end of P_{i-1}, producing the partial left pattern P_i.
      Find $\mathcal{R}(i)$, the set of all left words in $\mathcal{R}(i-1)$ that are greater than or equal to the left word l.
      if $ValidPartialLeftPattern(P_i, \mathcal{R}(i))$ **and** $MinIncPartialLeftPattern(P_i)$ **then**
        Call $EnumerateLeftPatterns(P_i, \mathcal{R}(i), L(W))$.
      end if
    end for
  end else
end

We have run our Left Pattern Algorithm (with optimizations) on each of the left word blocks given in Figure 4.1. We have also run our Left Pattern Algorithm on the

two left word blocks, the d_2^{**} and d_1^{**}-blocks, given in Figure 4.3. The results are given in Table 4.1. For each left word block W, Table 4.1 gives $|L(W)|$, the number of left patterns in the complete list $L(W)$ of inequivalent left patterns of W produced by our program. The table also gives the running time of our program for each left word block W. The program was run on a Solaris 2.5.

As Table 4.1 demonstrates, for each of the d_4, d_3, d_3^*, d_2, d_2^*, d_1, and f_1-blocks, the number of left patterns produced by the algorithm was relatively small. Furthermore, as expected, the number of left patterns produced for the d_1 and f_1-blocks agree with Theorems 3.12 and 3.14, respectively. On the other hand, the number of left patterns produced by the algorithm for each of the d_2^{**}, d_1^* and d_1^{**}-blocks was relatively large. For the cases of the d_2^{**} and d_1^*-blocks, this was due to the fact that for each choice for the first $n(W)-3$ columns of a left pattern, there are a relatively large number of choices for the last three columns of the left pattern (i.e. the columns that are supported by a weight 5 word, but not a weight 4 word). For the d_1^{**}-block, the reason for the larger number of left patterns is the fact that the weight 4 words in the block do not intersect.

We will not consider the d_2^{**}-block in our BIBD search algorithm since any code that contains a d_2^{**}-block must also contain a d_2-block (i.e. the d_2^{**}-block with the weight 5 word removed) and testing has shown us that, due to the large number of left patterns associated with the d_2^{**}-block, the algorithm runs faster if it uses the d_2-block instead of the d_2^{**}-block. Though more testing is required for the d_1, d_1^*, and d_1^{**}-blocks, we have found that our BIBD search algorithm works best with the d_1^*-block, followed by the d_1-block, and lastly the d_1^{**}-block. Since any code that contains a d_1^{**}-block must also contain a d_1-block, we will not consider the d_1^{**}-block in our algorithm. However, since any code that contains a d_1-block does not necessarily contain a d_1^*-block, we will consider both the d_1 and d_1^*-blocks in our algorithm.

1	1	1	1	0	0	0	0	0	0*		1	1	1	1	0	0	0	0	0	0*
1	1	0	0	1	1	0	0	0	0*		0	0	0	0	1	1	1	1	0	0*
1	1	0	0	0	0	1	1	1	0*		1	1	0	0	1	1	0	0	1	0*
				the d_2^{**}-block											the d_1^{**}-block					

Figure 4.3: The d_2^{**} and d_1^{**}-blocks.

W	$\|L(W)\|$	running time
d_4	14	8.6 min
d_3	20	12.1 min
d_3^*	2	35.8 sec
d_2	11	8.8 min
d_2^*	8	5.6 min
d_2^{**}	3732	39.8 hrs
d_1	3	1.4 min
d_1^*	346	4.3 hrs
d_1^{**}	14045	162.0 hrs
f_1	7	7.8 min

Table 4.1: Results of our Left Pattern Algorithm for each of the left word blocks W given in Figures 4.1 and 4.3.

4.4 Selecting a Word Block

Let L_C denote our list of 478 inequivalent $(33, 16)$ doubly-even self-orthogonal codes that we have not eliminated theoretically. Let C be a code in L_C. Then $C^{\perp}$ may contain many word blocks that are equivalent to one or more of the left word blocks in Figure 4.1. However, our algorithm for searching for an incidence matrix in C will only use one of the word blocks in $C^{\perp}$. Therefore, before we begin our search, we must select the word block in $C^{\perp}$ that our search algorithm will work with. In this section, we will discuss which word block in $C^{\perp}$ we select.

The choice of the word block can have a great effect on the running time of our BIBD search algorithm. Therefore, we will attempt to select the word block in $C^{\perp}$ that minimizes the running time of our BIBD search algorithm.

The main criteria we will consider in selecting our word block is the *type* of the word block. We consider two word blocks to be of the same *type* if they are equivalent. In order to quantify this concept, for a given word block W, we will use the notation $\mathcal{TYPE}(W)$ to denote the type of W. The value we assign to $\mathcal{TYPE}(W)$ is the name of the left word block in Figure 4.1 to which W is equivalent. For example, if W is equivalent to the d_4-block then $\mathcal{TYPE}(W) = d_4$.

The type of word block we select can have a great effect on the performance of our BIBD search algorithm. For example, if $C^{\perp}$ contains the word blocks W' and W'', where $\mathcal{TYPE}(W') = d_4$ and $\mathcal{TYPE}(W'') = d_1$, then our BIBD search algorithm will run much faster if we select W' instead of W''.

There are two factors to consider in determining which type of word block we should select. The first factor is that for a given left word block W and code C in $\mathcal{C}(W)$, our BIBD search algorithm will search for an incidence matrix in C that begins with the left pattern P, for each P in $L(W, C)$. (Recall that $L(W, C)$ is our list of left patterns of W with the property that C contains an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD if

and only if C contains an incidence matrix that begins with one of the left patterns in $L(W, C)$.) As we shall see in the next section, the number of left patterns in $L(W, C)$ is largely determined by the left word block W (and to a lesser extent, the partial left automorphism group of W and C). The second factor we must consider is that the number of distinct left words of W is not the same for the different left word blocks W. The relevance of this factor is that, generally speaking, the larger the number of left words of W there are, the faster our search algorithm will be for a given left pattern P in $L(W, C)$. The reason for this is that, generally speaking, the larger the number of left words l we have, the smaller the sets of right words $\mathcal{RIGHT}(W, C, l)$ will be, which in turn reduces the number of choices we have for the last $33 - n(W)$ columns of the rows of an incidence matrix that begins with P.

Ideally, we would like the type of word block we select to result in $L(W, C)$ containing the smallest number of left patterns. We would also ideally like the type of word block we select to have the largest number of left words associated with it. However, typically, the larger the number of left words, the larger the number of left patterns in $L(W, C)$. Therefore, in determining which type of word block to select, a balance must be struck between these two factors.

Through testing, keeping the above two factors in mind, we have established a precedence relationship among the different types of word blocks, with the word blocks with higher precedence being the ones that result in our BIBD search algorithm generally running faster. From highest to lowest, this precedence relationship is:

$$d_4, d_3^*, d_2^*, d_3, d_2, d_1^*, d_1, f_1.$$

In deciding which word block in $C^\perp$ to select, we first find all the different types of word blocks that $C^\perp$ contains. The word block we select will have the highest precedence among all the different types of word blocks in $C^\perp$.

Let T denote the type of the highest precedenced word block in $C^\perp$. Then $C^\perp$ may contain more than one word block W' in which $\mathcal{TYPE}(W') = T$. Therefore, once we have

determined T, we then select one word block from the set of all word blocks W' in $C^{\perp}$ in which $\mathcal{TYPE}(W') = T$. Simply put, the word block we select is the one that results in the largest set of right words $\mathcal{RIGHT}(W, C, l)$ having the least number of words in it. (If $C^{\perp}$ contains more than one such word block, we select the first one we encounter.) Whether or not this is the best one to select is a matter of further research. However, for the codes on which we have run our algorithm thus far, we have found that if $C^{\perp}$ does contain more than one type T word block, then the algorithm runs relatively quickly, and therefore, which of the type T word blocks we select is of little consequence.

Once we have selected the word block W' in $C^{\perp}$ that we will use with our BIBD search algorithm, we must then permute the coordinates of C in such a way that W' is permuted to the left word block W in Figure 4.1 to which W' is equivalent. This leaves us with a code that is an element of the set of codes $\mathcal{C}(W)$. Of course, we must also adjust the automorphism group of C to reflect the fact that we have permuted the coordinates of C.

Once we have selected the word block W' for each code C in L_C (and appropriately permuted the coordinates of C and $\mathcal{AUT}(C)$), we are left with a list of codes that can be partitioned into the sets $\mathcal{D}(d_4)$, $\mathcal{D}(d_3^*)$, $\mathcal{D}(d_2^*)$, $\mathcal{D}(d_3)$, $\mathcal{D}(d_2)$, $\mathcal{D}(d_1^*)$, $\mathcal{D}(d_1)$, and $\mathcal{D}(f_1)$, where $\mathcal{D}(W)$ is the set of codes in which the selected word block W' was permuted to the left word block W. Each set $\mathcal{D}(W)$ consists of all the codes D in which W is the left word block in $D^{\perp}$ our BIBD search algorithm will use in its search for an incidence matrix in D.

In Algorithm 4.21, we give a formal description of our algorithm for both selecting the word block in each code in L_C our BIBD search algorithm will use, and producing the sets of codes $\mathcal{D}(W)$, for each left word block W in Figure 4.1. Note that in our algorithm the elements in the list L_C and the sets $\mathcal{D}(W)$ are in fact pairs $(C, \mathcal{AUT}(C))$, where C is a code and $\mathcal{AUT}(C)$ is the automorphism group of C. (Of course, C is in fact stored with a generator matrix, while $\mathcal{AUT}(C)$ is stored with a set of generating permutations.)

Algorithm 4.21. *ProduceSetsOfCodesToSearch*:

- Input: our list L_C of pairs $(C, \mathcal{AUT}(C))$, where C is a $(33, 16)$ doubly-even self-orthogonal code and $\mathcal{AUT}(C)$ is the automorphism group of C. The list L_C contains one and only one pair for each of the 478 inequivalent $(33, 16)$ doubly-even self-orthogonal codes C that we have not eliminated theoretically.
- Output: for each of our left word blocks W, the set $\mathcal{D}(W)$ of pairs $(D, \mathcal{AUT}(D))$, where the code D has the property that $D^\perp$ contains W. Of course, the codes in the union of the sets $\mathcal{D}(W)$ give us a complete list of inequivalent $(33, 16)$ doubly-even self-orthogonal codes that we have not eliminated theoretically.

begin
for each left word block W in Figure 4.1 **do**
Set $\mathcal{D}(W) = \emptyset$.
for each pair $(C, \mathcal{AUT}(C))$ in L_C **do**
Find T, the type of the largest precedenced word block in $C^\perp$.
Select from the set of all type T word blocks in $C^\perp$, the word block W' that we will use in our BIBD search algorithm.
Let π be a permutation of the coordinates of C that permutes W' to the left word block W in Figure 4.1 to which W' is equivalent.
Apply π to C, producing the code D.
Apply π to $\mathcal{AUT}(C)$, producing $\mathcal{AUT}(D)$.
Insert $(D, \mathcal{AUT}(D))$ into $\mathcal{D}(W)$.
end for
end

Note that, for a given code C in $\mathcal{D}(W)$, even though we will use the left word block W to search for an incidence matrix in C, we will still classify C based on the weight 4

W	class of codes	number of codes in $\mathcal{D}(W)$ from each class
d_4	d_4	37
d_3^*	d_3	17
d_2^*	d_3	32
	d_2	83
d_3	d_3	44
d_2	d_2	122
d_1^*	d_1	112
d_1	d_1	20
f_1	distance 8	11

Table 4.2: The number of codes, from a given class, in each set $\mathcal{D}(W)$, for each left word block W.

blocks it contains (as discussed in Section 3.4). For example, if the largest precedenced weight 4 block in C is a d_3-block, but W is a d_2^*-block (which has higher precedence than the d_3-block), then we will still refer to C as a d_3-code. With this convention, the set of codes $\mathcal{D}(W)$ may contain more than one class of codes, and, there may be more than one set of codes $\mathcal{D}(W)$ that contains codes from a given class. In Table 4.2, for each left word block W, we give the number of codes in $\mathcal{D}(W)$ that are from each of the different classes of codes. For example, for the set $\mathcal{D}(d_2^*)$, Table 4.2 gives the number of codes in $\mathcal{D}(d_2^*)$ that are d_3-codes and the number of codes in $\mathcal{D}(d_2^*)$ that are d_2-codes.

We will now conclude this section by briefly discussing what we can learn about the point code of a $(22, 33, 12, 8, 4)$-BIBD even if we are not successful in searching for an incidence matrix in every code in each of the sets $\mathcal{D}(W)$.

We can prove that any $(33, k)$ doubly-even self-orthogonal code that contains a d_4-block cannot contain an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD by simply searching all the codes in $\mathcal{D}(d_4)$. That is, if for each of the d_4-codes in $\mathcal{D}(d_4)$ our BIBD search algorithm does not find an incidence matrix, then since we have eliminated all other codes that may contain a d_4-block (i.e. the e_3-codes, the e_{2i}-codes, where $i \geq 2$, and the d_i-codes, where $i \geq 5$), we know any $(33, 16)$-code, and thus any $(33, k)$ doubly-even self-orthogonal code that contains a d_4-block, cannot contain an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD. This, of course, implies that the point code of a $(22, 33, 12, 8, 4)$-BIBD cannot contain a d_4-block. Similarly, we can prove that the point code of a $(22, 33, 12, 8, 4)$-BIBD cannot contain a d_3-block by simply searching all of the d_3-codes (and the d_4-codes) in the sets $\mathcal{D}(W)$.

4.5 Left Patterns and the Code

In the last two sections, we described the work we perform before we begin searching for an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD in a given code C. In the next three sections, we will describe our algorithm that, for a given left word block W and code $C \in \mathcal{D}(W)$, searches the weight 12 words of C for an incidence matrix $A(W)$ for a $(22, 33, 12, 8, 4)$-BIBD.

Recall that for a given left word block W and code $C \in \mathcal{D}(W)$, our search algorithm works by first producing a list of left patterns $L(W, C)$, and then, for each P in $L(W, C)$, searches C for an incidence matrix $A(W)$ that begins with the left pattern P. In this section, we will determine which left patterns we need to include in the list $L(W, C)$. That is, we will have a look at how we can produce a list $L(W, C)$ of left patterns of W, with the property that C contains an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD if and only if C contains an incidence matrix $A(W)$ that begins with one of the left patterns in $L(W, C)$.

Let $L(W)$ denote our complete list of inequivalent left patterns that we produced by

running Algorithm 4.20 on the left word block W. We will begin by showing that we cannot simply use $L(W)$ for our list $L(W, C)$. Suppose the code C contains an incidence matrix A for a $(22, 33, 12, 8, 4)$-BIBD. Let P be the left pattern in the first $n(W)$ columns of A. Then $L(W)$ must contain a left pattern P' that is equivalent to P. That is, there must exist a P' in $L(W)$ and a permutation $\pi_l \in \mathcal{AUT}(W)$ such that $\pi_l^{n(W)}P$ and P' are the same, up to row rearrangement. However, the existence of an incidence matrix A in C that begins with P does not necessarily imply the existence of an incidence matrix in C that begins with the left pattern $\pi_l^{n(W)}P$. That is, there may not exist an automorphism π of C in which the left pattern in the first $n(W)$ columns of πA is equal to $\pi_l^{n(W)}P$. In other words, π_l may not necessarily be an element of $\mathcal{AUT}_{PART\_LEFT}(W, C)$. (Recall that $\mathcal{AUT}_{PART\_LEFT}(W, C)$ is the subset of permutations $\pi_l \in \mathcal{AUT}(W)$ for which there exists a permutation π_r that does not move the first $n(W)$ coordinates of C, such that $\pi_l\pi_r$ is an automorphism of C.)

Example 4.22. Suppose W is a d_3-block:

$$\begin{array}{cccccccccc} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0^* \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0^* \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0^* \end{array}$$

Then $\pi_l = (7\ 8)$ is an element of $\mathcal{AUT}(d_3)$. However, we have found there exist codes C in $\mathcal{D}(d_3)$ in which $(7\ 8) \notin \mathcal{AUT}_{PART\_LEFT}(d_3, C)$. Furthermore, there exist left patterns P in which P and $(7\ 8)P$ are not the same, up to row rearrangement. For example, the left patterns P and $(7\ 8)P$ in Figure 4.4 are not the same, up to row rearrangement. Therefore, the existence of an incidence matrix in C that begins with P does not necessarily imply the existence of an incidence matrix in C that begins with $(7\ 8)P$.

Therefore, for each left pattern P in $L(W)$, it is not sufficient to simply look for an incidence matrix that begins with the left pattern P.

Conversely, if for a given left pattern P in $L(W)$, we look for an incidence matrix that begins with the left pattern P', for each P' is the set $\{\pi^{n(W)}P \mid \pi \in \mathcal{AUT}(W)\}$ of

$$P = \begin{bmatrix}
0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&1\\
0&0&0&0&0&0&1&1\\
0&0&0&0&1&1&0&0\\
0&0&0&0&1&1&0&0\\
0&0&1&1&0&0&0&0\\
0&0&1&1&0&0&0&0\\
1&1&0&0&0&0&0&0\\
1&1&0&0&0&0&0&0\\
0&1&0&1&0&1&0&1\\
0&1&0&1&0&1&0&1\\
0&1&0&1&0&1&1&0\\
0&1&0&1&1&0&1&0\\
0&1&1&0&1&0&0&1\\
0&1&1&0&1&0&0&1\\
1&0&0&1&1&0&1&0\\
1&0&0&1&1&0&1&0\\
1&0&1&0&0&1&0&1\\
1&0&1&0&0&1&1&0\\
1&0&1&0&0&1&1&0\\
1&0&1&0&1&0&0&1
\end{bmatrix} \quad (7\ 8)P = \begin{bmatrix}
0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&1\\
0&0&0&0&0&0&1&1\\
0&0&0&0&1&1&0&0\\
0&0&0&0&1&1&0&0\\
0&0&1&1&0&0&0&0\\
0&0&1&1&0&0&0&0\\
1&1&0&0&0&0&0&0\\
1&1&0&0&0&0&0&0\\
0&1&0&1&0&1&1&0\\
0&1&0&1&0&1&1&0\\
0&1&0&1&0&1&0&1\\
0&1&0&1&1&0&0&1\\
0&1&1&0&1&0&1&0\\
0&1&1&0&1&0&1&0\\
1&0&0&1&1&0&0&1\\
1&0&0&1&1&0&0&1\\
1&0&1&0&0&1&1&0\\
1&0&1&0&0&1&0&1\\
1&0&1&0&0&1&0&1\\
1&0&1&0&1&0&1&0
\end{bmatrix}$$

Figure 4.4: Two left patterns of the d_3-block that are equivalent but not equal, up to row rearrangement.

all left patterns that are equivalent to P, then our algorithm may wind up performing a lot of unnecessary work. That is, there may exist equivalent left patterns, P and P', with the property that C contains an incidence matrix that begins with P if and only if C contains an incidence matrix that begins with P'.

In conclusion, for a given left pattern P in $L(W)$, it is inefficient to include every left pattern that is equivalent to P in our list $L(W,C)$, and insufficient to only include P in $L(W,C)$. So, for a given P in $L(W)$, let us now determine which of the left patterns that are equivalent to P we need to include in our list $L(W,C)$.

Since the existence in C of an incidence matrix that begins with a left pattern P implies the existence of an incidence matrix in C that begins with $\pi_r P$, where π_r is any permutation of the rows in P, we will only consider left patterns that are *increasing* left patterns (as defined on page 104). For a given left pattern P in $L(W)$, we will use the notation $S_{INC}(W,P)$ to denote the set of all increasing left patterns that are equivalent to P. That is, $S_{INC}(W,P) = \{\pi_r \pi_c^{n(W)} P \mid \pi_c \in \mathcal{AUT}(W)$ and π_r is any row permutation that sorts the rows of $\pi_c^{n(W)} P$ in increasing order $\}$.

In order to determine which left patterns in $S_{INC}(W,P)$ we need to include in $L(W,C)$, let us first write the group of permutations $\mathcal{AUT}(W)$ in terms of the partial left automorphism group of W and C. Let $G = \mathcal{AUT}(W)$ and let $H = \mathcal{AUT}_{PART\_LEFT}(W,C)$. By our definition of partial left automorphism in Section 4.1.3, we know H is a subgroup of G. Therefore, we can write G as the union of $m = |G|/|H|$ right cosets of H:

$$G = H\pi_1 \cup H\pi_2 \cup \cdots \cup H\pi_m$$

where $\pi_i \in G$ and $H\pi_i \cap H\pi_j = \emptyset$, for $i \neq j$. We will refer to the set of permutations $\{\pi_1, \pi_2, \ldots, \pi_m\}$ as a *set of coset leaders* for the left word block W and code C. We will use the notation $\mathcal{COSET}(W,C)$ to denote any set of coset leaders for W and C.

For a given left pattern P and permutation $\pi \in \mathcal{AUT}(W)$, we will use the notation $S_{PART\_LEFT}(W,C,P,\pi)$ to denote the set of all left patterns $\pi_r \pi_c^{n(W)} P$, where $\pi_c \in H\pi$

and π_r is any row permutation that sorts the rows of $\pi_c^{n(W)}P$ in increasing order. Using this notation, we can now rewrite our complete set $S_{INC}(W,C)$ of left patterns that are equivalent to P, up to row rearrangement, as follows:

$$\begin{aligned} S_{INC}(W,P) &= S_{PART\_LEFT}(W,C,P,\pi_1) \cup S_{PART\_LEFT}(W,C,P,\pi_2) \cup \cdots \\ &\quad \cup S_{PART\_LEFT}(W,C,P,\pi_m) \end{aligned}$$

where $\{\pi_1, \pi_2, \ldots, \pi_m\} = \mathcal{COSET}(W,C)$.

We will now show that we only need to include, in our list $L(W,C)$, one left pattern from each set $S_{PART\_LEFT}(W,C,P,\pi)$, where $\pi \in \mathcal{COSET}(W,C)$. Towards this goal, we first require the following lemma:

Lemma 4.23. *Let P be any left pattern of W and let π_l be a partial left automorphism of W and C. Then C contains an incidence matrix that begins with the left pattern P if and only if C contains an incidence matrix that begins with the left pattern $\pi_l^{n(W)}P$.*

Proof. By our definition of partial left automorphism, since π_l is a partial left automorphism of W and C, we know there exists a permutation π_r, that does not move the first $n(W)$ coordinates of C, such that $\pi_l\pi_r$ is an automorphism of C. Since $\pi_l\pi_r$ is an automorphism of C, we know C contains an incidence matrix A if and only if C contains an incidence matrix $\pi_l\pi_r A$. Thus, since π_r does not move the first $n(W)$ coordinates of C, C contains an incidence matrix that begins with P if and only if C contains an incidence matrix that begins with $\pi_l^{n(W)}P$. □

We are now ready to show that we only need to include in $L(W,C)$, one left pattern from each set $S_{PART\_LEFT}(W,C,P,\pi)$, where $\pi \in \mathcal{COSET}(W,C)$.

Theorem 4.24. *Let $\pi \in \mathcal{AUT}(W)$ and let $P_1, P_2 \in S_{PART\_LEFT}(W,C,P,\pi)$. Then C contains an incidence matrix that begins with P_1 if and only if C contains an incidence matrix that begins with P_2.*

Proof. Let $H = \mathcal{AUT}_{PART\_LEFT}(W, C)$. By our definition of $S_{PART\_LEFT}(W, C, P, \pi)$, since $P_1 \in S_{PART\_LEFT}(W, C, P, \pi)$ we know there exists a permutation $\pi_{c_1} \in H\pi$ and a row permutation π_{r_1} such that $P_1 = \pi_{r_1}\pi_{c_1}^{n(W)}P$. Similarly, there exists a permutation $\pi_{c_2} \in H\pi$ and a row permutations π_{r_2} such that $P_2 = \pi_{r_2}\pi_{c_2}^{n(W)}P$. Since $\pi_{c_1}, \pi_{c_2} \in H\pi$, we know $\pi_{c_1} = \pi_1\pi$ and $\pi_{c_2} = \pi_2\pi$ for some permutations $\pi_1, \pi_2 \in H$. Consider the permutation $\pi' = \pi_2\pi_1^{-1}$. Since H is a group, we know $\pi' \in H$. Therefore, by Lemma 4.23, we know C contains an incidence matrix that begins with the left pattern $\pi_{c_1}^{n(W)}P$ if and only if C contains an incidence matrix that begins with the left pattern $(\pi')^{n(W)}(\pi_{c_1}^{n(W)}P) = (\pi'\pi_{c_1})^{n(W)}P$. Since $\pi'\pi_{c_1} = (\pi_2\pi_1^{-1})(\pi_1\pi) = \pi_2(\pi_1^{-1}\pi_1)\pi = \pi_2\pi = \pi_{c_2}$, this implies C contains an incidence matrix that begins with $\pi_{c_1}^{n(W)}P$ if and only if C contains an incidence matrix that begins with $\pi_{c_2}^{n(W)}P$. Since P_1 and P_2 are equal to $\pi_{c_1}^{n(W)}P$ and $\pi_{c_2}^{n(W)}P$, respectively, up to row rearrangement, this implies C contains an incidence matrix that begins with P_1 if and only if C contains an incidence matrix that begins with P_2. □

Let us now summarize what we have just found. Let P be a left pattern in $L(W)$. Then the set:

$$S_{INC}(W, P) = \bigcup_{\pi \in \mathcal{COSET}(W,C)} S_{PART\_LEFT}(W, C, P, \pi)$$

contains, up to row rearrangement, every left pattern that is equivalent to P. Furthermore, by Theorem 4.24, for a given π in $\mathcal{COSET}(W, C)$, if P_1 and P_2 are any two left patterns in $S_{PART\_LEFT}(W, C, P, \pi)$ then C contains an incidence matrix that begins with P_1 if and only if C contains an incidence matrix that begins with P_2. Therefore, for a given P in $L(W)$, we only need to include one left pattern from each set $S_{PART\_LEFT}(W, C, P, \pi)$, where $\pi \in \mathcal{COSET}(W, C)$, in our list $L(W, C)$. Thus, if we let $Q(W, C, P, \pi)$ denote the left pattern in $S_{PART\_LEFT}(W, C, P, \pi)$ that we select, we can then state that C contains an incidence matrix that begins with a left pattern that is equivalent to P if and only if C contains an incidence matrix that begins with a left pattern in the set $\{Q(W, C, P, \pi) \mid \pi \in \mathcal{COSET}(W, C)\}$.

Let us now consider which left pattern in $S_{PART\_LEFT}(W, C, P, \pi)$ would should include in our list $L(W, C)$. Now, we have found there may exist permutations $\pi_1, \pi_2 \in \mathcal{COSET}(W, C)$ in which $S_{PART\_LEFT}(W, C, P, \pi_1) = S_{PART\_LEFT}(W, C, p, \pi_2)$. Therefore, if we arbitrarily select the left pattern $Q(W, C, P, \pi)$, for each $\pi \in \mathcal{COSET}(W, C)$, then our list $L(W, C)$ may contain more left patterns than necessary. Therefore, for each set $S_{PART\_LEFT}(W, C, P, \pi)$, we will select the *smallest* left pattern in the set, which we will denote by $Q_{min}(W, C, P, \pi)$, for inclusion in $L(W, C)$. What we mean by *smallest* is the unique left pattern in $S_{PART\_LEFT}(W, C, P, \pi)$ that is less than or equal to all other left patterns in the set (what is meant by "less than or equal to" is defined in Section 1.1.6).

Using our notation, we can now state that C contains an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD if and only if C contains an incidence matrix that begins with one of the left patterns in the set $\{Q_{min}(W, C, P, \pi) \mid \pi \in \mathcal{COSET}(W, C)$ and P is in $L(W)\}$. This give us our list of left patterns $L(W, C)$.

We will now conclude this section by giving a formal description of our algorithm for producing our list of left patterns $L(W, C)$.

Algorithm 4.25. *FindLeftPatternsForCode*:

- Input: A left word block W, a code $C \in \mathcal{D}(W)$, the automorphism group $\mathcal{AUT}(W)$ of W, the partial left automorphism group $\mathcal{AUT}_{PART\_LEFT}(W, C)$ of W and C, and $L(W)$, the complete list of inequivalent left patterns of W produced by Algorithm 4.20.
- Output: A list $L(W, C)$ of left patterns with the property that if there exists an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD in C then there exists an incidence matrix in C that begins with one of the left patterns in $L(W, C)$.

begin

Clear the list $L(W, C)$.

Use the group $\mathcal{AUT}(W)$ and the subgroup $\mathcal{AUT}_{PART\_LEFT}(W, C)$ to find a set $\mathcal{COSET}(W, C)$ of right coset leaders for W and C.
for each left pattern P in $L(W)$ **do**
for each permutation π in $\mathcal{COSET}(W, C)$ **do**
Produce the set $S_{PART\_LEFT}(W, C, P, \pi)$.
Find P', the smallest left pattern in $S_{PART\_LEFT}(W, C, P, \pi)$.
if P' is not an element of the list $L(W, C)$ **then**
Insert P' into the list $L(W, C)$.
end if
end for
end for
end

4.6 The BIBD Search Algorithm

Let W be a left word block, let C be a code in $\mathcal{D}(W)$, and let P be a left pattern in $L(W, C)$. In this section, we will describe in detail our algorithm for determining whether or not C contains an incidence matrix $A(W)$ that begins with the left pattern P. Throughout this section, we will refer to this algorithm as our *BIBD search algorithm.* That is, whenever we use the term *BIBD search algorithm* in this section, we are referring to our algorithm for determining if a particular code C contains an incidence matrix that begins with a particular left pattern P in $L(W, C)$ (and not our overall algorithm for determining whether or not C contains an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD).

Let us begin with a brief outline of how our BIBD search algorithm works. Let A be a 22×33 matrix whose first $n(W)$ columns contain the left pattern P. Let l_i denote the left word in the first $n(W)$ columns of row i of A. Our BIBD search algorithm works by trying to recursively fill in, row by row, the last $33 - n(W)$ columns of each row in A using the facts that: (1) the last $33 - n(W)$ columns of row i of A must be a right

word in the set $\mathcal{RIGHT}(W, C, l_i)$, and (2) every pair of rows in A must intersect in 4 positions. If the algorithm manages to completely fill in the last $33 - n(W)$ columns of A, it then checks whether or not each column of A has weight 8. If this is the case then the algorithm has found an incidence matrix A, for a $(22, 33, 12, 8, 4)$-BIBD, that begins with the left pattern P. If the algorithm does not find such a matrix A, then C does not contain an incidence matrix $A(W)$ that begins with the left pattern P.

In describing the details of our BIBD search algorithm, we will make use of the following notation: We will use A to denote our 22×33 matrix, that begins with the left pattern P, that we are trying to complete to an incidence matrix $A(W)$. We will use A_i, where $0 \leq i \leq 22$, to denote the matrix A with its first i rows completely filled. That is, we will use A_i to denote any 22×33 matrix A in which the first $n(W)$ columns of A contains P, the last $33 - n(W)$ columns of the first i rows of A contain appropriate right words, and the last $33 - n(W)$ columns of the last $22 - i$ rows of A contain nothing. We will use the notation l_m, where $1 \leq m \leq 22$, to denote the left word in the first $n(W)$ columns of row m of A_i (which, of course, is also the left word in row m of the left pattern P). Finally, we will use the notation r_m, where $1 \leq m \leq i$, to denote the right word in the last $33 - n(W)$ columns of row m of A_i.

Our BIBD search algorithm is a backtrack algorithm, and thus can be thought of as a preorder traversal of a tree T of matrices A_i. (In a preorder traversal, each node in the tree is visited *before* any of its children). The root of T is the 22×33 matrix A_0. Each node at level i of T, where $1 \leq i \leq 22$, is a distinct matrix A_i. The children of each A_i, where $0 \leq i \leq 21$, are all the matrices A_{i+1} whose first i rows are equal to A_i. That is, for each right word $r \in \mathcal{RIGHT}(W, C, l_{i+1})$ in which $l_{i+1}r$ intersects each of the first i rows of A_i in 4 positions, A_i has a child A_{i+1} that is equal to A_i with the right word r inserted into the last $33 - n(W)$ columns of row $i + 1$.

We will refer to the tree T as our *BIBD search tree* (for the code C and left pattern P). In describing many of the details of our BIBD search algorithm, we will find it beneficial

to think of our algorithm in terms of the BIBD search tree T.

Due to the potentially enormous size of the BIBD search tree T, we will prune T in our BIBD search algorithm. We use two different methods to prune T. One method uses the automorphism group of C to show that a given subtree T_2 does not contain a leaf node at level 22 of the tree T (and thus, the matrix A_i at the root of T_2 cannot be completed to an incidence matrix $A(W)$) based on the fact that a previously visited subtree T_1 did not contain a leaf node at level 22 of the tree. The other method determines that a subtree T_1 does not have a leaf node at level 22 of T based on the number of right words that are available for insertion into each of the last $22-i$ rows of the matrix A_i at the root of T_1.

4.6.1 The Basic Recursive Step of our BIBD Search Algorithm

We implement our BIBD search algorithm recursively. The input to the i^{th} level of the recursion is a matrix A_{i-1} and a list of sets $R_{i-1,m}$, for $m = i, i+1, \ldots, 22$. The set $R_{i-1,m}$ consists of all right words $r \in \mathcal{RIGHT}(W, C, l_m)$ that can be inserted into the last $33 - n(W)$ columns of row m of A_{i-1}. That is, the right word r is an element of $R_{i-1,m}$ if and only if $l_m r$ intersects each of the first $i-1$ rows of A_{i-1} in 4 positions.

For each right word r in the set $R_{i-1,i}$, the algorithm proceeds as follows: First, using the automorphism group of C, we determine whether or not inserting r into row i of A_{i-1} results in a matrix A_i that we know cannot be completed to an incidence matrix due to the fact that we were not able to complete some previously processed matrix. If this is not the case, we then continue by inserting r into the last $33 - n(W)$ columns of row i of A_{i-1}, producing a matrix A_i. For $m = i+1, i+2, \ldots, 22$, we then produce the sets $R_{i,m}$ by finding all the right words r' in $R_{i-1,m}$ in which $l_m r'$ intersects row i of A_i in 4 positions. We then check that the sets $R_{i,m}$ contain enough right words to, potentially, fill in the remaining rows of A_i. If this is the case, we then proceed to the next level of the recursion.

This gives us the basic recursive step of our BIBD search algorithm. In the remainder

of this section, we will have a detailed look at each of the operations we need to perform in our basic recursive step. That is, we will have a detailed look at how we produce the matrices A_i, how we produce the sets $R_{i,m}$, and how we prune the BIBD search tree using the automorphism group of C and the sets $R_{i,m}$. Since the basic recursive step may be called an enormous number of times, it is crucial that we implement each of these operations as efficiently as possible. Since the efficiency of these operations can be greatly affected by the manner in which we store the objects used by our algorithm, we will also describe how we store many of our objects.

4.6.2 Storing the Right Words of C

We will begin by discussing how we store the right words of C used by our BIBD search algorithm. More specifically, we will discuss how we store the right words in the sets $\mathcal{RIGHT}(W, C, l)$ and $R_{i,m}$, and the right words in the last $33 - n(W)$ columns of the matrix A we are trying to complete to an incidence matrix.

Let us first consider the sets $\mathcal{RIGHT}(W, C, l)$, where $l \in \mathcal{LEFT}(W)$. For the left word blocks W we use, we have found there exist many left words l_a and l_b in which $\mathcal{RIGHT}(W, C, l_a) = \mathcal{RIGHT}(W, C, l_b)$. This fact not only plays an important role in how we store the sets $\mathcal{RIGHT}(W, C, l)$, but also in how we both store and manipulate the sets $R_{i,m}$, as we shall later see. But first, we will demonstrate this fact with an example.

Example 4.26. Suppose W is a d_3-block:

$$\begin{array}{cccccccccc} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0^* \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0^* \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0^* \\ p_1 & & p_2 & & p_3 & & p_4 & & \end{array}$$

For this example, we will use the symbol p_i, where $1 \leq i \leq 4$, written below the d_3-block,

to denote the pair of consecutive equal coordinates in the d_3-block that is located above the symbol.

Let us first have a look at the left words in the set $\mathcal{LEFT}(d_3)$. Using the fact that each left word in $\mathcal{LEFT}(d_3)$ must intersect each word in the d_3-block in an even number of positions, but cannot intersect any of the weight 4 words in the d_3-block in 4 positions, it can easily be shown that the left words in $\mathcal{LEFT}(d_3)$ have weight 0, 2, or 4. This allows us to classify the left words in $\mathcal{LEFT}(d_3)$ into three different *types* of left words:

- the left word of weight 0, which we refer to as a type 0 left word,
- the left words with weight 2, which we refer to as type 1 left words, and
- the left words with weight 4, which we refer to as type 2 left words.

Each of the type 1 left words has a value of 1 1 in one of the pairs of coordinates p_i and 0 0 in all other pairs. Each of the type 2 left words has a value of either 1 0 or 0 1 in each pair p_i.

Now, it can easily be shown that for any code $C \in \mathcal{C}(d_3)$, if l_a and l_b are both type 1 left words then $\mathcal{RIGHT}(d_3, C, l_a) = \mathcal{RIGHT}(d_3, C, l_b)$. For example, if $l_a = 11000000$ and $l_b = 00110000$ then, since C contains the codeword 111100000*, we know $l_a r$ is a codeword in C if and only if $l_b r$ is a codeword in C.

It can also easily be shown that for any code $C \in \mathcal{C}(d_3)$, if l_a and l_b are both type 2 left words that contain an even number of pairs p_i with a value of 1 0 then $\mathcal{RIGHT}(d_3, C, l_a) = \mathcal{RIGHT}(d_3, C, l_b)$. This is also true if l_a and l_b both contain an odd number of pairs with a value of 1 0.

Together with the set $\mathcal{RIGHT}(d_3, C, l_a)$, where l_a is the type 0 left word, this gives us at most 4 distinct right sets $\mathcal{RIGHT}(d_3, C, l)$. The reason we say at most 4 is that, depending on the automorphism group of the code C, it may be true that for *any* type 2 left words l_a and l_b, we have $\mathcal{RIGHT}(d_3, C, l_a) = \mathcal{RIGHT}(d_3, C, l_b)$. If this is the case then there are 3 distinct right sets $\mathcal{RIGHT}(d_3, C, l)$.

We store all of the distinct right sets $\mathcal{RIGHT}(W, C, l)$ in any array of right words $RIGHT\_WORDS[0 \ldots num\_right\_words - 1]$, where num_right_words is the total number of right words in each of the distinct right sets. The right words in each distinct right set are stored in consecutive locations in $RIGHT\_WORDS[\,]$. In order to access the elements of a particular right set in $RIGHT\_WORDS[\,]$, we use two integer arrays, $RIGHT\_SET\_START[0 \ldots num\_sets - 1]$ and $RIGHT\_SET\_SIZE[0 \ldots num\_sets - 1]$, where num_sets is the number of distinct right sets. The first right word in the m^{th} right set in $RIGHT\_WORDS[\,]$ is located in position $RIGHT\_SET\_START[m-1]$, while the last right word in the m^{th} set is located in position $RIGHT\_SET\_START[m-1] + RIGHT\_SET\_SIZE[m-1] - 1$. The right words in each right set in $RIGHT\_WORDS[\,]$ are stored in *decreasing order* (as defined in Section 1.1.6).

The array $RIGHT\_WORDS[\,]$ contains all of the right words of C that our BIBD search algorithm uses. Therefore, if S is any collection of right words used by our algorithm, instead of physically storing each right word r in S, we will only store the position of r in the array $RIGHT\_WORDS[\,]$. That is, we will store r as the integer i, where $RIGHT\_WORDS[i] = r$. We will use the phrases "right word index" or "index of the right word r" to refer to the position i of r in the array $RIGHT\_WORDS[\,]$. We will use the notation $index(r)$ to denote the index i of the right word r.

We store the right words in the last $33 - n(W)$ columns of the matrix A (that we are trying to complete to an incidence matrix) as an array of right word indices $BIBD\_RIGHT\_INDEX[1 \ldots 22]$. That is, instead of physically inserting a right word r into the last $33 - n(W)$ columns of row i of A, our algorithm will simply assign the value of $index(r)$ to $BIBD\_RIGHT\_INDEX[i]$.

We also store each of the sets $R_{i,m}$ as an array of right word indices. However, as we shall soon see, since the right sets $\mathcal{RIGHT}(W, C, l)$ may not all be distinct, the sets $R_{i,i+1}, R_{i,i+2}, \ldots, R_{i,22}$ may not all be distinct. Therefore, separately storing every one of the sets $R_{i,m}$ may waste both time and space. In the next subsection, we will discuss

how we deal with the fact that the sets $R_{i,m}$ may not all be distinct.

4.6.3 Producing the Sets $R_{i,m}$

Recall that for a given matrix A_i produced during our BIBD search algorithm, where $0 \leq i \leq 21$, the set $R_{i,m}$, where $i+1 \leq m \leq 22$, consists of all right words r in $\mathcal{RIGHT}(W, C, l_m)$ in which $l_m r$ intersects each of the first i rows of A_i in 4 positions. Each set $R_{0,m}$ is simply the set $\mathcal{RIGHT}(W, C, l_m)$. Each set $R_{i,m}$, where $1 \leq i \leq 21$, is simply the set of all right words r in the set $R_{i-1,m}$ (that is associated with A_{i-1}, the parent of A_i) in which $l_m r$ intersects row i of A_i in 4 positions.

The reason we produce the set $R_{i,i+1}$ is that it consists of all the right words r in $\mathcal{RIGHT}(W, C, l_{i+1})$ that, when inserted into the last $33-n(W)$ columns of row $i+1$ of A_i, result in a matrix A_{i+1} whose $(i+1)^{th}$ row intersects each of its first i rows in 4 positions. There are two reasons we produce the remaining sets $R_{i,m}$, where $i+2 \leq m \leq 22$. First, as we shall see later in this section, these sets can be used to prune our BIBD search tree. Second, for any matrix A_{m-1} that is subsequently produced by the algorithm (i.e. any A_{m-1} that is a descendent of A_i) the set $R_{m-1,m}$ (associated with A_{m-1}) is a subset of the set $R_{i,m}$. Therefore, by producing the set $R_{i,m}$, we eliminate much of the redundant work that would result if we produced the set $R_{m-1,m}$ from scratch, for each of the matrices A_{m-1} subsequently produced by the algorithm.

As we saw in Example 4.26, for the left word blocks we use, there may exist many left words l_{m_1} and l_{m_2} in which $\mathcal{RIGHT}(W, C, l_{m_1}) = \mathcal{RIGHT}(W, C, l_{m_2})$. As we shall see next, this implies the sets $R_{i,i+1}, R_{i,i+2}, \ldots, R_{i,22}$ may not all be distinct. Furthermore, there may exist many sets, R_{i,m_1} and R_{i,m_2}, for which we can determine *before* we begin our search that R_{i,m_1} and R_{i,m_2} will always be equal, no matter how we fill in the last $33 - n(W)$ columns of the first i rows of A.

Example 4.27. Let W be a d_3-block and let P be the left pattern in Figure 4.5. As we saw in Example 4.26, for any $C \in \mathcal{C}(d_3)$, there are either three or four distinct right sets

$$P = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{matrix} l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \\ l_6 \\ l_7 \\ l_8 \\ l_9 \\ l_{10} \\ l_{11} \\ l_{12} \\ l_{13} \\ l_{14} \\ l_{15} \\ l_{16} \\ l_{17} \\ l_{18} \\ l_{19} \\ l_{20} \\ l_{21} \\ l_{22} \end{matrix}$$

Figure 4.5: A left pattern P of the d_3-block.

for the left words in $\mathcal{LEFT}(d_3)$. For this example, we will assume we have a code C in which there are three distinct right sets. Under this assumption, $\mathcal{RIGHT}(W, C, l_{m_1}) = \mathcal{RIGHT}(W, C, l_{m_2})$ if and only if the left words l_{m_1} and l_{m_2} have equal weight.

Let us first consider the sets $R_{0,m}$, for $m = 1, 2, \ldots, 22$. By our definition of $R_{i,m}$, the set $R_{0,m}$ is simply the set $\mathcal{RIGHT}(W, C, l_m)$, where l_m is the left word in row m of P. Since the left words in the first 12 rows of P all have weight 4, their right sets are equal, and thus $R_{0,1} = R_{0,2} = \cdots = R_{0,12}$. Similarly, $R_{0,13} = R_{0,14} = \cdots = R_{0,20}$ and $R_{0,21} = R_{0,22}$. Therefore, there are only 3 distinct sets $R_{0,m}$.

Let us now have at look at some of the sets $R_{i,m}$, where $i \geq 1$. Let A_i be any matrix that is produced at the i^{th} level of recursion, where $1 \leq i \leq 12$, of our BIBD search algorithm for our code C and left pattern P. Consider the sets $R_{i,13}, R_{i,14}, \ldots, R_{i,20}$ that are associated with the matrix A_i. Since $R_{0,13} = R_{0,14} = \cdots = R_{0,20}$, each set $R_{i,m}$, where $13 \leq m \leq 20$, consists of all the right words $r \in R_{0,13}$ in which $l_m r$ intersects each of the first i rows of A_i in 4 positions. Furthermore, since each of the left words $l_{13}, l_{14}, \ldots, l_{20}$ intersect each of the first 12 rows in P in exactly 1 position, $l_m r$ will intersect the first i rows of A_i in 4 positions if and only if the right word r intersects the right words in the first i rows of A in exactly 3 positions. Therefore, each set $R_{i,m}$, where $13 \leq m \leq 20$, consists of all of the right words in $R_{0,13}$ that intersect each of the right words in the first i rows of A_i in 3 positions. Thus, no matter how we fill in the last $33 - n(W)$ columns of the first i rows of A, where $1 \leq i \leq 12$, the set $R_{i,13}, R_{i,14}, \ldots, R_{i,20}$ will always be equal.

Of course, these are not the only sets $R_{i,m}$, where $i \geq 1$, that we can determine will always be equal before we begin our search. For example, since $\mathcal{RIGHT}(W, C, l_5) = \mathcal{RIGHT}(W, C, l_7)$ and since the left words l_5 and l_7 both intersect the first 4 rows in P in 2, 2, 1, and 2 positions, respectively, the sets $R_{i,5}$ and $R_{i,7}$ will always be equal, for $i = 1, 2, 3, 4$.

Let us now formalize our discussion in Example 4.27, to give us a description of how we can determine, before we begin our search, when two sets R_{i,m_1} and R_{i,m_2} will always

be equal. As Example 4.27 demonstrates, if for integers m_1 and m_2, where $i+1 \leq m_1, m_2 \leq 22$, the following two statements are true:

1. $\mathcal{RIGHT}(W, C, l_{m_1}) = \mathcal{RIGHT}(W, C, l_{m_2})$, and
2. the left words l_{m_1} and l_{m_2} both intersect each of the left words in the first i rows of the left pattern P in the same number of positions,

then no matter how we fill in the last $33-n(W)$ columns of the first i rows of A, producing the matrix A_i, the sets R_{i,m_1} and R_{i,m_2} will be equal. That is, since for $i_0 = 1, 2, \ldots, i$, we know l_{m_1} and l_{m_2} both intersect the left word l_{i_0} in the first $n(W)$ columns of row i_0 of A_i in the same number of positions, no matter what the right word r_{i_0} in the last $33-n(W)$ columns of row i_0 of A_i is, if r is a right word in $\mathcal{RIGHT}(W, C, l_{m_1})$ (which is equal to $\mathcal{RIGHT}(W, C, l_{m_2})$) then $l_{m_1}r$ intersects $l_{i_0}r_{i_0}$ in 4 positions if and only if $l_{m_2}r$ intersects $l_{i_0}r_{i_0}$ in 4 positions. Therefore r will be an element of R_{i,m_1} if and only if r is an element of R_{i,m_2}. Thus, the sets R_{i,m_1} and R_{i,m_2} will always be equal.

For the left patterns of the left word blocks we use, we have found there exists many pairs of integers, m_1 and m_2, for which we can determine, before we begin our search, that the sets R_{i,m_1} and R_{i,m_2} will always be equal. Therefore, if our algorithm separately stores and maintains every one of the sets $R_{i,m}$ then our algorithm may be performing a lot of unnecessary work that could have been prevented. Thus, we have found it to be beneficial to implement our algorithm in such a way that it does not spend time and space reproducing sets that we know will always be equal to sets that the algorithm has already produced.

In the remainder of this subsection, we will discuss how we prevent our algorithm from reproducing these sets. Towards this goal, we will first introduce the notation we will use to describe how our algorithm deals with these sets.

For a given integer m, where $i+1 \leq m \leq 22$, we will use the notation $S_{i,m}$ to denote the set of all integers x, where $i+1 \leq x \leq 22$, such that: (1) $\mathcal{RIGHT}(W, C, l_x) =$

$\mathcal{RIGHT}(W, C, l_m)$, and (2) l_x and l_m intersect each of the first i rows P in the same number of positions. Thus, $S_{i,m}$ consists of all integers x for which we can determine, before we begin our search, that $R_{i,m}$ and $R_{i,x}$ will be equal for each A_i produced by our algorithm. Of course, $S_{i,x} = S_{i,m}$ if and only if $x \in S_{i,m}$.

Example 4.28. Let us recursively find the sets $S_{i,m}$, for $i = 0, 1, 2, 3$, for the left pattern P and code C of Example 4.27. As we saw in Example 4.27, there are 3 distinct sets $R_{0,m}$, and thus, we have 3 distinct sets $S_{0,m}$:

$$\begin{aligned}
S_{0,1} &= \{1,2,3,4,5,6,7,8,9,10,11,12\} = S_{0,2} = S_{0,3} = \cdots = S_{0,12} \\
S_{0,13} &= \{13,14,15,16,17,18,19,20\} = S_{0,14} = S_{0,15} = \cdots = S_{0,20} \\
S_{0,21} &= \{21,22\} = S_{0,22}
\end{aligned}$$

Each set $S_{1,m}$, where $2 \leq m \leq 22$, consists of all the integers $x \in S_{0,m}$ in which l_m and l_x both intersect l_1 in the same number of positions. This gives us the six distinct sets:

$$\begin{aligned}
S_{1,2} &= \{2\} \\
S_{1,3} &= \{3\} \\
S_{1,4} &= \{4,5,7,9\} = S_{1,5} = S_{1,7} = S_{1,9} \\
S_{1,6} &= \{6,8,10,11,12\} = S_{1,8} = S_{1,10} = S_{1,11} = S_{1,12} \\
S_{1,13} &= \{13,14,15,16,17,18,19,20\} = S_{1,14} = S_{1,15} = \cdots = S_{1,20} \\
S_{1,21} &= \{21,22\} = S_{1,22}
\end{aligned}$$

Each set $S_{2,m}$, where $3 \leq m \leq 22$, consists of all the integers $x \in S_{1,m}$ in which l_m and l_x both intersect l_2 in the same number of positions. Of course, since in our left pattern P the left words l_1 and l_2 are equal, each set $S_{2,m}$ will be equal to the set $S_{1,m}$. This gives us the five distinct sets:

$$\begin{aligned}
S_{2,3} &= \{3\} \\
S_{2,4} &= \{4,5,7,9\} = S_{2,5} = S_{2,7} = S_{2,9} \\
S_{2,6} &= \{6,8,10,11,12\} = S_{2,8} = S_{2,10} = S_{2,11} = S_{2,12} \\
S_{2,13} &= \{13,14,15,16,17,18,19,20\} = S_{2,14} = S_{2,15} = \cdots = S_{2,20} \\
S_{2,21} &= \{21,22\} = S_{2,22}
\end{aligned}$$

Continuing on in the same manner, we find there are six distinct sets $S_{3,m}$:

$$\begin{array}{lllllllll}
S_{3,4} &=& \{4\} & & & & & & \\
S_{3,5} &=& \{5,7,9\} & = & S_{3,7} & = & S_{3,9} & & \\
S_{3,6} &=& \{6,8,10,11\} & = & S_{3,8} & = & S_{3,10} & = & S_{3,11} \\
S_{3,12} &=& \{12\} & & & & & & \\
S_{3,13} &=& \{13,14,15,16,17,18,19,20\} & = & S_{3,14} & = & S_{3,15} & = \cdots & = S_{3,20} \\
S_{3,21} &=& \{21,22\} & = & S_{3,22} & & & &
\end{array}$$

We will use the notation $n(i,m)$ to denote the smallest integer x in the set $S_{i,m}$, where $0 \leq i < m \leq 22$. In other words, $n(i,m)$ is the smallest integer x for which we can determine, before we begin our search, that the sets $R_{i,m}$ and $R_{i,x}$ will always be equal. Note that if $n(i,m) = x$ then we must have $n(i,x) = x$.

We will use the notation N_i to denote the set of all integers m in which $n(i,m) = m$, where $0 \leq i < m \leq 22$. For each A_i produced, our algorithm will only spend time and memory finding the sets $R_{i,m}$ in which $m \in N_i$. If $m \notin N_i$, then there is no need for our algorithm to spend time and memory finding $R_{i,m}$ since we know the set $R_{i,m}$ must be equal to the set $R_{i,n(i,m)}$ (which the algorithm will have already found).

For each matrix A_i produced by our algorithm, we can find the set $R_{i,m}$, where $m \in N_i$, by finding all the right words r in $R_{i-1,n(i-1,m)}$ (which is equal to the set $R_{i-1,m}$) in which $l_m r$ intersects row i of A_i in 4 positions. However, as the next example demonstrates, there may exist integers $m_1, m_2 \in N_i$ in which $n(i-1,m_1) = n(i-1,m_2)$, which implies R_{i,m_1} and R_{i,m_2} are both subsets of the same set (i.e. $R_{i-1,m_1} = R_{i-1,m_2} = R_{i,n(i-1,m_1)}$), and thus should be found at the same time. In other words, when producing the sets $R_{i,m}$, where $m \in N_i$, we only want our algorithm to make one pass through the right words in each set $R_{i-1,x}$, where $x \in N_{i-1}$.

Example 4.29. Let C be the code and P the left pattern in Example 4.27. Suppose our BIBD search algorithm for C and P has filled in the last $33 - n(W)$ columns of the first

3 rows of A, giving us the matrix A_3. We now want our algorithm to find the sets $R_{3,m}$, where $m \in N_3$, by only making one pass through the right words in each of the sets $R_{2,x}$, where $x \in N_2$.

Using the sets $S_{i,m}$ found in Example 4.28, we find the sets N_i, for $i = 0, 1, 2, 3$, are as follows:

$$\begin{aligned} N_0 &= \{1, 13, 21\} \\ N_1 &= \{2, 3, 4, 6, 13, 21\} \\ N_2 &= \{3, 4, 6, 13, 21\} \\ N_3 &= \{4, 5, 6, 12, 13, 21\} \end{aligned}$$

We will demonstrate how our algorithm finds the sets $R_{3,m}$, where $m \in N_3$, by describing how our algorithm finds the sets $R_{3,6}$ and $R_{3,12}$.

Since the left word l_6 intersects l_3 in 2 positions, the set $R_{3,6}$ will consist of all right words r in $R_{2,6}$ that intersect the right word r_3 (in row 3 of A_3) in $4 - 2 = 2$ positions. Similarly, since the left words l_{12} and l_3 intersect in 0 positions, the set $R_{3,12}$ will consist of all right words r in $R_{2,12}$ that intersect r_3 in $4 - 0 = 4$ positions. However, as the set $S_{2,6}$ in Example 4.28 demonstrates, $n(2, 12) = 6$, and thus the set $R_{2,12}$ is equal to the set $R_{2,6}$. Therefore, $R_{3,6}$ and $R_{3,12}$ are both subsets of $R_{2,6}$. Thus, we can find both of the sets $R_{3,6}$ and $R_{3,12}$ with one pass through the right words r in the set $R_{2,6}$. That is, for each $r \in R_{2,6}$, if we find that r intersects r_3 in 2 positions then we insert r into $R_{3,6}$, otherwise, if we find that r intersects r_3 in 4 positions then we insert r into $R_{3,12}$.

As Example 4.29 demonstrates, given the matrix A_i and the set $R_{i-1,x}$, where $x \in N_{i-1}$, if m is an element in N_i in which $n(i-1, m) = x$, then we can find the set $R_{i,m}$ by finding all the right words in $R_{i-1,x}$ that intersect the right word r_i in w positions, where r_i is the right word in row i of A_i and $4 - w$ is the number of positions in which the left words l_i and l_m intersect. We can produce all the sets $R_{i,m}$, where $m \in N_i$, that are subsets of the set $R_{i-1,x}$, by making one pass through the right words in $R_{i-1,x}$. That is, for each right word $r \in R_{i-1,x}$ we do the following: First, we find w, the number of

positions in which r and r_i intersect. If $0 \leq w \leq 4$, we then determine whether or not there exists an $m \in N_i$, where $n(i-1,m) = x$ (i.e. where $R_{i,m}$ is a subset of $R_{i-1,x}$), in which the left words l_i and l_m intersect in $4-w$ positions. If such an m exists we then insert r into the set $R_{i,m}$.

In order to determine for a given w, where $0 \leq w \leq 4$, whether or not there exists an integer m, where $m \in N_i$ and $n(i-1,m) = x$, in which the left words l_i and l_m intersect in $4-w$ positions, we will define the function $t(i-1,x,w)$, for $w = 0,1,2,3,4$. If there exists an $m \in N_i$, where $(n-1,m) = x$, in which l_i and l_m intersect in $4-w$ positions then $t(i-1,x,w)$ is assigned the value of m. If such an m does not exist, then $t(i-1,x,w)$ is set to -1.

Example 4.30. Let C be the code and P the left pattern in Example 4.27. As we saw in Example 4.29, for this C and P we have the following:

$$\begin{aligned} N_2 &= \{3,4,6,13,21\} \\ N_3 &= \{4,5,6,12,13,21\} \end{aligned}$$

Let us now find the value of the function $t(2,x,w)$ for $x \in N_2$ and $0 \leq w \leq 4$. Let us first consider the function $t(2,6,w)$. Since $6 \in N_3$, $n(2,6) = 6$, and l_6 intersects l_3 in $4-w = 2$ positions, the value of $t(2,6,2)$ is 6. Since $12 \in N_3$, $n(2,12) = 6$, and l_{12} intersects l_3 in $4-w = 0$ positions, the value of $t(2,6,4)$ is 12. Since 6 and 12 are the only integers $m \in N_3$ in which $n(2,m) = 6$, the value of $t(2,6,w)$ is -1 for $w = 0,1,3$. Thus:

$$\begin{aligned} t(2,6,0) &= -1 \\ t(2,6,1) &= -1 \\ t(2,6,2) &= 6 \\ t(2,6,3) &= -1 \\ t(2,6,4) &= 12 \end{aligned}$$

Similarly, $t(2,4,1) = 4$, $t(2,4,3) = 5$, $t(2,13,3) = 13$, and $t(2,21,4) = 21$. For all other $x \in N_2$ and $0 \leq w \leq 4$, we have $t(2,x,w) = -1$.

For a given matrix A_i, our BIBD search algorithm now produces the sets $R_{i,m}$, where $m \in N_i$, as follows: For each $x \in N_{i-1}$ and each right word $r \in R_{i-1,x}$, we first find w, the number of positions in which r intersects r_i, where r_i is the right word in the last $33 - n(W)$ columns of row i of A_i. If $0 \leq w \leq 4$ and $t(i-1, x, w) = m$, where $m \neq -1$, we then insert r into the set $R_{i,m}$.

We now have the logic that allows our algorithm to produce the sets $R_{i,m}$ without spending time and space reproducing the sets we know will always be equal to sets the algorithm has already produced. Let us now turn our attention to how we implement this operation.

We store each of the sets $R_{i,m}$, where $0 \leq i \leq 21$ and $m \in N_i$, in an array of right word indices $RIGHT\_LIST[0 \ldots num\_right\_lists - 1][0 \ldots max\_list\_size - 1]$, where num_right_lists is the total number of sets $R_{i,m}$ in which $m \in N_i$, and max_list_size is the maximum number of right words in a set $R_{i,m}$. The right word indices in each list $RIGHT\_LIST[i][\,]$ are stored in increasing order. We store the size of each set $R_{i,m}$ in an array $RIGHT\_LIST\_SIZE[0 \ldots num\_right\_lists - 1]$. That is, the number of right word indices in $RIGHT\_LIST[i][\,]$ is $RIGHT\_LIST\_SIZE[i]$.

In order to allow quick access to the set $R_{i,i+1}$ (which we use to fill in the last $33-n(W)$ columns of row $i+1$ of A_i), our algorithm will store $R_{i,i+1}$ in $RIGHT\_LIST[i][\,]$. The algorithm will store the remaining sets $R_{i,m}$, where $m \in N_i$ and $m \neq i+1$, in an arbitrarily chosen $RIGHT\_LIST[\,][\,]$ that is fixed before the search begins.

We implement the sets N_i and the function $t(i, m, w)$ with a set of 22 linked lists, which we will denote by $LINKED\_LIST[0 \ldots 21]$. The linked list $LINKED\_LIST[i]$ contains one node for each $m \in N_i$. The information contained in each node includes:

- *right_set_num*: This is the location in $RIGHT\_WORDS[\,]$ that $\mathcal{RIGHT}(W, C, l_m)$ is stored. (Note that this is only used to find the set $R_{0,m}$.)
- *list_num*: This is the location in $RIGHT\_LIST[\,][\,]$ in which the set $R_{i,m}$ will be

stored. That is, our algorithm will store the set $R_{i,m}$ in $RIGHT\_LIST[list\_num][\,]$. (Of course, as mentioned, if $m = i + 1$ then $list\_num = i$.)

- *num_equal_sets*: This is equal to $|S_{i,m}|$ (i.e. the number of sets $R_{i,x}$ that we know will always be equal to $R_{i,m}$).

- *child*$[0 \ldots 4]$: This is used to implement the function $t(i, m, w)$, where $0 \leq w \leq 4$. If $t(i, m, w) = z$, where $z \neq -1$, then *child*$[w]$ is the location in $RIGHT\_LIST[\,][\,]$ in which the set $R_{i+1,z}$ is stored. If $t(i, m, w) = -1$ then *child*$[w]$ is set to -1.

- *next*: This is a pointer to the next node in the linked list. If we are at the end of the list then *next* = *null*.

For each matrix A_i produced by our BIBD search algorithm, we use the linked list $LINKED\_LIST[i-1]$ to produce the sets $R_{i,m}$, where $m \in N_i$, as follows: For each node in $LINKED\_LIST[i-1]$, we run through each right word index, $index(r)$, in the list $RIGHT\_LIST[list\_num][\,]$. For each right word r (where $index(r)$ is in the list $RIGHT\_LIST[list\_num][\,]$), we find w, the number of positions in which r and r_i intersect. If $0 \leq w \leq 4$ and $child[w] \neq -1$ then we insert $index(r)$ into the list $RIGHT\_LIST[child[w]][\,]$.

One last aspect of our method for producing the sets $R_{i,m}$, where $m \in N_i$, that we must consider is that for each right word r, where $index(r)$ is in $RIGHT\_LIST[list\_num][\,]$, we must compute w, the number of positions in which r and r_i intersect. Due to the number of times our algorithm must make this calculation, we do not want our algorithm to compute w on the fly. Therefore, before we begin the search, we will build an array $INTERSECTION\_TABLE[0 \ldots num\_right\_words - 1][0 \ldots num\_right\_words - 1]$. The value of $INTERSECTION\_TABLE[i_1][i_2]$ is the number of positions in which the right words $RIGHT\_WORDS[i_1]$ and $RIGHT\_WORDS[i_2]$ intersect. Thus, our algorithm computes the number of positions in which the two right words r and r_i intersect by a simple table look up. Note that this table occupies a large amount of memory, but

the time saved by using it more than makes up for the space used.

We will now conclude our discussion of how our algorithm produces the sets $R_{i,m}$, where $0 \leq i \leq 21$ and $m \in N_i$, by giving a formal description of our algorithms for producing these sets. We will give two separate algorithms, one for producing the sets $R_{0,m}$, and one for producing the sets $R_{i,m}$, where $i \geq 1$. Both algorithms assume the linked lists *LINKED_LIST*[ ] and all other data structures used (such as *INTERSECTION_TABLE*[ ][ ]) have already been found and are available.

We begin with a formal description of our algorithm for producing the sets $R_{0,m}$, where $m \in N_0$:

Algorithm 4.31. *BuildFirstRightWordLists*:

begin
Set *curr* to point at the first node in the linked list *LINKED_LIST*[0].
while *curr* $\neq$ *null* **do**
for $i_0 = 0$ to *RIGHT_SET_SIZE*[*curr* $\rightarrow$ *right_set_num*] $- 1$ **do**
Set *RIGHT_LIST*[*curr* $\rightarrow$ *list_num*][i_0] = *RIGHT_SET_START*[*curr* $\rightarrow$ *right_set_num*] $+ i_0$.
end for
Set *RIGHT_LIST_SIZE*[*curr* $\rightarrow$ *list_num*] = *RIGHT_SET_SIZE*[*curr* $\rightarrow$ *right_set_num*].
Set *curr* = *curr* $\rightarrow$ *next*.
end while
end

We will now give a formal description of our algorithm for producing the sets $R_{i,m}$, where $m \in N_i$:

Algorithm 4.32. *BuildRightWordLists*$(index, i)$:

- Input: the integers $index$ and i, where $index$ is the right word index of the right word in row i of the matrix A_i currently being processed by our BIBD search algorithm.

begin
Set $curr$ to point at the first node in the linked list $LINKED\_LIST[i]$.
while $curr \neq null$ **do**
Set $RIGHT\_LIST\_SIZE[curr \rightarrow list\_num] = 0$.
Set $curr = curr \rightarrow next$.
end while
Set $curr$ to point at the first node in the linked list $LINKED\_LIST[i-1]$.
while $curr \neq null$ **do**
for $i_0 = 0$ to $RIGHT\_LIST\_SIZE[curr \rightarrow list\_num] - 1$ **do**
Let $index_0 = RIGHT\_LIST[curr \rightarrow list\_num][i_0]$.
Let $w = INTERSECTION\_TABLE[index][index_0]$.
if $w \leq 4$ **and** $curr \rightarrow child[w] \neq -1$ **then**
Let $list\_num = curr \rightarrow child[w]$.
Set $RIGHT\_LIST[list\_num][RIGHT\_LIST\_SIZE[list\_num]] = index_0$.
Set $RIGHT\_LIST\_SIZE[list\_num] = RIGHT\_LIST\_SIZE[list\_num] + 1$.
end if
end for
Set $curr = curr \rightarrow next$.
end while
end

This completes our discussion of how our algorithm produces the sets $R_{i,m}$, where $0 \leq i \leq 21$ and $m \in N_i$. We will now turn our attention to the methods we use to prune the BIBD search tree of our BIBD search algorithm.

4.6.4 Pruning the BIBD Search Tree

We have two different methods for pruning the BIBD search tree. One method uses the sets $R_{i,m}$ to determine that a matrix A_i cannot be completed to an incidence matrix. The other method uses the automorphism group of C to determine that a matrix A_i cannot be completed to an incidence matrix.

Pruning with the Sets $R_{i,m}$

Let us first consider our pruning method that uses the sets $R_{i,m}$. Simply put, this method works by determining whether or not each set $R_{i,m}$ contains enough right words to (potentially) complete A_i to an incidence matrix. How we determine whether or not each set $R_{i,m}$ contains enough right words is explained next.

Let A_i be any matrix produced at the i^{th} level of the recursion. Suppose for a given set $R_{i,m}$, where $m \in N_i$, we find that the number of right words in $R_{i,m}$ is less than $|S_{i,m}|$. Then we immediately know that A_i cannot be completed to an incidence matrix. The reason for this is that in order for the algorithm to complete A_i to an incidence matrix, it will eventually have to insert into each row x, where $x \in S_{i,m}$, one of the right words in $R_{i,m}$ (since $R_{i,x} = R_{i,m}$, and thus, $R_{x-1,x}$ is a subset of $R_{i,m}$). Furthermore, as we shall see next, each of the $|S_{i,m}|$ right words in $R_{i,m}$, that the algorithm must eventually insert into A_i, must all be distinct (for the left words blocks we use with our algorithm). Therefore, if $|R_{i,m}| < |S_{i,m}|$ then we know A_i cannot be completed to an incidence matrix.

Let us now see why the $|S_{i,m}|$ right words must be distinct. Suppose we have an incidence matrix A in which the right words r_{m_1} and r_{m_2} in the last $33 - n(W)$ columns of rows m_1 and m_2 of A, respectively, are equal. Consider the left words l_{m_1} and l_{m_2} in the first $n(W)$ columns of rows m_1 and m_2 of A, respectively. Since the rows of A have weight 12, the rows of A intersect in 4 positions, and the right words r_{m_1} and r_{m_2} are equal, A must contain at least 8 columns in which l_{m_1} has a value of 1 and l_{m_2} has a

value of 0. Similarly, A must also contain at least 8 columns in which l_{m_2} has a value of 1 and l_{m_1} has a value of 0. This implies the left words l_{m_1} and l_{m_2} contain at least 16 components, and thus, $n(W) \geq 16$. However, for the left word blocks we use, we have $n(W) < 16$. Therefore, A cannot contain two rows with equal right words. Thus, the $|S_{i,m}|$ right words must be distinct.

We will now give a formal description of our algorithm for determining whether or not there are enough right words in the sets $R_{i,m}$, where $m \in N_i$, to (potentially) complete the current matrix A_i to an incidence matrix. The algorithm is a Boolean function called *EnoughRightWordsLeft*. It returns a value of *true* if and only if $|R_{i,m}| \geq |S_{i,m}|$, for all $m \in N_i$. The algorithm uses the linked list *LINKED_LIST*$[i]$ to access each set $R_{i,m}$, where $m \in N_i$. Recall that the field in each node in *LINKED_LIST*$[i]$ that contains $|S_{i,m}|$ is *num_equal_sets*.

Algorithm 4.33. boolean *EnoughRightWordsLeft*(i):

- Input: the integer i, which is equal to the number of rows that have been completely filled in the matrix currently being processed by our BIBD search algorithm.
- Output: returns *true* if *RIGHT_LIST_SIZE*[*list_num*] >= *num_equal_sets* for each node in *LINKED_LIST*$[i]$, otherwise, a value of *false* is returned.

begin
Set *enough_left* = *true*.
Set *curr* to point at the first node in the linked list *LINKED_LIST*$[i]$.
while *curr* $\neq$ *null* **do**
if *RIGHT_LIST_SIZE*[*curr* $\rightarrow$ *list_num*] < *curr* $\rightarrow$ *num_equal_sets* **then**
Set *enough_left* = *false*.
Break from while loop.
end if
Set *curr* = *curr* $\rightarrow$ *next*.

end while

Return *enough_left.*

end

This completes our look at how we use the sets $R_{i,m}$ to prune the BIBD search tree.

Pruning with the Automorphism Group of C

Let us now consider our second method of pruning the BIBD search tree, in which we make use of the automorphism group of C. For this method, it is important that we understand the exact order in which we visit the nodes in our BIBD search tree. Therefore, before we begin our discussion of our second method of pruning, we will first describe the exact order in which the nodes in the (unpruned) BIBD search tree T are visited.

Let us first consider the order in which the children of a node are visited. Let A_{i-1} be a matrix in our tree T and let $R_{i-1,i}$ denote the set of right words that is associated with A_{i-1}. Recall that the children of A_{i-1} are the matrices A_i in which: (1) the first $i-1$ rows of A_i are equal to the first $i-1$ rows of A_{i-1}, and (2) the right word in the last $33-n(W)$ columns of A_i is an element of the set $R_{i-1,i}$. Let A_i and A_i' be any two children of A_{i-1}, and let r_i and r_i' be the right words in row i of A_i and A_i', respectively. Then our algorithm will visit the child A_i before the child A_i' if and only if, when considered as binary integers, r_i is greater than r_i'. In terms of the right word indices of r_i and r_i', our algorithm will visit A_i before A_i' if and only if $index(r_i) < index(r_i')$ (since the right words in each right set in *RIGHT_WORDS*[ ] are stored in descending order).

The order in which we visit the children of a node, together with the fact that our algorithm performs a preorder traversal of the tree T, gives us the order in which we visit all the nodes in T. That is, if A_{i_1} and A_{i_2} are any two matrices in T, where $i_1 \leq i_2$, then:

- if A_{i_2} is a descendent of A_{i_1} then A_{i_1} is visited first, otherwise

- there exists a smallest integer $i_0 \leq i_1$ in which row i_0 of A_{i_1} is not equal to row i_0 of A_{i_2}. If r and r' are the right words in row i_0 of A_{i_1} and A_{i_2}, respectively, then A_{i_1} is visited before A_{i_2} if and only if $index(r) < index(r')$.

Note that if the algorithm visits the matrix A_{i_1} before A_{i_2} (and A_{i_2} is not a descendent of A_{i_1}) then the algorithm will visit all descendents of A_{i_1} before it visits A_{i_2}.

We are now ready to begin our discussion of our second method of pruning the BIBD search tree T in which we make use of the automorphism group of C. This method of pruning T works as follows: Let A_{i_2} denote the matrix that is currently being processed by the algorithm. Using the automorphism group of C, our algorithm first determines if there exists a matrix A_{i_1} (that is neither an ancestor nor a descendent of A_{i_2} in T) with the property that A_{i_1} can be completed to an incidence matrix if and only if A_{i_2} can be completed to an incidence matrix. If such an A_{i_1} exists and has already been processed, our algorithm then prunes A_{i_2} from T (since the algorithm has already determined that A_{i_1} cannot be completed to an incidence matrix). The algorithm determines that it has already processed A_{i_1} by finding the first row i_0 in which A_{i_1} and A_{i_2} differ. If the right words in row i_0 of A_{i_1} and A_{i_2} are r and r', respectively, then as we've just seen, A_{i_1} has already been processed if and only if $index(r) < index(r')$.

We have two different approaches to using this method of pruning the BIBD search tree. In our first approach, we use the left automorphism group of W and C to determine *before* we begin our search that certain matrices produced by the algorithm can be pruned from the BIBD search tree. In our second approach, we use the right automorphism group of W and C to show the matrix currently being processed by the algorithm can be pruned from the BIBD search tree.

Pruning with the Left Automorphism Group

We will first discuss our approach that uses the left automorphism group of W and C. Recall that a left automorphism of W and C is an automorphism of C that does not

move the last $33 - n(W)$ coordinates of C. The left automorphism group of W and C, which we denote by $\mathcal{AUT}_{LEFT}(W, C)$, is the set of all left automorphisms of W and C.

For this discussion, we will use the notation $\pi_r^*(m, i)$, where $1 \leq m < i \leq 22$, to denote any permutation π_r of the rows of our left pattern P in which:

$$\pi_r = \begin{pmatrix} 1 & 2 & \cdots & m-1 & m & \cdots \\ 1 & 2 & \cdots & m-1 & i & \cdots \end{pmatrix}.$$

In other words, $\pi_r^*(m, i)$ will denote any permutation of the rows of P that moves row i of P to row m, but does not move the first $m - 1$ rows of P.

Our method of pruning that uses the left automorphism group of W and C is based on the following fact: Suppose for two integers i and m, where $1 \leq m < i \leq 22$, there exists a permutation $\pi_r^*(m, i)$ of the rows of P and a permutation π_c of the columns of P, where $\pi_c^{33} \in \mathcal{AUT}_{LEFT}(W, C)$, such that $\pi_r^*(m, i)\pi_c P = P$. Then, as we shall see next, our BIBD search algorithm can prune any matrix A_i, produced at the i^{th} level of the recursion, in which $index(r_i) < index(r_m)$, where r_i and r_m are the right words in the last $33 - n(W)$ columns of rows i and m of A_i, respectively.

Let us now see why this is so. Let A_i be any matrix produced at the i^{th} level of the recursion. Consider the matrix $A_i' = \pi_r^*(m, i)\pi_c^{33} A_i$. Since $\pi_r^*(m, i)\pi_c P = P$, we know the left pattern in the first $n(W)$ columns of A_i' is also P, and thus, A_i' is also an element of our BIBD search tree. Furthermore, since π_c^{33} is a left automorphism of W and C, and thus, π_c^{33} is an automorphism of C, we know A_i can be completed to an incidence matrix if and only if A_i' can be completed to an incidence matrix. Therefore, once our algorithm processes one of A_i and A_i', the other can be be pruned from our BIBD search tree. As we shall see next, if $index(r_i) < index(r_m)$, then our algorithm will process A_i' before A_i, and thus, our algorithm can prune the matrix A_i.

Now, suppose $index(r_i) < index(r_m)$. Then, since $\pi_r^*(m, i)$ does not move the first $m-1$ rows of A_i, and since π_c^{33} does not move the last $33-n(W)$ columns of A_i, we know the right words in the first $m - 1$ rows of A_i and A_i' are equal. Therefore, which one of

A_i and A'_i is processed first by the algorithm is determined by the right words in rows m of A_i and A'_i. The right word in row m of A_i is r_m. Since $\pi_r^*(m,i)$ moves the right word in row i of A_i to row m of A'_i, we know r_i is the right word in row m of A'_i. Therefore, since $index(r_i) < index(r_m)$, we know the algorithm will process A'_i before A_i. Thus, we can prune the matrix A_i.

Summarizing what we have found, if for two integers m and i, where $1 \leq m < i \leq 22$, there exist permutations π_r and π_c of the rows and columns of P, respectively, where π_r is a permutation of the form $\pi_r^*(m,i)$ and π_c^{33} is a left automorphism of W and C, such that $\pi_r\pi_c P = P$, then any matrix A_i in which $index(r_i) < index(r_m)$ can be pruned from our BIBD search tree.

Let us now consider how we implement this method of pruning in our BIBD search algorithm. We begin by finding, *before* we begin our search, the set S of all ordered pairs (m,i), where $1 \leq m < i \leq 22$, for which there exist permutations $\pi_r^*(m,i)$ and π_c, where $\pi_c^{33} \in \mathcal{AUT}_{LEFT}(W,C)$, in which $\pi_r^*(m,i)\pi_c P = P$. At the i^{th} level of the recursion, if there exists a pair $(m,i) \in S$, then our algorithm will skip over all of the right words r in $R_{i-1,i}$ in which $index(r) < index(r_m)$, where r_m is the right word in row m of the input matrix A_{i-1}. Thus, if there exists a pair $(m,i) \in S$, then the only matrices A_i produced at the i^{th} level of the recursion are those in which $index(r_i) > index(r_m)$.

Now, for a given integer i, there may exist more than one pair (m,i) in S, which means that for each $r \in R_{i,i-1}$, our algorithm may have to compare $index(r)$ to the right index of more than one right word r_m in the input matrix A_{i-1}. However, this is not a problem since, as we shall see next, our algorithm will in fact only need to compare $index(r)$ to the index of the right word r_m in which m is maximal and $(m,i) \in S$.

Let us now see why this is so. Let A_{i-1} be the input matrix to the i^{th} level of the recursion. Suppose the set S contains the pairs (m_1,i) and (m_2,i), where $m_1 < m_2$. Then for each right word $r \in R_{i,i-1}$, in order to determine whether or not it needs to insert r into row i of A_{i-1}, our algorithm must first check that $index(r) > index(r_{m_1})$ and

$index(r) > index(r_{m_2})$. However, if the set S also contains the pair (m_1, m_2), then the right words r_{m_1} and r_{m_2} in the input matrix A_{i-1} will have the property that $index(r_{m_1}) < index(r_{m_2})$. Therefore, if (m_1, m_2) is also in S, then our algorithm only needs to check that $index(r) > index(r_{m_2})$ (since if $index(r) > index(r_{m_2})$ we immediately know that $index(r) > index(r_{m_1})$). In other words, if the set S has the property that if for all $(m_1, i), (m_2, i) \in S$, where $m_1 < m_2$, we have $(m_1, m_2) \in S$, then for each $r \in R_{i-1,i}$, our algorithm will only need to check that $index(r) < index(r_m)$, where m is the largest integer in which $(m, i) \in S$.

We will now show that the set S does in fact have this transitive property. That is, we will show that if $(m_1, i), (m_2, i) \in S$, where $m_1 < m_2$, then $(m_1, m_2) \in S$. To accomplish this, we must show there exist permutations π_r and π_c, where π_r is a permutation of the form $\pi_r^*(m_1, m_2)$ (i.e. a permutation that does not move the first $m_1 - 1$ rows of P and moves row m_2 of P to row m_1) and π_c^{33} is a left automorphism of W and C, such that $\pi_r \pi_c P = P$.

Let us now find such permutations π_r and π_c. Since $(m_1, i) \in S$, we know there exist permutations π_{r_1} and π_{c_1}, where π_{r_1} is a permutation of the form $\pi_r^*(m_1, i)$ and $\pi_{c_1}^{33}$ is a left automorphism of W and C, such that $\pi_{r_1} \pi_{c_1} P = P$. Similarly, since $(m_2, i) \in S$, we know there exist appropriate permutations π_{r_2} and π_{c_2} such that $\pi_{r_2} \pi_{c_2} P = P$. Consider the permutations $\pi_{r_1} \pi_{r_2}^{-1}$ and $\pi_{c_1} \pi_{c_2}^{-1}$. We will show these two permutations are the sort of permutations we are looking for. Since $\pi_{c_1}^{33}$ and $\pi_{c_2}^{33}$ are both left automorphisms of W and C, we know $\pi_{c_1}^{33} (\pi_{c_2}^{33})^{-1} = \pi_{c_1}^{33} (\pi_{c_2}^{-1})^{33} = (\pi_{c_1} \pi_{c_2}^{-1})^{33}$ is also a left automorphism of W and C. Since π_{r_1} does not move the first $m_1 - 1$ rows of P, $\pi_{r_2}^{-1}$ does not move the first $m_2 - 1$ rows of P, and $m_1 < m_2$, we know $\pi_{r_1} \pi_{r_2}^{-1}$ does not move the first $m_1 - 1$ rows of P. Furthermore, since $\pi_{r_2}^{-1}$ moves row m_2 to row i and since π_{r_1} moves row i to row m_1, we know $\pi_{r_1} \pi_{r_2}^{-1}$ moves row m_2 to row m_1. Finally, since $\pi_{r_2} \pi_{c_2} P = P$, which implies

$\pi_{c_2}^{-1}\pi_{r_2}^{-1}P = P$, we have:

$$\begin{aligned}(\pi_{r_1}\pi_{r_2}^{-1})(\pi_{c_1}\pi_{c_2}^{-1})P &= (\pi_{r_1}\pi_{c_1})(\pi_{r_2}^{-1}\pi_{c_2}^{-1})P \\ &= \pi_{r_1}\pi_{c_1}P \\ &= P\end{aligned}$$

Therefore, the permutations $\pi_r = \pi_{r_1}\pi_{r_2}^{-1}$ and $\pi_c = \pi_{c_1}\pi_{c_2}^{-1}$ are just the sort of permutations we are looking for. Thus, if $(m_1, i), (m_2, i) \in S$, where $m_1 < m_2$, then $(m_1, m_2) \in S$.

We are now ready to give a precise description of how we implement our method of pruning the BIBD search tree, which uses the left automorphism group of W and C, in our BIBD search algorithm. Let S_{max} denote the set of all pairs $(m, i) \in S$ in which m is maximal. Before we begin our search, we find S_{max}. We store S_{max} in an integer array $MUST\_NOT\_COME\_BEFORE[1 \ldots 22]$. If there exists a pair $(m, i) \in S_{max}$ then $MUST\_NOT\_COME\_BEFORE[i]$ is assinged the value m. If no such pair exists then $MUST\_NOT\_COME\_BEFORE[i]$ is set to -1.

At the i^{th} level of the recursion, if $MUST\_NOT\_COME\_BEFORE[i] = m$, where $m \neq -1$, then our BIBD search algorithm will skip over all the right words in the set $R_{i-1,i}$ in which $index(r) < index(r_m)$, where r_m is the right word in row m of the input matrix A_{i-1}. Since the set $R_{i-1,i}$ is stored as an array $RIGHT\_LIST[i-1][\,]$ of increasing right word indices, the algorithm can skip over the appropriate right words in $R_{i-1,i}$ by simply sequentially scanning the array $RIGHT\_LIST[i-1][\,]$ until an integer i_0 is found in which $RIGHT\_LIST[i-1][i_0] > index(r_m)$. The algorithm can then continue by inserting into $BIBD\_RIGHT\_INDEX[i]$, one by one, all of the right word indices in $RIGHT\_LIST[i-1][\,]$ in positions i_0 and beyond.

This concludes our discussion on our first approach of pruning the BIBD search tree using the automorphism group of C, in which we make use of the left automorphism group of W and C. We will now consider our second approach in which we make use of the right automorphism group of W and C to prune the BIBD search tree.

Pruning with the Right Automorphism Group

Recall, the right automorphism group of W and C, which we denote by $\mathcal{AUT}_{RIGHT}(W, C)$, is the set of all permutations in $\mathcal{AUT}(C)$ that do not move the first $n(W)$ coordinates of C. The way in which we use the right automorphism group to prune our BIBD search tree T works by determining, for a given matrix A_i, whether or not there exists a permutation $\pi \in \mathcal{AUT}_{RIGHT}(W, C)$, that does not alter the first $i-1$ rows of A_i, such that $index(r_i') < index(r_i)$, where r_i and r_i' are the right words in row i of A_i and πA_i, respectively. If such a permutation π exists then, as we shall see, we can prune A_i from T.

Let A_{i-1} be our input matrix and let $R_{i-1,i}$ be the set of right words associated with it. Let r and r' be any two right words in $R_{i-1,i}$ in which $index(r) < index(r')$. Let A_i and A_i' denote the matrix A_{i-1} with the right words r and r' inserted into row i, respectively. Suppose there exists a right automorphism $\pi \in \mathcal{AUT}_{RIGHT}(W, C)$ that permutes the first i rows of A_i to the first i rows of A_i'. That is, suppose there exists a $\pi \in \mathcal{AUT}_{RIGHT}(W, C)$ that: (1) does not alter the first $i-1$ rows of A_i, and (2) permutes the right word r in the last $33 - n(W)$ columns of A_i to the right word r' in the last $33 - n(W)$ columns of A_i'. Then, since π is a right automorphism, and thus does not alter the left pattern P in A_i, we have $\pi A_i = A_i'$. Furthermore, since π is also an automorphism of C, we know A_i can be completed to an incidence matrix if and only if A_i' can be completed to an incidence matrix. Thus, since our algorithm will process the matrix A_i' first (since $index(r') < index(r)$), our algorithm does not have to produce the matrix A_i, and thus A_i is pruned from our BIBD search tree. This gives us our method for pruning the BIBD search tree using the right automorphism group of C.

Let us now consider how we can implement this method of pruning in our BIBD search algorithm. First, since we will only be working with right words and right automorphisms, we will rewrite the permutations in $\mathcal{AUT}_{RIGHT}(W, C)$ in terms of the coordinates our right words, giving us a group of permutations on the symbols $1, 2, \ldots, 33 - n(W)$ that

we will denote by $RIGHT\_SUBGROUP[0]$. That is, if:

$$\begin{pmatrix} 1 & 2 & \cdots & n(W) & n(W)+1 & n(W)+2 & \cdots & 33 \\ 1 & 2 & \cdots & n(W) & a_{n(W)+1} & a_{n(W)+2} & \cdots & a_{33} \end{pmatrix}$$

is a permutation in $\mathcal{AUT}_{RIGHT}(W, C)$ then

$$\begin{pmatrix} n(W)+1-n(W) & n(W)+2-n(W) & \cdots & 33-n(W) \\ a_{n(W)+1}-n(W) & a_{n(W)+2}-n(W) & \cdots & a_{33}-n(W) \end{pmatrix}$$

is the corresponding permutation in $RIGHT\_SUBGROUP[0]$. Since, for a given A_{i-1}, we are only interested in the permutations in $\mathcal{AUT}_{RIGHT}(W, C)$ that do not alter the first $i-1$ rows of A_{i-1}, and thus the permutations in $RIGHT\_SUBGROUP[0]$ that do not alter the right words $r_1, r_2, \ldots, r_{i-1}$, we will maintain a set of such permutations, which we will denote by $RIGHT\_SUBGROUP[i-1]$. That is, for a given A_{i-1}, where $1 \leq i-1 \leq 21$, the set $RIGHT\_SUBGROUP[i-1]$ will consist of all permutations in $RIGHT\_SUBGROUP[0]$ that do not alter the right words $r_0, r_1, \ldots, r_{i-1}$ in the first $i-1$ rows of A_{i-1}. Note that $RIGHT\_SUBGROUP[i-1]$ is a subgroup of $RIGHT\_SUBGROUP[0]$.

At the i^{th} level of the recursion, our algorithm uses $RIGHT\_SUBGROUP[i-1]$ to prune the BIBD search tree, and produce the group $RIGHT\_SUBGROUP[i]$, as follows: For each right word $r \in R_{i-1,i}$, before our algorithm inserts r into row i of A_{i-1}, it will first determine whether or not there exists a permutation π in $RIGHT\_SUBGROUP[i-1]$ in which $index(\pi r) < index(r)$. (Note that since the right automorphism of W and C, that corresponds to π, is an automorphism of C that does not alter the first $i-1$ rows of A_{i-1}, nor the left pattern P in first $n(W)$ columns of A_{i-1}, we know the right word πr must also be an element of the set $R_{i-1,i}$.) If such a permutation π exists then the algorithm throws away the right word r (since the algorithm has already tried inserting πr into row i of A_{i-1} and did not find an incidence matrix) and then continues on with the next right word in $R_{i-1,i}$. If such a permutation π does not exist then the algorithm proceeds by inserting r into row i of A_{i-1}, producing the matrix A_i. The algorithm then produces $RIGHT\_SUBGROUP[i]$ by finding all permutations π in $RIGHT\_SUBGROUP[i-1]$ in which $\pi r = r$.

We will now conclude our discussion of how we use the right automorphism group to prune our BIBD search tree by giving formal descriptions of our algorithm for pruning using the group $RIGHT\_SUBGROUP[i-1]$ and our algorithm for producing the group $RIGHT\_SUBGROUP[i]$.

First we will give a formal description of our algorithm for pruning the BIBD search tree with the group $RIGHT\_SUBGROUP[i-1]$. The algorithm is a Boolean function called *NonIsomorphicRightWord*. The algorithm returns a value of *false* if there exists a permutation π in $RIGHT\_SUBGROUP[i-1]$ in which $index(\pi r) < index(r)$, where r is the right word our BIBD search algorithm is preparing to insert into row i of the current input matrix A_{i-1}. Note that the algorithm uses the fact that $index(\pi r) < index(r)$ if and only if, when considered as binary integers, πr is greater than r, to determine whether or not $index(\pi r) < index(r)$.

Algorithm 4.34. **boolean** *NonIsomorphicRightWord*$(index, i)$:

- Input: the right *index* of the right word our BIBD search algorithm is preparing to insert into row i of the current input matrix A_{i-1}.
- Output: returns *false* if there exists a permutation π in $RIGHT\_SUBGROUP[i-1]$ in which $index(\pi r) < index(r)$, where $r = RIGHT\_WORDS[index]$, otherwise a value of *true* is returned.

begin
Set $non\_isomorphic = true$.
for each π in $RIGHT\_SUBGROUP[i-1]$ **do**
Let $r' = \pi r$, where $r = RIGHT\_WORDS[index]$.
if the binary value of r' is greater than the binary value of r **then**
Set $non\_isomorphic = false$.
Break from for loop.
end if

end for

Return *non_isomorphic*.

end

Next, we give our algorithm for producing the group $RIGHT\_SUBGROUP[i]$. The algorithm is called *BuildRightSubgroup* and uses $RIGHT\_SUBGROUP[i-1]$ and the right word r in row i of the current matrix A_i to produce the set $RIGHT\_SUBGROUP[i]$, where $1 \leq i \leq 21$.

Algorithm 4.35. *BuildRightSubgroup*$(index, i)$:

- Input: The right *index* of the right word in row i of the current matrix A_i being processed by our BIBD search algorithm.
- Output: The right subgroup $RIGHT\_SUBGROUP[i]$ for the matrix A_i.

begin

Initialize $RIGHT\_SUBGROUP[i]$ to the trivial group.

for each π in $RIGHT\_SUBGROUP[i-1]$ **do**

Let $r' = \pi r$, where $r = RIGHT\_WORDS[index]$.

if $r' = r$ **then**

Insert π into $RIGHT\_SUBGROUP[i]$.

end if

end for

end

This concludes our discussion of the different methods we use to prune the BIBD search tree our BIBD search algorithm.

4.6.5 A Formal Description of our BIBD Search Algorithm

We will now conclude this section by giving a formal description of our BIBD search algorithm for a given left word block W, a given code $C \in \mathcal{D}(W)$, and a given left pattern P in $L(W, C)$.

We will begin by giving the basic recursion step of our algorithm for determining for a given W, C, and P, whether or not C contains an incidence matrix A of a $(22, 33, 12, 8, 4)$-BIBD, that begins with the left pattern P. The algorithm starts the recursion with the call *FillNextBIBDRow*(1). If the call to *FillNextBIBDRow*(1) returns then we know the code C does not contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD that begins with the left pattern P.

Algorithm 4.36. *FillNextBIBDRow*(i):

- Input: the integer i, which is the next position to be filled in the (global) array *BIBD_RIGHT_INDEX*[ ]. (Recall that *BIBD_RIGHT_INDEX*[ ] consists of the right word indices of the right words in A. We do not physically insert right words into the last $33-n(W)$ columns of A, unless we complete it to an incidence matrix.)

begin
if $i = 22$ **then**
if the last row of A can be filled appropriately **then**
We have found an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD.
Print A and stop program.
end if
end if
else
Set *first_index* $= 0$.
if *MUST_NOT_COME_BEFORE*$[i] \neq -1$ **then**

```
        while first_index < RIGHT_LIST_SIZE[i − 1] and
         RIGHT_LIST[i − 1][first_index] < BIBD_RIGHT_INDEX[
         MUST_NOT_COME_BEFORE[i]] do
          Set first_index = first_index + 1.
        end while
      end if
      for index = first_index to RIGHT_LIST_SIZE[i − 1] − 1 do
        if NonIsomorphicRightWord(index, i) then
          Set BIBD_RIGHT_INDEX[i] = index.
          Call BuildRightWordLists(index, i).
          if EnoughRightWordsLeft(i) then
            Call BuildRightSubgroup(index, i).
            Call FillNextBIBDRow(i + 1).
          end if
        end if
      end for
    end else
  end
```

Next, we give the function *BIBDSearch* which is used to start the search for an incidence matrix in C that begins with the left pattern P. Before beginning the search, the function builds all of the data structures used in the search that are dependent on the code C and left pattern P (such as the linked lists *LINKED_LIST*[ ] and the array *MUST_NOT_COME_BEFORE*[ ]). The function assumes all other data structures used in the search (such as the array *INTERSECTION_TABLE*[ ][ ] and the group *RIGHT_SUBGROUP*[0]) have been found and are available.

Algorithm 4.37. *BIBDSearch*(W, C, P):

- Input: a left word block W, a code $C \in \mathcal{D}(W)$, and a left pattern $P \in L(W, C)$.

begin
  Create the linked lists *LINKED_LIST*$[i]$, for $i = 0, 1, \ldots, 21$.
  Call *BuildFirstRightWordLists*.
  if *EnoughRightWordsLeft*(0) **then**
    Set the array *MUST_NOT_COME_BEFORE*[ ].
    Call *FillNextBIBDRow*(1).
  end if
  Dispose of the linked lists *LINKED_LIST*$[i]$, for $i = 0, 1, \ldots, 21$.
end

If the call to the function *FillNextBIBDRow*(1) returns, then we know the code C does not contain an incidence matrix A for a $(22, 33, 12, 8, 4)$-BIBD that begins with the left pattern P. Note that the reason we call the function *EnoughRightWordsLeft* is that for one of the left word blocks W we use (specifically the d_4-block) there exist codes in $\mathcal{D}(W)$ that do not contain many of the left words in $\mathcal{LEFT}(W)$, and thus may not contain the left pattern P. We can detect this situation by calling *EnoughRightWordsLeft* to make sure none of the sets $R_{0,m}$ are empty.

4.7 Sorting the Left Pattern Rows

The running time of our BIBD search algorithm is greatly affected by the way in which the rows of the left pattern P in the matrix A are ordered. Therefore, before we begin our BIBD search algorithm for a particular code C and left pattern P, we sort the rows of P. In this section, we will discuss how we sort the rows of our left patterns.

Ideally, we would like to sort the rows of our left patterns P in such a way that our BIBD search algorithm performs the least amount of work. One way we may attempt to do this is to sort the rows of P "on the go". For example, at the start of the i^{th} level of the recursion, we could pick as the i^{th} row of P, the left word in the last $22 - i$ rows of P in which the set $R_{i-1,m}$ is the smallest. However, the linked lists $LINKED\_LIST[\,]$ used by our algorithm to generate the sets $R_{i,m}$, where $m \in N_i$, cannot be built without prior knowledge of the order of the rows in P. Therefore, if we sort the rows of P on the go then our algorithm will either have to build the linked lists $LINKED\_LIST[\,]$ on the go or independently produce every one of the sets $R_{i,m}$, for $m = i+1, i+2, \ldots, 22$. In either case, additional operations are required for each recursive call made by the algorithm. Due to the large number of recursive calls made by the algorithm, these additional operations may mean this "on the go" method for sorting the rows of P may not be the best method. Furthermore, this method is a greedy algorithm, and thus may not be optimal. Therefore, we have chosen to sort the rows of our left patterns before we begin the search.

We sort the rows of our left patterns P by first defining a canonical form for the rows of our left patterns. We then sort the rows of P, producing a left pattern whose rows are in canonical form. The canonical form we have chosen for the rows of our left patterns is the one we hope will result in our BIBD search algorithm performing as few operations as possible. We have considered many factors in determining the canonical form of our left patterns P such as: the sizes of the right sets of the left words in P, which left words in P have equal right sets, and the number of positions in which the left words in P intersect. We have in fact tested several different canonical forms. Based on this testing, we have selected the canonical form that seems to work best, in general. In the remainder of this section, we will describe the canonical form we have chosen and briefly discuss how we sort the rows of any left pattern P to a left pattern whose rows are in our canonical form.

We will begin by discussing how we deal with the left words in P whose right sets are not equal. First, we group together the left words in P whose right sets are equal.

This gives us a left pattern P whose rows can be partitioned into consecutive blocks of rows $B_1, B_2, \ldots, B_n$, where two rows are in the same block if and only if the left words in the rows have equal right sets. Next, we sort the blocks $B_1, B_2, \ldots, B_n$ using several heuristics. For any two blocks of rows, B_i and B_j, the heuristics we use to determine which block occurs first are, in decreasing order of relevance, as follows:

1. The block whose left words have greater weight occurs first.
2. If the left words in both blocks have the same weight then the block whose left words have a smaller right set occurs first.
3. If the sizes of the right sets are the same for the left words in both blocks then the block that contains the most left words occurs first.
4. If both blocks contain the same number of left words then we arbitrarily chose the block whose right set occurs first in the array $RIGHT\_WORDS[\,]$.

These heuristics uniquely define the order in which we sort the blocks in P. We will refer to any left pattern whose blocks obey the above heuristics as a left pattern whose *blocks* are in canonical form.

We will now define our canonical form for the rows of our left patterns. Let $\mathcal{S}(P)$ denote the set of all left patterns P' in which the rows of P' are a permutation of the rows of P and the blocks of P' are in canonical form. For a given $P' \in \mathcal{S}(P)$, let $f(P', i_0, i_1)$ denote the number of positions in which the left words in rows i_0 and i_1 of P intersect. The unique left pattern $P' \in \mathcal{S}(P)$, that we will select as our left pattern whose rows are in our canonical form, will have the property that for each $P'' \in \mathcal{S}(P)$:

- if there exist integers i_0 and i_1 such that $f(P', i_0, i_1) \neq f(P'', i_0, i_1)$, then if m_1 is the smallest integer for which there exists an integer m, where $m < m_1$, such that $f(P', m, m_1) \neq f(P'', m, m_1)$, and if m_0 is the smallest integer for which $f(P', m_0, m_1) \neq f(P'', m_0, m_1)$, then $f(P', m_0, m_1)$ is "theoretically better" than

$f(P'', m_0, m_1)$. What we mean by "theoretically better" is as follows: Let $X(w)$ denote the total number of pairs of binary vectors of length $33 - n(W)$ that intersect in $4 - w$ positions, where $0 \leq w \leq 4$. Let $w' = f(P', m_0, m_1)$ and let $w'' = f(P'', m_0, m_1)$. Then we consider w' to be theoretically better than w'' if $X(w') < X(w'')$.

If we find $\mathcal{S}(P)$ contains more than one left pattern P' with the above property, then we simply select, as our unique left pattern P' whose rows are in our canonical form, the "smallest" left pattern in $\mathcal{S}(P)$ with the above property. What we mean by "smallest" is the left pattern P' that is "less than" all such left patterns P''. (What we mean by P' being less than P'' is defined in Section 1.1.6.)

For a given left pattern P, we find the unique left pattern $P' \in \mathcal{S}(P)$, whose rows are in our canonical form, by creating a tree T of partial left patterns. Level i of T consists of all the different possibilities for the first i rows of a left pattern in $\mathcal{S}(P)$. The children of each partial left pattern P_i at level i of T are all the different possibilities for the $(i+1)^{th}$ row of a left pattern in $\mathcal{S}(P)$ whose first i rows are equal to P_i. We find our canonical P' by performing a preorder traversal, with pruning, of T.

4.8 Putting It All Together

In this section, we will piece together each of the different stages of our overall BIBD search algorithm, discussed in the previous five sections, to give us our algorithm for determining whether or not a $(22, 33, 12, 8, 4)$-BIBD exists.

First, for each left word block W depicted in Figure 4.1, we run our algorithm *EnumerateLeftPatterns* (given in Section 4.3) on W. This gives us a complete list $L(W)$ of inequivalent left patterns of W, for each of our left word blocks W.

Next, we run our algorithm *ProduceSetsOfCodesToSearch* (given in Section 4.4) on our complete list L_C of 478 inequivalent $(33, 16)$ doubly-even self-orthogonal codes that

we have not eliminated theoretically. This gives us our sets of codes $\mathcal{D}(W)$, for each of our left word blocks W. If for each code C in each set $\mathcal{D}(W)$, we find C does not contain an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD, then we know such a design does not exist.

For a given left word block W and code $C \in \mathcal{D}(W)$, we determine whether or not C contains an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD as follows: First, we run our algorithm *FindLeftPatternsForCode* (given in Section 4.5) on our code C and list $L(W)$. This gives us a list $L(W, C)$ of left patterns with the property that C contains an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD if and only if C contains an incidence matrix that begins with one of the left patterns in $L(W, C)$. For each left pattern $P \in L(W, C)$, we then sort the rows of P (as described in Section 4.7), giving us a left pattern whose rows are in our canonical form. We then run our algorithm *BIBDSearch* (given in Section 4.6) on our code C and canonical left pattern P. If C does contain an incidence matrix that begins with P then *BIBDSearch* will find such an incidence matrix. If for all left patterns $P \in L(W, C)$, *BIBDSearch* does *not* find an incidence matrix that begins with P, then we know C does not contain an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD.

Unfortunately, at the time of writing, our algorithm has proven too slow to search every code in the sets $\mathcal{D}(W)$ for the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. Therefore, in Section 4.10, we will discuss which codes we have searched (none of which contained an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD), and in Section 4.11, we will discuss some ideas we have for improving our algorithm, in order to search the remaining codes. But first, we will give an example of the performance of our algorithm for a particular code.

4.9 An Example of the Program's Performance

In this section, we will demonstrate the performance of our algorithm by giving a detailed description of the amount of work performed by the algorithm for a particular code C and two left patterns in $L(W, C)$, where W is the left word block in C used by our algorithm

in its search. By giving these examples, we hope to demonstrate several attributes of our algorithm such as the size of the search space for a given code C and left pattern P, the amount of work that is avoided with our pruning, how the performance of the algorithm can vary greatly over the different left patterns in $L(W, C)$, and just how close the algorithm comes to finding an incidence matrix in C.

In Appendix C, we give tables that summarize the results of our algorithm for each of the codes we have searched thus far. As the tables demonstrate, the overall running time of our search algorithm varies greatly over the different left word blocks W and the different codes C in each set $\mathcal{D}(W)$. Therefore, the characteristics of both the left word block W and the code C greatly affect the performance of our algorithm. Thus, the example we give in this section does not generally reflect the performance of our algorithm for all codes. However, we do hope our example will give a general feeling of how our algorithm will run for any particular code C and left word block W and why there is such a great variation in the running time of the algorithm. Towards these goals, once we have finished our example, we will conclude this section by discussing the different characteristics of the codes and left word blocks that affect the relative running time of our algorithm.

Since the performance of our algorithm is dependent on the code C and left word block W, we will begin our example by describing the characteristics of the code C and left word block W we have chosen for our example.

The code C we have chosen is code number 345 (i.e. the code with id 345 in the tables in Appendix C). This code is a d_3-code. The number of weight 4 words in C is 11 and the number of weight 12 words in C is 10347. The number of weight 12 words in C that do not intersect any of the weight 4 words in C in 4 positions is 7380. The order of the automorphism group of C is 294912.

The left word block W in $C^{\perp}$ that our algorithm uses in its search for an incidence matrix in C is a d_2^*-block. The size of the support of W is $n(W) = 8$. The number of left

i	$\lvert\mathcal{RIGHT}(W,C,i)\rvert$	$\{l \in \mathcal{LEFT}(W) \mid \mathcal{RIGHT}(W,C,l) = \mathcal{RIGHT}(W,C,i)\}$
1	104	{10100111, 10011011, 01101011, 01010111}
2	220	{10101010, 10010110, 01100110, 01011010}
3	220	{10101001, 10010101, 01100101, 01011001}
4	389	{10100100, 10011000, 01101000, 01010100}
5	388	{11000010, 00110010, 00001110}
6	388	{11000001, 00110001, 00001101}
7	584	{00000011}
8	736	{00000000}

Figure 4.6: Characteristics of the 8 distinct right sets $\mathcal{RIGHT}(W,C,i)$ for our code C and left word block W.

words in $\mathcal{LEFT}(W)$ is 24. The order of $\mathcal{AUT}(W)$, the automorphism group of W, is 48. The number of left patterns in our complete list $L(W)$ of inequivalent left patterns of W is 8.

The number of distinct right sets $\mathcal{RIGHT}(W,C,l)$, where $l \in \mathcal{LEFT}(W)$, is 8. For our example, we will denote these sets by $\mathcal{RIGHT}(W,C,i)$, for $i = 1, 2, \ldots, 8$. In Figure 4.6, we give the size of each set $\mathcal{RIGHT}(W,C,i)$ and the left words $l \in LEFT(W)$ in which $\mathcal{RIGHT}(W,C,l) = \mathcal{RIGHT}(W,C,i)$. For each left word l we will use the phrase "the right set number of l" to refer to the integer i in which $\mathcal{RIGHT}(W,C,i) = \mathcal{RIGHT}(W,C,l)$.

The order of $\mathcal{AUT}_{LEFT}(W,C)$ is 24. The order of $\mathcal{AUT}_{RIGHT}(W,C)$ is 6144. The order of $\mathcal{AUT}_{PART\_LEFT}(W,C)$ is 48. Since $|\mathcal{AUT}(W)|$ is 48, this tells us the set $\mathcal{COSET}(W,C)$ of right coset leaders contains $48/48 = 1$ element. This implies that for each of the 8 left patterns P in $L(W)$, there is exactly one left pattern in $L(W,C)$ that is equivalent to P. Thus, the total number of left patterns in $L(W,C)$ is 8.

$$P_1 = \left[\begin{array}{cccccccc|c}
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 2 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 2 \\
1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 2 \\
1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 2 \\
0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 2 \\
0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 2 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 6 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 6 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 6 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 6 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 6 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 6 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 4 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 4 \\
0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 4 \\
0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 4 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 4 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 4 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 7 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 7 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8
\end{array}\right]$$

$$P_2 = \left[\begin{array}{cccccccc|c}
1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 2 \\
1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 2 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 2 \\
1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 5 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 5 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 5 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 6 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 6 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 6 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 6 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 6 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 6 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 4 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 4 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 4 \\
0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 4 \\
0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 4 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 4 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 7 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8
\end{array}\right]$$

Figure 4.7: The left patterns P_1 and P_2. To the right of both left patterns are the right set numbers of the left words in each row.

The two left patterns in $L(W, C)$ with which we will demonstrate our program are the left patterns P_1 and P_2 given in Figure 4.7. For both left patterns, we also give the right set numbers of the left words in the left pattern.

In order to demonstrate the performance of our algorithm for searching C for an incidence matrix that begins with our left patterns, we will now rewrite the basic recursive step of our BIBD search algorithm for C and P (i.e. Algorithm 4.36) with several counts added:

Algorithm 4.38. *FillNextBIBDRow*(i):

begin
if $i = 22$ **then**
if the last row of A can be filled appropriately **then**
We have found an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD.
Print A and stop program.
end if
end if
else
Set $count1[i] = count1[i] + 1$.
Set $count2[i] = count2[i] + RIGHT\_LIST\_SIZE[i-1]$.
Set $first\_index = 0$.
if $MUST\_NOT\_COME\_BEFORE[i] \neq -1$ **then**
while $first\_index < RIGHT\_LIST\_SIZE[i-1]$ **and**
$RIGHT\_LIST[i-1][first\_index] < BIBD\_RIGHT\_INDEX[$
$MUST\_NOT\_COME\_BEFORE[i]]$ **do**
Set $first\_index = first\_index + 1$.
end while
end if
for $index = first\_index$ to $RIGHT\_LIST\_SIZE[i-1] - 1$ **do**

```
      Set count3[i] = count3[i] + 1.
      if NonIsomorphicRightWord(index, i) then
        Set count4[i] = count4[i] + 1.
        Set BIBD_RIGHT_INDEX[i] = index.
        Call BuildRightWordLists(index, i).
        if EnoughRightWordsLeft(i) then
          Set count5[i] = count5[i] + 1.
          Call BuildRightSubgroup(index, i).
          Call FillNextBIBDRow(i + 1).
        end if
      end if
    end for
  end else
end
```

For a given code C, left word block W, and left pattern P, our counts provide us with the following information:

- *count1*$[i]$: this is the number of times the recursive step of our algorithm was called at level i of the recursion. In other words, *count1*$[i]$ is equal to the number of matrices A_{i-1} produced by the algorithm for which the algorithm has yet to determine whether or not A_{i-1} can be completed to an incidence matrix.

- *count2*$[i]$: this is the total number of right words in the sets $R_{i-1,i}$ that are associated with each of the *count1*$[i]$ matrices A_{i-1} that are input into level i of the recursion. The average number of right words in the set $R_{i-1,i}$, for each A_{i-1} produced by the algorithm, is equal to *count2*$[i]$/*count1*$[i]$.

- *count3*$[i]$: this is equal to the number of times the recursive step was *not* aborted due to the pruning we do using the left automorphism group of W and C (i.e. the

pruning due to the array *MUST_NOT_COME_BEFORE*[ ]).

- *count4*[i]: this is equal to the number of times the recursive step was *not* aborted due to the pruning we do using the right automorphism group of W and C (i.e. the pruning due to the subgroups *RIGHT_SUBGROUP*[$i-1$]). In other words, *count4*[i] is equal to the total number of right words that the algorithm inserts into row i of each A_{i-1} produced, and thus is equal to the total number of matrices A_i produced by the algorithm.

- *count5*[i]: this is equal to the number of times the recursive step was *not* aborted due to the pruning we do using the sets $R_{i,i+1}, R_{i,i+2}, \ldots, R_{i,22}$. In other words, *count5*[i] is equal to the total number of matrices A_i produced by the algorithm for which the algorithm has yet to determine whether or not A_i can be completed to an incidence matrix. Of course, *count5*[i] is equal to *count1*[$i+1$].

In Figures 4.8 and 4.9, we give the counts produced by our program in its search for an incidence matrix in C that begins with the left patterns P_1 and P_2, respectively.

In both figures, the value of *count3*[i] $-$ *count2*[i] is equal to the number of right words in $R_{i-1,i}$ that our algorithm pruned using the array *MUST_NOT_COME_BEFORE*[ ]. The contents of this array for the left pattern P_1 is:

i	1	2	3	4	5	6	7	8	9	10	11
MUST_NOT_COME_BEFORE[i]	−1	1	1	3	3	5	−1	7	−1	9	−1
i	12	13	14	15	16	17	18	19	20	21	22
MUST_NOT_COME_BEFORE[i]	11	−1	13	−1	15	−1	17	−1	19	−1	21

The contents of the array for the left pattern P_2 is:

i	1	2	3	4	5	6	7	8	9	10	11
MUST_NOT_COME_BEFORE[i]	−1	−1	2	−1	−1	−1	−1	−1	8	−1	10
i	12	13	14	15	16	17	18	19	20	21	22
MUST_NOT_COME_BEFORE[i]	−1	12	−1	−1	15	−1	17	−1	−1	−1	21

i	$count1[i]$	$count2[i]$	$count3[i]$	$count4[i]$	$count5[i]$
1	1	220	220	16	13
2	13	113	63	18	18
3	18	1637	1138	226	108
4	108	287	94	47	22
5	22	558	348	134	18
6	18	24	7	7	4
7	4	268	268	268	148
8	148	1448	568	568	472
9	472	10464	10464	10464	4528
10	4528	12728	4996	4996	3208
11	3208	27504	27504	27504	10352
12	10352	14272	5856	5856	2016
13	2016	6624	6624	6624	3168
14	3168	3168	1584	1584	1584
15	1584	3168	3168	3168	3168
16	3168	3168	1584	1584	1584
17	1584	5760	5760	5760	2304
18	2304	2304	1152	1152	1152
19	1152	2304	2304	2304	0

Figure 4.8: Results of our search algorithm for code 345 and the left pattern P_1

i	$count1[i]$	$count2[i]$	$count3[i]$	$count4[i]$	$count5[i]$
1	1	104	104	12	12
2	12	1080	1080	128	128
3	128	6812	3122	654	630
4	630	38918	38918	12843	12387
5	12387	418109	418109	170545	154032
6	154032	2777414	2777414	1879136	1562985
7	1562985	13412286	13412286	12091115	8078913
8	8078913	88216052	88216052	86138225	29107072
9	29107072	43145114	21805112	21683304	13885284
10	13885284	69830556	69830556	69675783	7239478
11	7239478	8694697	4382970	4380752	949256
12	949256	2753444	2753444	2752948	166776
13	166776	184144	93600	93600	40224
14	40224	53568	53568	53568	11232
15	11232	29376	29376	29376	0

Figure 4.9: Results of our search algorithm for code 345 and the left pattern P_2

Of course, if the value of $MUST\_NOT\_COME\_BEFORE[i]$ is -1 then, as Figures 4.8 and 4.9 demonstrate, the algorithm will not perform any such pruning at level i, which explains why, in these cases, we have $count3[i] = count2[i]$.

The value of $count4[i] - count3[i]$ is equal to the number of right words that our algorithm pruned using the subgroup $RIGHT\_SUBGROUP[i-1]$. Of course, since these subgroups eventually become the trivial group (the further we go into the recursion), the majority of the pruning our algorithm performed using this method occurred at the early levels of the recursion.

The value of $count5[i] - count4[i]$ is equal to the number of right words that our algorithm pruned using the sets $R_{i,i+1}, R_{i,i+2}, \ldots, R_{i,22}$. Of course, since these subsets shrink in size the further we go into the recursion, the majority of the pruning our algorithm performed using this method occurred at the later levels of the recursion.

The amount of pruning our algorithm performed was relatively the same for both its search for an incidence matrix in C that begins with the left pattern P_1 and its search for an incidence matrix that begins with the left pattern P_2. However, the search space for the search involving the left pattern P_1 was much smaller than search space for P_2. As a result, the time our algorithm took to search C for an incidence matrix that begins with P_1 was only 34.5 seconds, while the time our algorithm took to search for an incidence matrix that begins with P_2 was 2.2 hours. On the other hand, even though the algorithm took a lot more time in its search for an incidence matrix that begins with P_2, the algorithm reached a deeper level in its search for an incidence matrix that begins with P_1.

The overall time it took our algorithm to search for an incidence matrix in C that begins with P, for each of the 8 left patterns P in $L(W, C)$, was 25.8 hours. The algorithm did not find an incidence matrix in C. The minimum and the maximum deepest level reached by the algorithm, over all left patterns in $L(W, C)$, was 14 and 20, respectively.

This concludes our example of the performance of our algorithm for code 345.

Since our example is not a general reflection of how our algorithm performs for all codes, we will now conclude this section by discussing the main factors that affect the running time of our algorithm.

Of course, the running time of our algorithm is determined principally by the size of the search space for the code C. There are many factors that affect the size of the search space such as the number of left patterns in $L(W, C)$, the sizes of the sets $\mathcal{RIGHT}(W, C, l)$, and the amount of pruning that is done. The number of left patterns in $L(W, C)$ is a factor since we must run our algorithm on every left pattern in $L(W, C)$. The sizes of the sets $\mathcal{RIGHT}(W, C, l)$ are a factor since they consist of all the right words that (potentially) may be inserted into row m of our incidence matrix A. The amount of pruning is a factor since the more pruning we do the smaller the search space will be.

The number of left patterns in $L(W, C)$, the sizes of the sets $\mathcal{RIGHT}(W, C, l)$, and the amount of pruning that is done are all basically determined by the left word block W and the code C. As we saw in Section 4.5, the number of left patterns in $L(W, C)$ is determined by the number of inequivalent left patterns of W and by the partial left automorphism group of C. The sizes of the sets $\mathcal{RIGHT}(W, C, l)$ are largely determined by the number of left words of W and by the number of weight 4 words in C. The number of left words of W affects the size of each set $\mathcal{RIGHT}(W, C, l)$ since, generally speaking, the more left words we have, the more right sets we have, and thus, the smaller each set will be. The number of weight 4 words in C affects the size of each set $\mathcal{RIGHT}(W, C, l)$ since, as Appendix A demonstrates (in which we give the weight distributions of our codes) the more weight 4 words we have, the fewer weight 12 words we have. Furthermore, generally speaking, the larger the number of weight 4 words, the larger the number of weight 12 words that intersect a weight 4 word in 4 positions, and thus, the smaller the sets $\mathcal{RIGHT}(W, C, l)$ (since there is a one-to-one correspondence between the right words in the sets $\mathcal{RIGHT}(W, C, l)$ and the weight 12 words in C that do not intersect any of the weight 4 words in C in 4 positions). The amount of pruning our algorithm performs is largely determined by the left word block W and the automorphism group of C. The left

word block W can affect the pruning since the number of distinct sets $\mathcal{RIGHT}(W, C, l)$ we have is largely determined by W. The number of distinct sets $\mathcal{RIGHT}(W, C, l)$ can have an affect on the amount of pruning we do using the sets $R_{i,m}$. The automorphism group of C affects the amount of pruning we can do since, generally speaking, the larger the automorphism group of C is, the more pruning we can do using the left and right automorphism groups of W and C.

Of course, the particular left pattern P in $L(W, C)$ on which the algorithm is running also affects the running time of our algorithm. However, if for two codes $C_1, C_2 \in \mathcal{D}(W)$ we have $L(W, C_1) = L(W, C_2)$, then we have generally found that if for a particular P in $L(W, C_1)$, the running time of our algorithm, when run on C_1 and P, is relatively slow (when compared to the running times for each of the other left patterns in $L(W, C_1)$), then the running time of our algorithm will also be relatively slow when run on C_2 and P (compared to the running times for each of the other left patterns in $L(W, C_2)$). Therefore, the relative overall running times of C_1 and C_2 are more a function of the different characteristics of C_1 and C_2 than the left patterns in $L(W, C_1)$ and $L(W, C_2)$.

4.10 The Results Thus Far

Thus far, we have used our algorithm to search 155 of our 478 codes, leaving us with 323 codes to search. The incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD has not been found in any of the codes we have searched.

We have searched each of the 37 codes in $\mathcal{D}(d_4)$, each of the 17 codes in $\mathcal{D}(d_3^*)$, and each of the 44 codes in $\mathcal{D}(d_3)$. We have also searched 54 of the 115 codes in $\mathcal{D}(d_2^*)$ including all of the d_3-codes in $\mathcal{D}(d_2^*)$. Finally, we have searched 3 of the 122 codes in $\mathcal{D}(d_2)$.

Overall, we have searched all of the d_4-codes and the d_3-codes. We have also searched all of the codes that contain 13 or more weight 4 words.

In Appendix C, we give a one line summary of the results of our search algorithm for each code. Included for each code C is the overall running time of the algorithm, the number of left patterns in $L(W, C)$, and the minimum and maximum deepest level reached by our BIBD search algorithm, over all all left patterns in $L(W, C)$. The maximum deepest level reached by our algorithm gives us an indication of just how close we came to finding an incidence matrix in C. The minimum deepest level reached by our algorithm gives us an indication as to how "difficult" it may be to theoretically eliminate the left patterns in $L(W)$. That is, if for a given left word block W, we find there exists a code $C \in \mathcal{D}(W)$ in which the minimum deepest level reached by our algorithm was relatively large, say ≥ 10, then it may be difficult to theoretically prove that any of the left patterns in $L(W)$ cannot occur in the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD.

4.11 Where To Go From Here

The codes we have left to search are the 11 distance 8 codes, all the 132 d_1-codes, and 180 of the 205 d_2-codes (including 61 of the 83 d_2-codes in $\mathcal{D}(d_2^*)$).

In this section, we will discuss several ideas we have that may help us in searching for an incidence matrix in these remaining 323 codes. First, we will discuss ideas we have for improving the aspects of our algorithm that occur before we begin our BIBD search algorithm for a particular code C and left pattern P, such as the different left word blocks W we use and the number of left patterns produced by our Left Pattern Algorithm. We will then discuss ideas we have for improving both our BIBD search algorithm for a particular code C and left pattern P, and the way in which we run our algorithm. Our BIBD search algorithm for a particular C and P and the manner in which we run our algorithm are the areas we feel we have the greatest chance of improving our search.

Let us first consider the left word blocks we use. One idea that may be worth investigating is to determine whether or not there exist left word blocks, other than the ones listed in Figure 4.1, that result in our overall search algorithm for a particular code

running faster (as opposed to the time our algorithm would take using one of the left word blocks in Figure 4.1). For example, one can investigate whether or not there exist combinations of weight 5 words that will result in our overall algorithm running faster. However, we believe we have found all the left word blocks that have a relatively small number of left patterns associated with them. Therefore, the left word blocks we have not considered all probably have too many left patterns associated with them to be of practical use.

Let us now consider the number of left patterns produced by our Left Pattern Algorithm for a particular code (i.e. the number of left patterns in $L(W)$). The idea we have for improving this aspect of our search is to try to eliminate left patterns from the list $L(W)$ produced by our Left Pattern Algorithm, for a given left word block W. However, as our results in Appendix C indicate, our Left Pattern Algorithm probably already eliminates all the left patterns from $L(W)$ that can be easily eliminated (using methods other than searching all codes for an incidence matrix that begins with a given left pattern P in $L(W)$). For example, for code 345 in Appendix C, in which we used the d_2^*-block in our search, our algorithm managed to fill in at least 14 rows of an incidence matrix that begins with P, for each left pattern P in $L(W, C)$. Therefore, it will probably be quite difficult to theoretically eliminate any of the left patterns in $L(d_2^*)$.

Other ideas for improving our search that may be worth considering include the specific word block in a code we select, and examining the different structures in a particular code to see if we can find anything that will aid us in our search. However, we feel the aspects of our search that we have the greatest chance of improving are our BIBD search algorithm for a particular code C and left pattern P (both the algorithm and how we implement it) and the manner in which we run our algorithm (both the hardware we use and how we use it).

Some of the ideas we have for speeding up our BIBD search algorithm include finding better ways of sorting the rows of our left patterns and looking at ways in which we can

eliminate the redundant work performed by the algorithm (i.e. further ways in which we can prune the BIBD search tree).

Let us first consider sorting the rows of P. As was pointed out in Section 4.7, the running time of our BIBD search algorithm, for a given code C and left pattern P, is greatly affected by the way in which the rows of P are ordered. Therefore, before we begin our BIBD search algorithm, we sort the rows of the left pattern P (as described in Section 4.7). However, our method of sorting the rows of P may not necessarily be the best method. Therefore, it may be worth investigating whether or not there exist better ways to sort the rows of our left patterns. Furthermore, we have not yet tested the "on the go" method (mentioned in Section 4.7) of sorting the rows of P. Testing may show this method actually works better than our method of sorting the rows of P before the search begins. Even if the "on the go" method does not prove better, it may still aid us in finding better ways to sort the rows of our left patterns.

Let us now consider the redundant work performed by the algorithm. Even though we have included pruning in our BIBD search algorithm, we have not necessarily eliminated all of the redundant work performed by the algorithm. For example, since $\mathcal{AUT}(C)$ may contain permutations that are neither left nor right automorphisms, our algorithm does not take full advantage of the automorphism group of C. Therefore, it may be worth investigating whether or not there exist ways in which we can further use the automorphism group of C and the sets $R_{i,m}$ to further prune our BIBD search tree. For example, if our code contains a word block W' that is equivalent to the left word block W we are using in our search, then $\mathcal{AUT}(C)$ may contain automorphisms that swap W and W'. Whether or not we can efficiently use such automorphisms to reduce the amount of work performed by our algorithm is a topic that still needs to be researched.

Another idea for improving our BIBD search algorithm is to look for ways in which we can further optimize our implementation of the algorithm. For example, the manner in which we produce and manipulate the sets $R_{i,m}$, where $m \in N_i$, in our program may

not necessarily be the most efficient.

Let us now consider the manner in which we run our BIBD search algorithm (i.e. the hardware point of view). One obvious improvement we can make is to run our search algorithm on a faster computer (we have run our algorithm on a Solaris 2.5). Another approach is to run our search algorithm on more than once machine (i.e. run it in parallel). The advantage in this approach is it would result in us having much more cpu time for our search (thus far, we've only run our algorithm sequentially on one machine). For example, if for a set of say 100 codes, we find it takes, on average, six months to search each code, with an upper bound of nine months, then if we search these codes in parallel on 100 machines then our search will complete in nine months (as opposed to 600 months, or 50 years, if we searched each code sequentially on one machine).

Of the ideas discussed in this section, this parallel approach looks to be the most promising one for searching the remaining 323 undecided codes.

4.12 Concluding Remarks

We will now conclude this chapter by reviewing the main results presented in the chapter.

In this chapter, we described in detail our algorithm for determining whether or not a given $(33, 16)$ doubly-even self-orthogonal code C contains an incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. The algorithm works by performing a backtrack search, with pruning, on the weight 12 words in C.

Thus far, we have been able to use our algorithm to search 155 of the 478 $(33, 16)$ doubly-even self-orthogonal codes that we have not been able to eliminate theoretically. None of the codes we haved searched contained an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD. Included among the codes we have searched are all the d_4-codes and the d_3-codes. We have also searched all of the codes that contain 13 or more weight 4 words.

Though we have not been able to search every code in our list of 478 codes (not

eliminated theoretically), and thus have not determined whether or not a $(22, 33, 12, 8, 4)$-BIBD exists, we do believe our algorithmic approach to this problem may still work. Therefore, we also gave, in this chapter, several ideas on how we may be able to improve our search approach in order to search the remaining codes.

Chapter 5

Summary

We conclude this thesis with a summary of the results we have presented in the thesis.

In Chapter 2, we proved that any $(33, k)$ doubly-even self-orthogonal code over $GF(2)$ is contained in a $(33, 16)$ doubly-even self-orthogonal code over $GF(2)$. Since the point code of a $(22, 33, 12, 8, 4)$-BIBD is a $(33, k)$ doubly-even self-orthogonal code, this implies a $(22, 33, 12, 8, 4)$-BIBD exists if and only if there exists a $(33, 16)$ doubly-even self-orthogonal code that contains the incidence matrix of such a design. This fact forms the basis of the approach we took to the problem we investigated in this thesis: "does a $(22, 33, 12, 8, 4)$-BIBD exist?"

In [2], we enumerated a list of inequivalent $(2k, k)$ self-dual codes over $GF(2)$, for $2k = 2, 4, \ldots, 32$. We have since used the methods discussed in [2] to enumerate a list of inequivalent $(34, 17)$ self-dual codes over $GF(2)$. We have also made improvements to our algorithms in [2] that have allowed us to find the automorphism groups of each of the codes we have enumerated. Using our list of $(34, 17)$ self-dual codes, along with their automorphism groups, we have enumerated a list L of inequivalent $(33, 16)$ doubly-even self-orthogonal codes (that do not have coordinates of zeros) along with their automorphism groups. The list L contains 594 codes.

The importance of our list L, with respect to the question of whether or not a $(22, 33, 12, 8, 4)$-BIBD exists, is as follows: Suppose there exists a $(22, 33, 12, 8, 4)$-BIBD B. Then there must exist a $(33, 16)$ doubly-even self-orthogonal code that contains the incidence matrix of B. Since the columns of the incidence matrix of B must all have weight 8, the code C does not have a coordinate of zeros. Furthermore, any code that is equivalent to C will contain an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD that is isomorphic to B. Therefore, our list L of codes must contain a code that contains the incidence matrix of a design that is isomorphic to B. Thus, our complete list L of 594 inequivalent $(33, 16)$ doubly-even self-orthogonal codes (that do not have a coordinate of zeros) has the property that a $(22, 33, 12, 8, 4)$-BIBD exists if and only if there exists a code in L that contains the incidence matrix of such a design.

In Section 2.5 we counted the total number of $(33, 16)$ doubly-even self-orthogonal codes that do not have a coordinate of zeros. We also showed how we can use this total, along with the automorphism groups of our codes, to check whether or not our list L is in fact complete. Generator matrices and generators for the automorphism groups for each code in L are available on the world wide web at:

www.cs.umanitoba.ca/~umbilou1/DoublyEvenCodes/

In Chapter 3, we gave proofs that certain classes of codes in L cannot contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. Two of our proofs made use of our column weight solution algorithm, which is a generalization of the methods used by Hamada and Kobayashi [9] to find the block intersection patterns of a $(22, 33, 12, 8, 4)$-BIBD. This algorithm also proved useful in developing a proof, independent of the results of Greig [8], that there are three different possibilities, up to row and column rearrangement, for the columns, in the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD, that are supported by a weight 4 word in the orthogonal complement of the point code of the design.

The classes of codes in L that we were able to theoretically eliminate in Chapter 3 were the e_3-codes, the e_{2i}-codes, for $i \geq 2$, and the d_i-codes, for $i \geq 5$. This eliminates

116 codes from our list L of 594 codes, leaving us with the 478 d_i-codes, for $1 \leq i \leq 4$, and distance 8 codes in L.

Ideally, if a $(22, 33, 12, 8, 4)$-BIBD does not exist, we would like to find theoretical proofs that show every code in L cannot contain the incidence matrix of such a design. Unfortunately, we have not been able to find such proofs for the d_i-codes, where $1 \leq i \leq 4$, and the distance 8 codes. Therefore, we must search the weight 12 words of each of these codes to determine whether or not they contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD.

In Chapter 4, we described our algorithm for determining whether or not a given code in L contains the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. At the time of this writing, we have used our algorithm to search 155 of the 478 codes in L that we have not eliminated theoretically, leaving us with 323 undecided codes. Included in the codes we have searched are all the d_4-codes and all the d_3-codes in L. We have also searched all the codes in L that contain 13 or more weight 4 words. We did not find an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD in any of the codes we have searched. Thus, we know the d_4-codes and the d_3-codes do not contain an incidence matrix for such a design.

Overall, through both theory and with our search algorithm, we have shown that the point code of a $(22, 33, 12, 8, 4)$-BIBD cannot contain an e_3-block, an e_{2i}-block, for $i \geq 2$, and a d_i-block, for $i \geq 3$. This implies the point code of such a design cannot contain three or more weight 4 words that all have a value of 1 in one coordinate of C.

Unfortunately, due to the large number of weight 12 words in each of our codes, we have not been able to search every code in L that we need to search. Therefore, it is still not known whether or not a $(22, 33, 12, 8, 4)$-BIBD exists. However, we do believe our algorithm demonstrates it is not infeasible to search every code. We also believe our algorithms may aid in finding proofs that certain classes of codes do not contain the incidence matrix of such a design. In Section 4.10, we gave several ideas on how we might both improve our algorithm, and improve the manner in which we execute our algorithm,

in order to search the remaining codes in our list L.

Appendix A

Weight Distributions

In this appendix, we give tables containing the weight distributions of the $(33, 16)$ doubly-even self-orthogonal codes that do not have a coordinate of zeros. Since we are only dealing with codes that are doubly-even, we only give the number of weight $4i$ words in each distribution. We also include with each weight distribution the number of equivalence classes of codes that have the given distribution.

The weight distributions of the dual codes can also be easily obtained from the tables. That is, if the weight distribution of a code C is $(A_0, A_4, \ldots, A_{32})$, then the number of weight $33 - 4i$ words in $C^{\perp}$ is equal to the number of weight $4i$ words in $C^{\perp}$ which is equal to A_{4i}.

As the tables demonstrate, the weight distributions for the $(33, 16)$ doubly-even self-

orthogonal codes that do not have a coordinate of zeros all have the general form:

$$\begin{aligned} A_0 &= 1 \\ A_4 &= x \\ A_8 &= 411 + 10x \\ A_{12} &= 10886 - 49x \\ A_{16} &= 35511 + 76x \\ A_{20} &= 17556 - 49x \\ A_{24} &= 1165 + 10x \\ A_{28} &= 6 + x \\ A_{32} &= 0 \end{aligned}$$

where x is the number of weight 4 words. Therefore, the number of weight 4 words in such a code uniquely determines the weight distribution of the code.

weight distribution									number of equivalence classes with distribution
0	**4**	**8**	**12**	**16**	**20**	**24**	**28**	**32**	
1	0	411	10886	35511	17556	1165	6	0	11
1	1	421	10837	35587	17507	1175	7	0	21
1	2	431	10788	35663	17458	1185	8	0	24
1	3	441	10739	35739	17409	1195	9	0	35
1	4	451	10690	35815	17360	1205	10	0	39
1	5	461	10641	35891	17311	1215	11	0	35
1	6	471	10592	35967	17262	1225	12	0	41
1	7	481	10543	36043	17213	1235	13	0	41
1	8	491	10494	36119	17164	1245	14	0	28
1	9	501	10445	36195	17115	1255	15	0	35
1	10	511	10396	36271	17066	1265	16	0	29
1	11	521	10347	36347	17017	1275	17	0	25
1	12	531	10298	36423	16968	1285	18	0	25
1	13	541	10249	36499	16919	1295	19	0	22
1	14	551	10200	36575	16870	1305	20	0	17
1	15	561	10151	36651	16821	1315	21	0	18
1	16	571	10102	36727	16772	1325	22	0	18
1	17	581	10053	36803	16723	1335	23	0	11
1	18	591	10004	36879	16674	1345	24	0	13
1	19	601	9955	36955	16625	1355	25	0	13
1	20	611	9906	37031	16576	1365	26	0	9
1	21	621	9857	37107	16527	1375	27	0	8
1	22	631	9808	37183	16478	1385	28	0	8
1	23	641	9759	37259	16429	1395	29	0	4

weight distribution									number of equivalence classes with distribution
0	**4**	**8**	**12**	**16**	**20**	**24**	**28**	**32**	
1	24	651	9710	37335	16380	1405	30	0	6
1	25	661	9661	37411	16331	1415	31	0	8
1	26	671	9612	37487	16282	1425	32	0	3
1	27	681	9563	37563	16233	1435	33	0	8
1	28	691	9514	37639	16184	1445	34	0	3
1	29	701	9465	37715	16135	1455	35	0	3
1	30	711	9416	37791	16086	1465	36	0	5
1	31	721	9367	37867	16037	1475	37	0	3
1	33	741	9269	38019	15939	1495	39	0	3
1	34	751	9220	38095	15890	1505	40	0	1
1	35	761	9171	38171	15841	1515	41	0	2
1	36	771	9122	38247	15792	1525	42	0	3
1	37	781	9073	38323	15743	1535	43	0	2
1	39	801	8975	38475	15645	1555	45	0	2
1	41	821	8877	38627	15547	1575	47	0	2
1	43	841	8779	38779	15449	1595	49	0	1
1	45	861	8681	38931	15351	1615	51	0	4
1	55	961	8191	39691	14861	1715	61	0	1
1	57	981	8093	39843	14763	1735	63	0	2
1	65	1061	7701	40451	14371	1815	71	0	1
1	85	1261	6721	41971	13391	2015	91	0	1

Appendix B

The Codes Eliminated by Theory

In this appendix, we give tables describing the characteristics of the 116 inequivalent $(33, 16)$ doubly-even self-orthogonal codes that we have theoretically proven cannot contain the incidence matrix of a $(22, 33, 12, 8, 4)$-BIBD. The information provided in the tables for each code C is:

- **id**: a unique number we assign to each code.
- **type**: the highest precedenced weight 4 block in C (as defined in Section 3.4).
- **weight 4**: the number of weight 4 words in C.
- $|\mathcal{AUT}(\mathbf{C})|$: the size of the automorphism group of C.

The weight distribution of each code C can easily be found since, as Appendix A demonstrates, the weight distribution of each code is uniquely determined by the number of weight 4 words in C.

id	type	weight 4	$\|\mathcal{AUT}(\mathbf{C})\|$
593	e_3	85	4284987369062400
592	d_{10}	65	78479622144000
591	d_8	57	29964946636800
590	e_4	57	20975462645760
589	e_3	55	7491236659200
588	e_8	45	1664719257600
587	e_4	45	1165303480320
586	e_4	45	832359628800
585	d_8	45	428070666240
584	e_3	43	1248539443200
583	d_7	41	237817036800
582	e_3	41	166471925760
581	d_6	39	178362777600
580	e_4	39	124853944320
579	e_3	37	178362777600
578	d_6	37	118908518400
577	e_3	36	218494402560
576	e_4	36	152946081792
575	e_3	36	72831467520
574	d_5	35	84934656000
573	d_7	35	23781703680
572	e_3	34	31213486080
571	d_6	33	23781703680
570	e_4	33	16647192576
569	e_4	33	11890851840
568	e_3	31	17836277760

id	type	weight 4	$\|\mathcal{AUT}(\mathbf{C})\|$
567	d_5	31	8493465600
566	d_6	31	5945425920
565	e_3	30	20808990720
564	e_3	30	14863564800
563	d_6	30	13377208320
562	e_3	30	10404495360
561	e_4	30	9364045824
560	e_3	29	23781703680
559	d_5	29	4246732800
558	e_3	29	2972712960
557	e_3	28	4954521600
556	e_3	28	2229534720
555	d_6	28	1486356480
553	e_3	27	5462360064
552	e_3	27	5462360064
551	d_5	27	5096079360
550	e_3	27	2601123840
549	d_5	27	2548039680
548	d_5	27	2548039680
547	e_4	27	1189085184
545	e_3	26	1486356480
544	e_3	26	1486356480
543	e_3	25	7134511104
539	e_3	25	2378170368
538	d_6	25	1981808640
536	d_5	25	424673280

id	type	weight 4	$\lvert\mathcal{AUT}(\mathbf{C})\rvert$
535	e_3	24	1486356480
534	e_3	24	1040449536
532	d_5	24	637009920
531	e_3	24	520224768
530	e_4	24	445906944
529	e_3	23	1189085184
528	d_5	23	424673280
527	e_3	23	371589120
525	e_3	22	2675441664
524	e_3	22	445906944
520	e_3	22	148635648
519	e_3	22	123863040
518	d_5	22	106168320
516	e_3	21	260112384
515	e_3	21	198180864
514	e_4	21	198180864
511	e_3	21	130056192
510	d_5	21	70778880
506	e_3	20	148635648
504	e_3	20	74317824
503	e_3	20	74317824
502	e_3	20	74317824
500	e_3	19	445906944
499	e_3	19	334430208
495	d_5	19	141557760
492	e_3	19	55738368

id	type	weight 4	$\|\mathcal{AUT}(C)\|$
488	e_3	19	37158912
487	e_3	18	222953472
485	e_3	18	173408256
484	e_4	18	148635648
481	d_5	18	53084160
475	e_3	18	12386304
473	e_3	17	66060288
469	e_3	17	18579456
468	e_3	17	18579456
467	e_3	17	18579456
462	e_3	16	24772608
455	e_3	16	13934592
454	e_3	16	12386304
448	e_3	16	6193152
445	d_5	15	928972800
443	e_4	15	650280960
442	e_3	15	495452160
433	e_3	15	6193152
428	e_3	15	1548288
424	e_3	14	8257536
416	e_3	14	2064384
410	e_3	13	33030144
404	e_3	13	3096576
395	e_3	13	1548288
394	e_3	13	1548288
388	e_3	12	24772608

id	type	weight 4	$\lvert\mathcal{AUT}(\mathbf{C})\rvert$
375	e_3	12	1032192
370	e_3	12	774144
357	e_3	11	2064384
346	e_3	11	344064
336	e_3	10	7741440
318	e_3	10	258048
308	e_3	9	3096576
297	e_3	9	290304
274	e_3	8	596090880
272	e_3	8	387072
246	e_3	7	23224320
243	e_3	7	943488

Appendix C

Results of our BIBD Search Algorithm

In this appendix, we give tables summarizing the results of our BIBD search algorithm. The tables contain one entry for each of the 478 inequivalent $(33, 16)$ doubly-even self-orthogonal codes that we have not eliminated theoretically. The information provided in the tables for each code C includes:

- **id**: a unique number we assign to each code.
- **type**: the highest precedenced weight 4 block in C (as defined in Section 3.4).
- **block**: the highest precedenced word block in $C^{\perp}$ (as defined in Section 4.4). This is the type of the left word block W that our search algorithm used for the code.
- **wt 4**: the number of weight 4 words in C.
- **wt 12**: the number of weight 12 words in C that do not intersect any of the weight 4 words in C in 4 positions.
- $|\mathcal{AUT}(\mathbf{C})|$: the size of the automorphism group of C.

At the time of writing, we have managed to search 155 of the codes, leaving us with 323 codes we have not yet searched. For each of the codes that we have searched, the tables also include the following information:

- **pats**: the number of left patterns in the set $L(W, C)$. That is, the number of left patterns P for which our BIBD search algorithm searched the weight 12 words of C for an incidence matrix $A(W)$ that begins with P. Note that for some of the d_4-codes, **pats** has a value a 0. The reason for this is that these d_4-codes do not contain many of the left words in $\mathcal{LEFT}(d_4)$, and as a result, do not contain any of the left patterns in $L(d_4, C)$.

- **min**: the minimum deepest level reached by our BIBD search algorithm in its search for an incidence matrix in C. That is, if for each left pattern P in $L(W, C)$, if we let $x(P)$ denote the the maximum number of rows our BIBD search algorithm for C and P managed to fill in, in an incidence matrix A that begins with P, then **min** is equal to the minimum value of $x(P)$ over each P in $L(W, C)$.

- **max**: the maximum deepest level reached by our BIBD search algorithm in its search for an incidence matrix in C.

- **time**: the total time it took to search for an incidence matrix in C. The system we used to search our codes was a Solaris 2.5.

Thus far, for each of the codes we have searched, we have *not* found an incidence matrix for a $(22, 33, 12, 8, 4)$-BIBD.

As our tables indicate, we have completely searched the sets of codes $\mathcal{D}(d_4)$, $\mathcal{D}(d_3^*)$, and $\mathcal{D}(d_3)$. We have also searched the codes in $\mathcal{D}(d_2^*)$ up to code number 345, and the codes in $\mathcal{D}(d_2)$ up to code number 385.

Since $\mathcal{D}(d_4)$ contains every d_4-code, we have eliminated the d_4-codes with our search. Each d_3-code is an element of one of the three sets $\mathcal{D}(d_3^*)$, $\mathcal{D}(d_2^*)$, and $\mathcal{D}(d_3)$. Since we

have searched every code in $\mathcal{D}(d_3^*)$ and $\mathcal{D}(d_3)$, plus every d_3-code in $\mathcal{D}(d_2^*)$ (as our tables indicate), we have eliminated the d_3-codes with our search. As our tables demonstrate, we have also eliminated every code that contains 13 or more weight 4 words.

Generator matrices for each of the codes listed in the tables (along with generators for their automorphism groups) can be found on the world wide web at:

www.cs.umanitoba.ca/~umbilou1/DoublyEvenCodes/

Codes						Search Results			
id	**type**	**block**	**wt 4**	**wt 12**	$\lvert\mathcal{AUT}(C)\rvert$	**pats**	**min**	**max**	**time**
554	d_4	d_4	27	4608	5662310400	21	2	4	4.5 min
546	d_4	d_4	26	4160	2123366400	21	1	4	1.6 min
542	d_4	d_4	25	5056	3397386240	21	1	5	3.3 min
541	d_4	d_4	25	4352	3397386240	21	1	5	2.5 min
540	d_4	d_4	25	3328	3397386240	0	0	0	22.1 sec
537	d_4	d_4	25	4800	1698693120	21	2	5	3.1 min
533	d_4	d_4	24	4480	707788800	21	3	11	3.2 min
526	d_4	d_4	23	4544	283115520	21	2	5	2.5 min
523	d_4	d_4	22	5056	424673280	21	3	6	1.8 min
522	d_4	d_4	22	4832	212336640	21	2	6	1.7 min
521	d_4	d_4	22	4648	212336640	21	3	5	1.4 min
517	d_3	d_3^*	21	4864	339738624	2	5	5	35.7 sec
513	d_3	d_3^*	21	5120	169869312	2	5	5	1.1 min
512	d_4	d_4	21	5088	141557760	21	3	20	2.1 min
509	d_4	d_4	20	3760	637009920	0	0	0	22.5 sec
508	d_4	d_4	20	5296	212336640	21	2	7	1.2 min
507	d_4	d_4	20	5184	212336640	14	3	6	54.5 sec
505	d_4	d_4	20	5160	106168320	21	2	7	1.1 min
501	d_4	d_4	20	5184	35389440	21	4	17	1.6 min
498	d_3	d_2^*	19	5632	169869312	8	1	12	2.7 min
497	d_3	d_3^*	19	5504	169869312	2	6	6	39.7 sec
496	d_3	d_3	19	5568	169869312	20	4	13	6.1 min
494	d_3	d_2^*	19	5568	84934656	8	4	9	2.7 min
493	d_3	d_2^*	19	5760	84934656	8	5	11	4.8 min
491	d_4	d_4	19	5476	53084160	21	4	17	1.1 min

Codes						Search Results			
id	**type**	**block**	**wt 4**	**wt 12**	$\lvert\mathcal{AUT}(\mathbf{C})\rvert$	**pats**	**min**	**max**	**time**
490	d_4	d_4	19	5448	53084160	21	3	16	1.3 min
489	d_4	d_4	19	5376	47185920	14	5	19	52.8 sec
486	d_3	d_3^*	18	5528	191102976	2	6	6	26.4 sec
483	d_3	d_2^*	18	5952	127401984	8	4	13	2.6 min
482	d_3	d_2^*	18	5760	63700992	8	4	12	3.9 min
480	d_3	d_3^*	18	5480	42467328	2	5	5	23.9 sec
479	d_4	d_4	18	5696	35389440	21	4	17	58.2 sec
478	d_3	d_2^*	18	5792	21233664	13	3	12	11.6 min
477	d_4	d_4	18	5652	17694720	21	3	16	1.2 min
476	d_4	d_4	18	5532	17694720	21	3	9	1.3 min
474	d_4	d_4	17	5856	94371840	14	5	20	48.1 sec
472	d_3	d_3	17	6016	56623104	20	5	13	8.3 min
471	d_3	d_3	17	6080	28311552	35	3	18	15.8 min
470	d_3	d_3	17	6080	28311552	35	4	15	13.3 min
466	d_3	d_2^*	17	5952	14155776	8	4	13	3.7 min
465	d_3	d_3^*	17	5696	14155776	2	19	20	57.4 sec
464	d_4	d_4	17	5740	8847360	21	5	19	1.2 min
463	d_3	d_3	16	6432	63700992	35	1	18	9.8 min
461	d_3	d_2^*	16	6248	21233664	8	5	13	13.9 min
460	d_3	d_3^*	16	5992	21233664	2	9	9	22.5 sec
459	d_3	d_2^*	16	6080	21233664	8	4	9	2.3 min
458	d_3	d_2^*	16	6368	21233664	8	5	13	18.4 min
457	d_3	d_2^*	16	6144	21233664	8	3	13	3.8 min
456	d_3	d_3^*	16	5944	21233664	2	5	5	17.5 sec
453	d_3	d_2^*	16	6320	10616832	13	1	13	4.4 min

Codes						Search Results			
id	**type**	**block**	**wt 4**	**wt 12**	$\lvert\mathcal{AUT}(C)\rvert$	**pats**	**min**	**max**	**time**
452	d_3	d_2^*	16	6272	10616832	8	4	12	4.7 min
451	d_3	d_2^*	16	6128	10616832	13	3	13	29.2 min
450	d_3	d_2^*	16	6328	10616832	13	3	13	11.8 min
449	d_3	d_2^*	16	6144	7077888	8	5	13	26.6 min
447	d_4	d_4	16	5944	5898240	14	5	18	51.7 sec
446	d_4	d_4	16	6008	5898240	21	6	19	1.2 min
444	d_4	d_4	15	4416	707788800	0	0	0	24.9 sec
441	d_2	d_2	15	6720	63700992	11	4	18	18.2 min
440	d_3	d_3	15	6592	28311552	20	5	18	7.0 min
439	d_3	d_3^*	15	6164	15925248	2	17	17	28.8 sec
438	d_3	d_2^*	15	6412	15925248	8	3	15	5.4 min
437	d_2	d_2^*	15	6492	15925248	8	4	14	9.7 min
436	d_2	d_2^*	15	6620	15925248	8	3	13	29.7 min
435	d_4	d_4	15	6272	15728640	21	2	19	1.0 min
434	d_3	d_2^*	15	6464	7077888	8	5	20	35.2 min
432	d_3	d_2^*	15	6392	5308416	13	3	19	1.0 hrs
431	d_3	d_2^*	15	6476	5308416	13	3	14	1.1 hrs
430	d_3	d_3	15	6368	4718592	35	4	18	21.8 min
429	d_4	d_4	15	6200	2949120	14	5	19	59.0 sec
427	d_2	d_2^*	14	6856	10616832	8	5	13	40.6 min
426	d_2	d_2^*	14	6740	10616832	8	3	15	8.3 min
425	d_2	d_2^*	14	6616	10616832	8	5	17	51.4 min
423	d_4	d_4	14	6432	5898240	14	5	18	42.2 sec
422	d_3	d_3	14	6592	4718592	20	6	19	20.0 min
421	d_3	d_3	14	6740	3538944	20	8	17	12.3 min

Codes						Search Results			
id	**type**	**block**	**wt 4**	**wt 12**	$\lvert\mathcal{AUT}(C)\rvert$	**pats**	**min**	**max**	**time**
420	d_3	d_2^*	14	6656	3538944	8	6	13	1.7 hrs
419	d_3	d_2^*	14	6664	3538944	13	3	15	10.6 min
418	d_3	d_3^*	14	6352	3538944	2	18	19	49.5 sec
417	d_4	d_4	14	6428	2949120	21	4	18	1.0 min
415	d_3	d_2^*	14	6724	1769472	13	5	14	1.9 hrs
414	d_3	d_2^*	14	6660	1769472	13	3	13	1.7 hrs
413	d_3	d_2^*	14	6604	1769472	8	5	19	1.7 hrs
412	d_3	d_3	14	6732	1769472	35	7	18	1.1 hrs
411	d_3	d_2^*	14	6704	1769472	13	5	19	2.0 hrs
409	d_3	d_2^*	13	6656	28311552	8	4	13	1.1 min
408	d_3	d_3^*	13	6624	9437184	2	19	19	49.1 sec
407	d_2	d_2^*	13	6960	7962624	13	1	13	39.0 min
406	d_2	d_2^*	13	7002	5308416	13	3	14	1.9 hrs
405	d_3	d_3	13	7040	4718592	35	3	19	56.4 min
403	d_2	d_2^*	13	7060	2654208	8	5	17	3.4 hrs
402	d_2	d_2^*	13	7000	2654208	8	5	15	20.2 min
401	d_2	d_2^*	13	7002	2654208	13	3	14	4.2 hrs
400	d_2	d_2^*	13	6946	2654208	8	5	17	2.2 hrs
399	d_2	d_2^*	13	6832	2654208	13	3	13	2.9 hrs
398	d_3	d_3	13	7040	2359296	35	9	20	1.2 hrs
397	d_3	d_2^*	13	6880	2359296	8	5	19	18.8 min
396	d_3	d_3	13	6916	1769472	20	8	19	22.2 min
393	d_4	d_4	13	6676	1474560	21	4	20	1.4 min
392	d_2	d_2^*	13	7072	1327104	13	5	15	4.3 hrs
391	d_2	d_2^*	13	7056	1327104	13	3	15	4.4 hrs

Codes						Search Results			
id	**type**	**block**	**wt 4**	**wt 12**	$\lvert\mathcal{AUT}(\mathbf{C})\rvert$	**pats**	**min**	**max**	**time**
390	d_3	d_3	13	7008	1179648	35	14	20	1.4 hrs
389	d_3	d_2^*	13	6908	442368	13	5	17	5.8 hrs
387	d_2	d_2^*	12	7106	7962624	13	1	14	2.3 hrs
386	d_2	d_2	12	7360	5308416	17	8	18	10.2 hrs
385	d_2	d_2	12	7360	5308416	17	8	18	10.0 hrs
384	d_3	d_3	12	7120	3538944	20	7	19	16.9 min
383	d_2	d_2^*	12	7352	3538944	8	13	17	3.7 hrs
382	d_4	d_4	12	6894	2211840	14	4	17	47.4 sec
381	d_3	d_3^*	12	6808	1769472	2	20	20	46.8 sec
380	d_2	d_2	12	7396	1769472				
379	d_2	d_2	12	7404	1769472				
378	d_2	d_2^*	12	7244	1769472	8	5	19	3.0 hrs
377	d_2	d_2^*	12	7120	1769472	8	4	14	1.0 hrs
376	d_3	d_3	12	7216	1179648	20	9	19	45.6 min
374	d_3	d_3	12	7212	884736	35	3	18	2.0 hrs
373	d_2	d_2^*	12	7244	884736	8	5	17	6.2 hrs
372	d_2	d_2^*	12	7244	884736	8	5	15	54.6 min
371	d_2	d_2^*	12	7352	884736	13	5	19	32.7 min
369	d_2	d_2^*	12	7226	663552	13	3	16	7.1 hrs
368	d_3	d_3	12	7224	589824	35	11	19	1.8 hrs
367	d_3	d_2^*	12	7172	589824	13	5	15	29.4 min
366	d_3	d_3	12	7176	589824	35	11	19	2.0 hrs
365	d_3	d_2^*	12	7152	589824	8	6	19	11.3 hrs
364	d_3	d_3	12	7180	294912	35	14	19	4.0 hrs
363	d_3	d_3^*	11	7040	4718592	2	19	20	37.6 sec

Codes						Search Results			
id	**type**	**block**	**wt 4**	**wt 12**	$\lvert\mathcal{AUT}(C)\rvert$	**pats**	**min**	**max**	**time**
362	d_2	d_2	11	7648	4718592				
361	d_2	d_2^*	11	7488	4718592				
360	d_4	d_4	11	7144	4423680	14	4	19	33.2 sec
359	d_3	d_3	11	7456	3145728	20	8	19	19.5 min
358	d_2	d_2	11	7636	2654208				
356	d_2	d_2	11	7536	1769472				
355	d_2	d_2	11	7636	884736				
354	d_3	d_3	11	7456	786432	20	15	20	1.5 hrs
353	d_2	d_2^*	11	7584	442368				
352	d_2	d_2^*	11	7498	442368				
351	d_2	d_2^*	11	7442	442368				
350	d_2	d_2^*	11	7488	442368				
349	d_2	d_2^*	11	7532	442368				
348	d_2	d_2	11	7548	442368				
347	d_3	d_3	11	7488	393216	35	15	20	5.5 hrs
345	d_3	d_2^*	11	7380	294912	8	14	20	25.8 hrs
344	d_3	d_3	11	7432	294912	35	9	19	3.8 hrs
343	d_2	d_2^*	11	7542	221184				
342	d_2	d_2	11	7594	221184				
341	d_2	d_2^*	11	7542	221184				
340	d_2	d_2^*	11	7526	221184				
339	d_3	d_3	11	7444	147456	35	15	20	7.2 hrs
338	d_4	d_4	10	5336	19585843200	0	0	0	49.3 sec
337	d_4	d_4	10	7416	11059200	21	2	17	43.8 sec
335	d_2	d_2^*	10	7552	3538944				

Codes						Search Results			
id	**type**	**block**	**wt 4**	**wt 12**	$\lvert AUT(C)\rvert$	**pats**	**min**	**max**	**time**
334	d_3	d_3^*	10	7304	2359296	2	18	18	24.3 sec
333	d_2	d_2	10	7872	1179648				
332	d_2	d_2^*	10	7784	1179648				
331	d_2	d_2	10	7916	589824				
330	d_2	d_2	10	7872	589824				
329	d_3	d_3	10	7776	393216	35	17	19	4.1 hrs
328	d_2	d_2	10	7842	331776				
327	d_3	d_3	10	7760	294912	35	15	20	8.4 hrs
326	d_3	d_3	10	7688	294912	35	15	19	3.6 hrs
325	d_2	d_2^*	10	7728	294912				
324	d_2	d_2	10	7888	294912				
323	d_2	d_2	10	7880	294912				
322	d_2	d_2^*	10	7720	294912				
321	d_2	d_2	10	7916	294912				
320	d_2	d_2^*	10	7828	294912				
319	d_2	d_2^*	10	7864	294912				
317	d_3	d_3	10	7670	221184	20	15	20	4.3 hrs
316	d_2	d_2^*	10	7738	221184				
315	d_3	d_3	10	7732	196608	20	15	19	3.7 hrs
314	d_2	d_2	10	7880	147456				
313	d_2	d_2	10	7880	147456				
312	d_2	d_2^*	10	7828	147456				
311	d_3	d_3	10	7720	98304	35	15	19	9.9 hrs
310	d_2	d_2^*	10	7780	55296				
309	d_2	d_2	9	8224	4718592				

Codes						Search Results			
id	**type**	**block**	**wt 4**	**wt 12**	$\lvert\mathcal{AUT}(\mathbf{C})\rvert$	**pats**	**min**	**max**	**time**
307	d_2	d_2	9	8224	1572864				
306	d_3	d_3^*	9	7516	1327104	2	18	20	1.0 min
305	d_2	d_2^*	9	7812	1327104				
304	d_2	d_2	9	8224	1179648				
303	d_3	d_3	9	7928	884736	20	11	19	1.7 hrs
302	d_2	d_2^*	9	8064	786432				
301	d_2	d_2^*	9	7996	442368				
300	d_2	d_2	9	8062	442368				
299	d_2	d_2	9	8072	331776				
298	d_3	d_3	9	7916	294912	20	16	20	2.8 hrs
296	d_2	d_2^*	9	8038	221184				
295	d_2	d_2	9	8224	196608				
294	d_2	d_2	9	8176	147456				
293	d_2	d_2^*	9	8100	147456				
292	d_2	d_2^*	9	7964	147456				
291	d_2	d_2	9	8114	147456				
290	d_2	d_2	9	8170	147456				
289	d_3	d_3	9	7976	98304	20	17	19	8.6 hrs
288	d_3	d_3	9	8004	73728	35	16	20	20.4 hrs
287	d_2	d_2^*	9	8096	73728				
286	d_2	d_2	9	8148	73728				
285	d_2	d_2	9	8124	73728				
284	d_2	d_2^*	9	8100	73728				
283	d_2	d_2	9	8116	73728				
282	d_2	d_2^*	9	8100	73728				

Codes						Search Results			
id	**type**	**block**	**wt 4**	**wt 12**	$\lvert\mathcal{AUT}(C)\rvert$	**pats**	**min**	**max**	**time**
281	d_2	d_2	9	8180	73728				
280	d_2	d_2^*	9	8090	73728				
279	d_2	d_2	9	8142	73728				
278	d_3	d_3	9	7988	49152	35	16	19	29.2 hrs
277	d_2	d_2	9	8152	36864				
276	d_2	d_2^*	9	8054	36864				
275	d_2	d_2	9	8106	36864				
273	d_2	d_2^*	8	8384	393216				
271	d_2	d_2^*	8	8248	294912				
270	d_2	d_2	8	8418	147456				
269	d_2	d_2^*	8	8284	147456				
268	d_2	d_2	8	8500	98304				
267	d_2	d_2	8	8408	98304				
266	d_2	d_2	8	8436	98304				
265	d_2	d_2	8	8464	98304				
264	d_2	d_2^*	8	8384	98304				
263	d_3	d_3	8	8238	49152	20	18	20	14.0 hrs
262	d_2	d_2	8	8444	49152				
261	d_2	d_2	8	8444	49152				
260	d_2	d_2	8	8444	49152				
259	d_2	d_2	8	8436	49152				
258	d_2	d_2^*	8	8420	49152				
257	d_3	d_3	8	8254	36864	35	17	20	44.5 hrs
256	d_2	d_2^*	8	8338	36864				
255	d_2	d_2	8	8390	36864				

Codes						Search Results			
id	**type**	**block**	**wt 4**	**wt 12**	$\lvert\mathcal{AUT}(C)\rvert$	**pats**	**min**	**max**	**time**
254	d_2	d_2^*	8	8334	36864				
253	d_2	d_2	8	8354	36864				
252	d_2	d_2	8	8472	24576				
251	d_2	d_2	8	8472	24576				
250	d_2	d_2	8	8414	18432				
249	d_2	d_2	8	8396	18432				
248	d_2	d_2^*	8	8334	18432				
247	d_2	d_2	8	8414	18432				
245	d_3	d_3^*	7	8024	4423680	2	9	20	59.7 sec
244	d_3	d_3	7	8408	2949120	20	15	20	25.1 min
242	d_2	d_2	7	8536	884736				
241	d_1	d_1^*	7	8800	393216				
240	d_1	d_1^*	7	8800	393216				
239	d_2	d_2	7	8704	196608				
238	d_3	d_3	7	8552	147456	35	15	20	14.0 hrs
237	d_2	d_2	7	8760	98304				
236	d_1	d_1^*	7	8800	98304				
235	d_3	d_3	7	8508	82944	35	16	20	20.3 hrs
234	d_2	d_2^*	7	8530	73728				
233	d_2	d_2	7	8680	73728				
232	d_1	d_1^*	7	8800	65536				
231	d_3	d_3	7	8524	55296	20	16	20	16.4 hrs
230	d_2	d_2	7	8760	49152				
229	d_2	d_2	7	8732	49152				
228	d_2	d_2^*	7	8656	49152				

Codes						Search Results			
id	**type**	**block**	**wt 4**	**wt 12**	$\lvert\mathcal{AUT}(C)\rvert$	**pats**	**min**	**max**	**time**
227	d_2	d_2	7	8664	36864				
226	d_2	d_2^*	7	8666	36864				
225	d_2	d_2^*	7	8620	36864				
224	d_2	d_2	7	8764	36864				
223	d_2	d_2^*	7	8588	27648				
222	d_2	d_2	7	8686	27648				
221	d_2	d_2	7	8732	24576				
220	d_2	d_2^*	7	8680	24576				
219	d_2	d_2	7	8736	24576				
218	d_2	d_2	7	8746	24576				
217	d_2	d_2	7	8672	24576				
216	d_2	d_2	7	8736	24576				
215	d_2	d_2	7	8760	12288				
214	d_2	d_2	7	8718	12288				
213	d_2	d_2	7	8700	12288				
212	d_2	d_2	7	8736	12288				
211	d_2	d_2	7	8746	12288				
210	d_2	d_2^*	7	8684	12288				
209	d_2	d_2	7	8668	9216				
208	d_2	d_2	7	8736	6144				
207	d_2	d_2	7	8708	6144				
206	d_2	d_2	7	8764	6144				
205	d_3	d_3^*	6	8240	23224320	2	9	20	1.1 min
204	d_2	d_2^*	6	8520	23224320				
203	d_1	d_1^*	6	9120	589824				

Codes						Search Results			
id	**type**	**block**	**wt 4**	**wt 12**	$\lvert\mathcal{AUT}(C)\rvert$	**pats**	**min**	**max**	**time**
202	d_3	d_3	6	8768	221184	20	17	19	2.7 hrs
201	d_1	d_1^*	6	9092	131072				
200	d_2	d_2	6	8972	110592				
199	d_2	d_2^*	6	8864	110592				
198	d_2	d_2^*	6	8788	110592				
197	d_1	d_1^*	6	9120	98304				
196	d_1	d_1^*	6	9120	98304				
195	d_3	d_3	6	8812	82944	35	16	20	19.4 hrs
194	d_2	d_2	6	8972	82944				
193	d_2	d_2	6	9046	73728				
192	d_1	d_1^*	6	9036	73728				
191	d_2	d_2	6	9004	36864				
190	d_2	d_2^*	6	8874	36864				
189	d_1	d_1^*	6	9064	32768				
188	d_1	d_1^*	6	9064	32768				
187	d_2	d_2	6	8954	24576				
186	d_1	d_1^*	6	9120	24576				
185	d_1	d_1^*	6	9036	24576				
184	d_2	d_2	6	8940	20736				
183	d_2	d_2	6	9014	18432				
182	d_2	d_2	6	9014	18432				
181	d_2	d_2^*	6	8952	12288				
180	d_2	d_2	6	8954	12288				
179	d_2	d_2^*	6	8938	9216				
178	d_1	d_1^*	6	9092	8192				

Codes						Search Results			
id	type	block	wt 4	wt 12	$\|\mathcal{AUT}(C)\|$	pats	min	max	time
177	d_1	d_1^*	6	9092	8192				
176	d_1	d_1^*	6	9064	8192				
175	d_1	d_1^*	6	9064	8192				
174	d_2	d_2	6	8990	6144				
173	d_2	d_2	6	9014	6144				
172	d_2	d_2^*	6	8934	6144				
171	d_2	d_2	6	9018	6144				
170	d_2	d_2	6	9018	3072				
169	d_2	d_2	6	9032	3072				
168	d_2	d_2	6	9032	3072				
167	d_2	d_2	6	9032	3072				
166	d_2	d_2	6	8990	3072				
165	d_2	d_2	6	9014	1536				
164	d_2	d_2^*	5	9040	221184				
163	d_2	d_2	5	9256	98304				
162	d_1	d_1^*	5	9344	65536				
161	d_2	d_2	5	9312	49152				
160	d_1	d_1^*	5	9400	49152				
159	d_1	d_1^*	5	9400	49152				
158	d_2	d_2	5	9224	13824				
157	d_1	d_1^*	5	9372	12288				
156	d_1	d_1^*	5	9386	10240				
155	d_2	d_2	5	9260	9216				
154	d_1	d_1^*	5	9400	8192				
153	d_1	d_1^*	5	9372	8192				

Codes						Search Results			
id	type	block	wt 4	wt 12	$\|\mathcal{AUT}(C)\|$	pats	min	max	time
152	d_1	d_1^*	5	9358	8192				
151	d_2	d_2^*	5	9204	6144				
150	d_2	d_2	5	9312	6144				
149	d_2	d_2^*	5	9222	4608				
148	d_2	d_2	5	9330	4608				
147	d_1	d_1^*	5	9386	4096				
146	d_1	d_1^*	5	9358	4096				
145	d_1	d_1^*	5	9330	4096				
144	d_1	d_1^*	5	9330	4096				
143	d_2	d_2	5	9284	3072				
142	d_2	d_2	5	9284	3072				
141	d_2	d_2	5	9316	3072				
140	d_1	d_1^*	5	9400	2048				
139	d_1	d_1^*	5	9386	2048				
138	d_1	d_1^*	5	9386	2048				
137	d_1	d_1^*	5	9358	2048				
136	d_1	d_1^*	5	9358	2048				
135	d_1	d_1^*	5	9358	2048				
134	d_1	d_1^*	5	9386	2048				
133	d_2	d_2	5	9316	1536				
132	d_2	d_2	5	9298	1536				
131	d_2	d_2	5	9302	1536				
130	d_2	d_2	5	9302	768				
129	d_2	d_2^*	4	9328	552960				
128	d_2	d_2^*	4	9400	368640				

Codes						Search Results			
id	type	block	wt 4	wt 12	$\lvert\mathcal{AUT}(\mathbf{C})\rvert$	pats	min	max	time
127	d_2	d_2	4	9480	110592				
126	d_1	d_1^*	4	9666	98304				
125	d_2	d_2	4	9508	55296				
124	d_2	d_2^*	4	9492	18432				
123	d_2	d_2	4	9600	12288				
122	d_2	d_2^*	4	9474	10368				
121	d_2	d_2^*	4	9506	6912				
120	d_1	d_1^*	4	9596	6144				
119	d_1	d_1^*	4	9680	6144				
118	d_2	d_2	4	9572	4608				
117	d_1	d_1^*	4	9694	4096				
116	d_1	d_1^*	4	9638	4096				
115	d_1	d_1^*	4	9666	4096				
114	d_1	d_1^*	4	9652	3072				
113	d_1	d_1^*	4	9666	3072				
112	d_2	d_2	4	9614	2304				
111	d_2	d_2	4	9568	2304				
110	d_1	d_1^*	4	9666	2048				
109	d_1	d_1^*	4	9624	2048				
108	d_1	d_1^*	4	9666	2048				
107	d_2	d_2	4	9600	1536				
106	d_1	d_1^*	4	9680	1536				
105	d_1	d_1^*	4	9680	1536				
104	d_2	d_2	4	9586	1152				
103	d_2	d_2	4	9600	1152				

Codes						Search Results			
id	type	block	wt 4	wt 12	$\|\mathcal{AUT}(C)\|$	pats	min	max	time
102	d_2	d_2	4	9614	1152				
101	d_2	d_2	4	9582	1152				
100	d_1	d_1^*	4	9638	1024				
99	d_1	d_1^*	4	9680	1024				
98	d_1	d_1^*	4	9680	1024				
97	d_1	d_1^*	4	9680	1024				
96	d_1	d_1^*	4	9666	1024				
95	d_1	d_1^*	4	9638	1024				
94	d_1	d_1^*	4	9638	1024				
93	d_1	d_1^*	4	9680	768				
92	d_1	d_1^*	4	9666	256				
91	d_1	d_1^*	4	9680	256				
90	d_2	d_2	3	9864	73728				
89	d_1	d_1	3	10000	36864				
88	d_2	d_2^*	3	9756	27648				
87	d_1	d_1^*	3	9888	24576				
86	d_1	d_1^*	3	9944	16384				
85	d_2	d_2^*	3	9788	10368				
84	d_1	d_1	3	10000	6144				
83	d_2	d_2	3	9864	4608				
82	d_1	d_1	3	10000	3456				
81	d_1	d_1^*	3	9930	3072				
80	d_1	d_1^*	3	9958	3072				
79	d_2	d_2	3	9864	2304				
78	d_1	d_1^*	3	9958	2304				

Codes						Search Results			
id	**type**	**block**	**wt 4**	**wt 12**	$\lvert\mathcal{AUT}(\mathbf{C})\rvert$	**pats**	**min**	**max**	**time**
77	d_1	d_1^*	3	9972	2048				
76	d_1	d_1^*	3	9916	1536				
75	d_1	d_1^*	3	9958	1536				
74	d_1	d_1^*	3	9958	1536				
73	d_1	d_1	3	10000	1536				
72	d_2	d_2	3	9896	1152				
71	d_1	d_1^*	3	9916	1024				
70	d_1	d_1^*	3	9972	1024				
69	d_1	d_1^*	3	9930	768				
68	d_2	d_2	3	9896	576				
67	d_1	d_1^*	3	9972	512				
66	d_1	d_1^*	3	9986	512				
65	d_1	d_1^*	3	9958	384				
64	d_1	d_1^*	3	9958	384				
63	d_1	d_1^*	3	9972	256				
62	d_1	d_1^*	3	9958	256				
61	d_1	d_1^*	3	9986	256				
60	d_1	d_1^*	3	9972	256				
59	d_1	d_1^*	3	9986	256				
58	d_1	d_1^*	3	9972	128				
57	d_1	d_1^*	3	9972	128				
56	d_1	d_1^*	3	9958	128				
55	d_1	d_1^*	2	10232	12288				
54	d_1	d_1^*	2	10204	6144				
53	d_1	d_1^*	2	10260	4096				

Codes						Search Results			
id	type	block	wt 4	wt 12	$\lvert\mathcal{AUT}(\mathbf{C})\rvert$	pats	min	max	time
52	d_1	d_1^*	2	10204	3456				
51	d_1	d_1^*	2	10232	3072				
50	d_1	d_1^*	2	10218	1152				
49	d_1	d_1^*	2	10260	1024				
48	d_1	d_1^*	2	10260	1024				
47	d_1	d_1^*	2	10274	768				
46	d_1	d_1	2	10288	512				
45	d_1	d_1^*	2	10246	384				
44	d_1	d_1^*	2	10246	256				
43	d_1	d_1^*	2	10274	192				
42	d_1	d_1^*	2	10260	192				
41	d_1	d_1^*	2	10274	192				
40	d_1	d_1	2	10288	128				
39	d_1	d_1^*	2	10274	128				
38	d_1	d_1^*	2	10260	128				
37	d_1	d_1^*	2	10260	128				
36	d_1	d_1^*	2	10274	128				
35	d_1	d_1	2	10288	128				
34	d_1	d_1	2	10288	64				
33	d_1	d_1^*	2	10260	64				
32	d_1	d_1^*	2	10274	32				
31	d_1	d_1	1	10584	258048				
30	d_1	d_1	1	10584	36864				
29	d_1	d_1	1	10584	10752				
28	d_1	d_1	1	10584	10752				

Codes						Search Results			
id	type	block	wt 4	wt 12	$\lvert\mathcal{AUT}(C)\rvert$	pats	min	max	time
27	d_1	d_1^*	1	10556	3072				
26	d_1	d_1^*	1	10542	576				
25	d_1	d_1^*	1	10570	384				
24	d_1	d_1	1	10584	256				
23	d_1	d_1^*	1	10556	192				
22	d_1	d_1	1	10584	192				
21	d_1	d_1	1	10584	192				
20	d_1	d_1	1	10584	144				
19	d_1	d_1	1	10584	96				
18	d_1	d_1^*	1	10556	96				
17	d_1	d_1	1	10584	64				
16	d_1	d_1^*	1	10570	64				
15	d_1	d_1^*	1	10570	64				
14	d_1	d_1^*	1	10570	64				
13	d_1	d_1^*	1	10570	48				
12	d_1	d_1	1	10584	48				
11	d_1	d_1	1	10584	12				
10	f_0	f_0	0	10886	11520				
9	f_0	f_0	0	10886	1536				
8	f_0	f_0	0	10886	1080				
7	f_0	f_0	0	10886	384				
6	f_0	f_0	0	10886	384				
5	f_0	f_0	0	10886	192				
4	f_0	f_0	0	10886	144				
3	f_0	f_0	0	10886	60				

Codes						Search Results			
id	**type**	**block**	**wt 4**	**wt 12**	$\lvert AUT(C)\rvert$	**pats**	**min**	**max**	**time**
2	f_0	f_0	0	10886	24				
1	f_0	f_0	0	10886	24				
0	f_0	f_0	0	10886	8				

Bibliography

[1] Th. Beth, D. Junghickel, and H. Lenz. *Design Theory.* Bibliographisches Institut, Zurich, 1985.

[2] R. T. Bilous. A recursive enumeration of the inequivalent binary $(2k, k)$ self-dual codes, for $2k \leq 32$. Master's thesis, University of Manitoba, 1998.

[3] R. T. Bilous. *The* $(33, 16)$ *Doubly-Even Self-Orthogonal Codes.* (Internet site), http://www.cs.umanitoba.ca/~umbilou1/DoublyEvenCodes/, August 2000.

[4] J. H. Conway and V. Pless. On the enumeration of self-dual codes. *J. Combin. Theory Ser. A 28*, pages 26–53, 1980.

[5] J. H. Conway, V. Pless, and N. J. A. Sloane. The binary self-dual codes of length up to 32: a revised enumeration. *J. Combin. Theory Ser. A 60*, pages 183–195, 1992.

[6] R. A. Fisher and F. Yates. *Statistical Tables for Biological, Agricultural and Medical Research.* Longman, London, 1^{st} edition, 1938.

[7] R. A. Fisher and F. Yates. *Statistical Tables for Biological, Agricultural and Medical Research*, volume 93. Hafner, New York, 6^{th} edition, 1963.

[8] M. Greig. An improvedment to connor's criteria. preprint.

[9] N. Hamada and Y. Kobayshi. On the block structure of bib designs with parameters $v = 22$, $b = 33$, $r = 12$, $k = 8$, and $\lambda = 4$. *J. Combin. Theory, Ser. A 24*, pages 75–83, 1978.

[10] M. Hall Jr., R. Roth, G. H. J. van Rees, and S. A. Vanstone. On designs $(22, 33, 12, 8, 4)$. *J. Combin. Theory 47*, pages 157–175, 1988.

[11] S. Kapralov. Combinatorial $2-(22, 8, 4)$ designs with automorphisms of order 3 fixing one point. In *Math. and Education in Math., Proc. of the XVI Spring Conference of Union of Bulgarian Mathematicians*, pages 453–458, Sunny Beach, 1987.

[12] I. Landgev and V. Tonchev. Automorphisms of $2 - (22, 8, 4)$ designs. *Discrete Mathematics 77*, pages 177–189, 1989.

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

[14] F. J. MacWilliams, N. J. A. Sloane, and J. G. Thompson. Good self dual codes exist. *Discrete Mathematics 3*, pages 153–162, 1972.

[15] F. J. MaxWilliams. A theorem on the distribution of weights in a systematic code. *Bell System Tech. J. 42*, pages 79–94, 1963.

[16] B.D. McKay and S.P. Radziszowski. Towards deciding the existence of 2-$(22, 8, 4)$ designs. *J. Combin. Theory 22*, pages 211–222, 1996.

[17] B.D. McKay and S.P. Radziszowski. 2-$(22, 8, 4)$ designs have no blocks of type 3. *J. Combin. Theory 30*, pages 251–253, 1999.

[18] V. Pless. A classification of self-orthogonal codes over $gf(2)$. *Discrete Math. 3*, pages 209–246, 1972.

[19] G. H. J. van Rees. $(22, 33, 12, 8, 4)$-BIBD, an update. In *Computational and Constructive Design Theory*, pages 337–357. W.D. Wallis, Kluwer Academic Publ., 1996.

[20] V.Pless and N. J. A. Sloane. On the classification and enumeration of self-dual codes. *J. Combin. Theory Ser. A 18*, pages 313–335, 1975.

Index

Glossary of Symbols