© Copyright by Ari Trachtenberg, 1996

COMPUTATIONAL METHODS IN CODING THEORY

BY

ARI TRACHTENBERG

S.B., Massachusetts Institute of Technology, 1994

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1996

Urbana, Illinois

COMPUTATIONAL METHODS IN CODING THEORY

Ari Trachtenberg, M.S.
Computer Science
University of Illinois at Urbana-Champaign, 1996
Alexander Vardy, C.L. Liu, Advisor

We consider various computational techniques in algebraic coding theory along two lines of work. First we investigate optimization of non-linear codes by relaxing minimum distance constraints, developing, in the process, two algorithms for improving a given non-linear code and a method of visualizing algebraic codes in three dimensions. Secondly, we study the Generalized Lexicographic Construction, and show that it produces as special cases the lexicodes and derivatives with properties such as trellis-orientation, trellis-state boundedness, and local optimality. We implement algorithms for generating these families of codes and, in the process, improve upon work by Conway and Sloane [9], Brualdi and Pless [6], Kschischang and Horn [24], and Zhang [48].

To my future wife Felicia, the $\sqrt{2}$, and my family

Acknowledgments

First and foremost, the author would like to thank his Electrical Engineering advisor Professor Alexander Vardy for his patience, guidance, and encouragement throughout the course of this work. The author would also like to thank his Computer Science advisor Professor Dave Liu, whose support made this cross-disciplinary work possible.

In addition, the author would like to thank the Computational Science in Engineering program for their financial support in the form a fellowship that has allowed for this continuing work. The author would also be remiss were he not to thank his father, Lazar Trachtenberg, for his professional criticism and guidance throughout this process.

Finally, the author is required to thank his future wife Felicia Moss whose persistent encouragement and support allowed for a timely completion of this work.

Contents

Chapter

1 Introduction . . . 1
1.1 Motivation . . . 1
1.2 Algebraic Codes . . . 2
1.3 Communications Model . . . 3
1.4 Trellis Decoding . . . 4
1.5 Problem Statement . . . 6
1.6 Thesis Organization . . . 6

2 Relaxed Minimum Distance Constraints . . . 9
2.1 Maximum Likelihood Decoding . . . 9
2.2 Probability of Error . . . 11
2.3 Optimality of Perfect Codes . . . 14
2.4 Improving Known Codes . . . 14
2.5 Augmenting Non-Linear Codes . . . 15
2.6 Visualization . . . 17

3 The Lexicographic Construction . . . 21
3.1 Lexicodes . . . 21
3.2 Theoretical Underpinnings . . . 23
3.2.1 Equivalence of Lexicodes and Lexicographic Construction . . . 23
3.2.2 Relation among Subsequent Lexicodes . . . 25
3.3 Bounds on Code Parameters . . . 29

4 Generalized Lexicographic Construction . . . 36
4.1 Terminology . . . 37
4.2 Choosing Subsequent Vectors . . . 39
4.2.1 Standard Lexicographic Construction . . . 39
4.2.2 Trellis-Oriented . . . 42
4.2.3 Corollaries . . . 44

4.2.4 Locally Optimal Codes 44
4.3 The Algorithm 46
4.4 State Bounded Codes 49
4.5 Computations 50
4.5.1 Data 50
4.5.2 Analysis 51

5 Conclusion 53
5.1 Summary 53
5.2 Future Directions 54

Appendix

A Sample Algorithm Output 56

B *GLC* Code Parameters 58
B.1 Lexicodes and Trellis-Oriented *GLC* codes 58
B.2 State Bounded *GLC* Codes 67

C Comparisons 71

Bibliography 74

List of Tables

2.1 Displayed are improvements to several known codes. 15
2.2 Augmented codes generated by the algorithm in Section 2.5. 17

3.1 Generator matrix for the $(8,4,4)$ lexicode. 22
3.2 A graphical diagram relating to the inductive proof of Theorem 3.1. . 24
3.3 Coset leaders and buddies for the $(4,1,4)$ lexicode. 26
3.4 A partial table of coset leaders of the $(6,2,4)$ lexicode. 28

4.1 *MSGM* for the $(8,4,4)$ code in Table 3.1. 39

B.1 Parameters of $\mathbf{d = 4}$ codes. 59
B.2 Parameters of $\mathbf{d = 6}$ codes. 62
B.3 Parameters of $\mathbf{d = 8}$ codes. 65
B.4 Codes with a trellis log-state bound of 4. 68
B.5 Codes with a trellis log-state bound of 5. 69
B.6 Codes with a trellis log-state bound of 6. 70

List of Figures

1.1 An example of a non-linear code. 8
1.2 An example of a linear code. 8
1.3 A minimal trellis of the $(6, 2, 4)$ linear code in Figure 1.2. 8

2.1 A Binary Symmetric Channel with crossover probability p. 10
2.2 This graph compares the probability of error of a code and its improvement. 16
2.3 Code improvement as a function of crossover probability. 16
2.4 A visualization of the $(6, 2, 4)$ linear code of Figure 1.2. 20

3.1 A simple $(n = 3, M = 4, d = 2)$ binary lexicode. 22
3.2 The length of the dimension 3 lexicodes plotted versus their minimum distance d. 32
3.3 Lower bounds on lexicode dimension. 35

4.1 State-complexity comparison of a trellis-oriented code and extended BCH code. 51
4.2 State-complexity comparison of a trellis-oriented code and regular BCH code. 52

A.1 The $(64, 50, 6)$ trellis-oriented GLC code. 57

C.1 Decoding complexity comparison of $d = 6$ codes. 72
C.2 Decoding complexity comparison of $d = 8$ codes. 72
C.3 Maximum state comparison of $d = 6$ codes. 73
C.4 Maximum state comparison of $d = 8$ codes. 73

Notation

Unless otherwise stated or inferred from context, all arithmetic operations throughout this paper are carried out over the binary field $\mathbb{F}_2$. Furthermore, examples are based on the following code, $\mathbb{C}$, also found in Figure 1.2 on page 8:

0	0	0	0	0	0
0	0	1	1	1	1
1	1	1	1	0	0
1	1	0	0	1	1

Introduced	*Expression*	*Denotes*	*Example*
Page 2	(n, M, d) code	a code with length n containing M vectors of minimum Hamming distance d from each other.	See Figure 1.1 on page 8
Page 2	(n, k, d) code	a linear code of length n representing a k dimensional subspace of vectors with minimum Hamming distance d from each other.	See Figure 1.2 on page 8
Page 11	$\delta(v, \mathbb{C})$ or $\delta(v)$	the minimum distance from a vector v to a code $\mathbb{C}$; $\mathbb{C}$ is omitted in context.	$\delta(000011, \mathbb{C}) = 2$
Page 11	$\mathbf{B}_c$	the ball of vectors that are closer to the codeword c than to any other codeword, ties broken arbitrarily.	$\mathbf{B}_{000000}$ =all weight 0, 1 and 2 vectors
Page 18	$\text{bin}(i)$	The binary representation of the integer i	$\text{bin}(6) = 110$
Page 18	$a\|b$	The concatenation of a and b	$(111 \mid 010) = 111010$
Page 23	a^i	The concatenation of a with itself i times	$(01)^3 = 010101$
Page 23	$\mathcal{L}_k^d(\mathbb{C})$ or $\mathcal{L}_k^d$ or $\mathcal{L}_k$	the k'th code with minimum distance d produced by the lexicographic construction starting with the code $\mathbb{C}$ code; d is omitted in context; if $\mathbb{C}$ is omitted, we assume that $\mathbb{C}$ is the zero code (i.e. no codewords)	$\mathcal{L}_4^8$ is given in Figure 3.1 on page 22
Page 23	$L(n, k, d)$ or L_k	the lexicode with parameters (n, k, d); n and k omitted in context.	we show that these are equivalent to $\mathcal{L}_k^d$ so the same example holds

continued on next page...

continued from previous page...

Introduced	*Expression*	*Denotes*	*Example*
Page 26	$\beta_v(l)$ or $\beta(l)$	the buddy of coset leader l under v; i.e. the coset leader of the coset containing $l+v$. v is omitted in context.	$\beta_{0110}(0101) = 0011$
Page 26	$\kappa(v)$	the coset containing the vector v	$\kappa(000001) = \{000001, 001110, 111101, 110010\}$; see also the standard array on page 4
Page 27	$\bar{a}$	the binary complement of a.	$\overline{01010} = 10101$
Page 27	$\mathrm{f}(a,l)$	notational convenience for $(a\|l)$.	$\mathrm{f}(000,101) = 000101$
Page 27	$\mathrm{h}(a,l)$	notational convenience for $(\bar{a}\|\beta_v(l))$; v is understood from context.	based on $v = 0110$, $\mathrm{h}(010,0101) = 1010011$
Page 30	n_m	the length of $\mathcal{L}_m^d$; d is supplied in context.	for $d=4$, n_3=7 as per Table B.1 on page 59
Page 38	$\mathrm{lsl}(v)$	the location of the least significant bit of a bit sequence.	$\mathrm{lsl}(10110) = 2$
Page 38	$\mathrm{msl}(v)$	the most significant location of a bit sequence.	$\mathrm{msl}(10110) = 5$
Page 38	*MSGM*	a Minimum Span Generator Matrix.	Table 4.1 on page 39
Page 40	G_i	row i of the generator matrix G.	G_3 for the code generator matrix in Table 3.1 on page 22 is 1010101
Page 37	$\mathrm{GLC}^i(r,w,\mathbb{C})$ or GLC^i or GLC	the Generalized Lexicographic Construction; if arguments are supplied, refers to the GLC based on $\mathbb{C}$ over the functions r and w iterated i times, where missing arguments are inferred from context	See Example 6 on page 37
Page 37	$r_l(\mathcal{C})$	a function returning the covering radius of the code $\mathcal{C}$	$r_l(\mathbb{C}) = 3$
Page 37	$w_l(\mathcal{C})$	a function returning the lexicographically earliest vector of distance $r_l(\mathcal{C})$ from $\mathcal{C}$	$w_l(\mathbb{C}) = 010101$
Page 43	$w_t(\mathcal{C})$	a function returning the lexicographically latest vector of distance $r_l(\mathcal{C})$ from $\mathcal{C}$	$w_t(\mathbb{C}) = 101010$

continued on next page...

continued from previous page...

Introduced	*Expression*	*Denotes*	*Example*
Page 39	$w_o(\mathcal{C})$	a function returning the vector of distance $r_l(\mathcal{C})$ from $\mathcal{C}$ that produces locally optimal code parameters for subsequent codes	$w_o(\mathbb{C}) = 101010$

Chapter 1

Introduction

Algebraic coding theory (henceforth coding theory) is a branch of engineering with roots in mathematics and applications to computer science. It mainly concerns the development and design of efficient communication schemes over unreliable channels. Obvious forms of channels include a noisy telephone line, an inaccurately controlled Compact Disk laser, or even data bus communications on a computer amid bombardment by outside radiation. Since algebraic codes are used to correct errors induced over such channels, they are sometimes called error-correcting codes. At its mathematical foundation, coding theory is a classical problem in sphere packing in which codewords are represented by spheres, and our objective is to pack as many spheres as possible into an $n-$dimensional space. From its applied perspective, coding theory includes the polynomial approximation of global searches that are known to be NP-hard.

1.1 Motivation

The applications of coding theory have grown to some importance in the last several years, making the computational study of codes quite exciting. Codes have been used in various applications for various different reasons.

Codes are often used when there is little or no error tolerance across a communications channel, as some information loses all value when any part of it is garbled. For example, encrypted messages generally cannot be decrypted if even the smallest error is introduced into the transmission. As another example, consider programming code in *C* or *FORTRAN*. If any part of the text is garbled, the programming code is rendered un-compilable at best, and erroneous at worst.

Another use of codes concerns increasing throughput. This has mainly been evidenced in the last several years by the upgraded modem speeds from 9.6K Baud to 28.8K Baud. Essentially, a regular phone line introduces so many errors at 28.8K

Baud, that normal transmission becomes useless. However, with the aid of error-correcting codes, it is possible to transmit at the higher speed and correct the errors incurred, maintaining a good transmission quality. Codes are currently used in Compact Disks to correct inevitable storage errors induced by imprecise laser technology [17]. They are also used in Random Access Memory storage for maintaining memory coherence; in smaller RAM's a simple parity-check code is employed, but more sophisticated codes are required when memory storage increases. Modem lines, internet lines, and especially wireless cellular phones (which communicate over very noisy media) also employ various error-correcting codes. Finally, satellite and deep space communications require rather complicated codes, such as the Golay Code. Thus, the study of algebraic codes is quite practical, and the development of complicated codes inevitably will require the use of vast computational resources.

1.2 Algebraic Codes

A code is any set of M vectors over a finite field $\mathbb{F}_q^n$ with the property that any two vectors in the code are of Hamming distance greater than or equal to d. The code is said to have parameters (n, M, d) and is sometimes called a non-linear block code. Figure 1.1 depicts examples and counter-examples of a $(6, 3, 4)$ code.

It is often convenient to restrict codes to a certain structure. For this purpose, we make use of a *linear* code, which is any k-dimensional subspace of $\mathbb{F}_q^n$ with the property that any two vectors in the code are of Hamming distance greater than or equal to d. A linear code thus has parameters (n, k, d) and is depicted in Figure 1.2.

Because of its structure, an (n, k, d) linear code $\mathbb{C}$ is also said to have a $k \times n$ *generator matrix* G such that $\mathbb{C} = \left\{vG : v \in \mathbb{F}_q^k\right\}$. Similarly, the code $\mathbb{C}$ also has a *parity check matrix* H such that $Hc^T = 0 \quad \forall c \in \mathbb{C}$; for any vector v, Hv^t is called the *syndrome* of v and is 0 precisely for $v \in \mathbb{C}$. For example, the generator matrix and parity check matrix for the linear code in Figure 1.2 are:

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

1.3 Communications Model

In the basic coded-communication model, a message is encoded at a source, sent over a noisy channel, and decoded at the receiver. As explained in [4], encoding of a non-linear code is achieved by an explicit, one-to-one correspondence between message bits and codewords, and decoding is achieved by finding the closest codeword to the channel output and inverting the correspondence. For a linear code, the correspondence is represented by the generator matrix, which describes a transformation from the space of messages to the space of encodings. The decoding is done by using a *standard array*[4, p.41] which organizes all the vectors in the encoding space into cosets of the code over $\mathbb{F}_2$, each with its own *coset leader* of lowest weight.[1] Given this model of communications, a code of minimum distance d can be used to correct $(d-1)/2$ errors. The following example illustrates a typical coded communication.

Example 1 *Consider that we want to send a message "1 0 1 1" over a noisy channel using the linear code in Figure 1.2. Then we would follow these steps in its coded transmission:*

1. *Encoding:*

   (a) *We separate our message into blocks of length 2, since we are dealing with a dimension 2 code. Thus, our message becomes "1 0   1 1".*

   (b) *We encode each block of the message, by left-multiplying it by the generator matrix:*

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

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

---

[1] Cases of more than one minimum-weight vector are divided in an arbitrary, but fixed, manner.

2. *Transmission: The message "0 0 1 1 1 1 1 1 0 0 1 1" is transmitted over a noisy channel. Let us suppose that on error is made in each block, and that the received message is "**1** 0 1 1 1 1 1 0 **1** 0 1 1."*

3. *Decoding:*

   (a) *We generate the standard array of the code by listing codewords on the first row, and cosets of the code on each subsequent row, with the least-weight coset members (i.e. the coset leaders) in the first column:*

000000	001111	111100	110011
000001	001110	111101	110010
000010	001101	111110	110001
000011	001100	111111	110000
	$\ddots$		
100000	101111	011100	010011
100100	101011	011000	010111

   (b) *We find the cosets containing the received vectors in the standard array. In this case, the received words (emphasized above), are found on the last two lines of the standard array and are translated to the codeword in their column,* 001111 *in this case.*

   (c) *The errors in the received word are given precisely by the coset leader of the found cosets, if that coset is of weight $\leq (d-1)/2$.*

Though the standard array imparts a good understanding of the structure of a code, its size prevents it from being used in decoding all but the simplest of codes. Instead, for complicated codes, trellises [10, 13] are often used together with the Viterbi algorithm [33].

1.4 Trellis Decoding

A trellis is a time-indexed graph that represents a particular code. Massey [37] was the first to rigorously define the trellis, but the terminology of the definition has evolved since, and the following definition is loosely based on [33, p. 3].

Definition 1.1 ([37, 33]) *A* trellis $T = (V, E)$ *of rank n is a finite, directed, edge-labelled graph with the following properties:*

1. each vertex $v \in V$ has an associated depth $d \in \{1, 2, 3, \ldots, n\}$
2. each edge $e \in E$ connects vertices at depth i to depth $i+1$, for some i
3. there is one vertex A at depth 0 and one vertex B at depth n

The edge labels on a trellis can be used to represent any code. For example, Figure 1.3 shows the trellis for the code in Figure 1.2. There is a one-to-one correspondence between the paths from A to B and the codewords in the code, where the correspondence can be seen by concatenating the labels along the path chosen. For example, the topmost path from A to B corresponds to the codeword 111100.

The Viterbi algorithm [45] decodes a received message through a modified breadth-first search of the trellis which tries to find the codeword that is closest in Hamming distance to the received word (and, hence, the most likely transmitted word). Specifically, for any vector v received over a communications line, the Viterbi algorithm goes through a trellis level-by-level, determining, for each vertex w at depth i, which path from A to w is closest to the received vector v, making use of the previous iteration to avoid extra work. Thus, when the algorithm reaches depth n, it has found the path from A to B which is closest to v, and, hence, the codeword that was most likely sent over the channel.

[33] demonstrates that the Viterbi algorithm requires $2|E| - |V| + 1$ operations to run on a trellis with $|E|$ edges and $|V|$ vertices. Thus, in order to reduce the decoding complexity, it is often advantageous to use a *minimal* trellis. By definition, any trellis representing a code may not have fewer edges or vertices at any time index than a minimal trellis, so that the Viterbi algorithm has minimal running time on a minimal trellis.

Bahl, Cocke, Jelinek, and Raviv [2] discovered a method for constructing the unique minimal trellis representing a particular linear code $\mathbb{C}$. Their *BCJR* trellis is based on the parity-check matrix for the code being represented and is explained in further detail in [33][p.13]. However, it is important to note that, while the *BCJR* trellis is minimal for a particular code, trellis structure is actually very dependent on code permutation. Hence, the optimal trellis for a code would be one that minimizes edges and vertices over all permutations of the code.

1.5 Problem Statement

We consider two problems in this thesis.

Our first problem concerns codes in general, and involves their optimization by relaxing minimum distance constraints. Specifically, though the minimum distance of a code is the traditional measure of a code's performance, we analyze different factors that are more important than the minimum distance in determining the probability of decoding error for a code in practical and non-asymptotic cases.

Our second problem concerns linear codes, where we study parametrically good lexicographic codes (henceforth lexicodes) generated by the lexicographic construction. We consider how to construct these codes and generalize their construction for use in several applications.

1.6 Thesis Organization

Chapter 2 addresses our first problem. There we determine an expression for the probability of decoding error of a code, show the optimality of the Perfect Codes with respect to this expression, and develop two algorithms for improving non-linear codes based on these ideas. We conclude the chapter with a method of visualizing and interactively designing these highly-dimensional codes on a three-dimensional projection such as a rendered effect on a computer screen.

We address the second problem in Chapters 3 and 4. Specifically, in Chapter 3 we analyze the lexicographic construction that generates the linear lexicodes, which have very good code parameters. We discuss the equivalence of the lexicographic construction and the lexicodes discussed in [9]. More importantly, we develop a relationship between subsequent codes in the family generated by the lexicographic construction; this relationship is quite general and is also used in Chapter 4. Finally, we develop bounds on code parameters of the codes in the lexicographic construction, improving on pre-existing bounds in [6].

In Chapter 4, we generalize the lexicographic construction to produce specifically designed codes whose parameters are almost as good as the parameters of the lexicodes. We develop algorithms to generate the lexicodes and codes with locally optimal parameters. We also design two types of codes for fast decoding: trellis-oriented codes and trellis-state bounded codes. Finally, we implement the algorithms

designed to significantly extend the list of computed lexicodes first published in [9], produce codes with lower decoding complexity than the heuristically improved codes in [24], and generate trellis-state bounded codes that demonstrate some improvement over the similar work in [48].

Finally, in Chapter 5 we summarize the work of this thesis and present directions for future research. The several appendices contain the analyzed data from the algorithms we have implemented.

0	0	0	0	0	0
0	0	1	1	1	1
1	0	1	1	0	0

0	0	0	0	0	0
0	0	1	1	1	1
1	1	1	1	0	0

Figure 1.1: An example of a non-linear code. The left code *is not* a $(6, 3, 4)$ code over $\mathbb{F}_2^6$ because the Hamming distance between the first and third vectors is 3, which is less than the required distance of 4. The right code *is* a non-linear $(6, 3, 4)$ code over $\mathbb{F}_2^6$. Each row represents a vector in the code.

0	0	0	0	0	0
0	0	1	1	1	1
1	0	1	1	0	0
1	0	0	0	1	1

0	0	0	0	0	0
0	0	1	1	1	1
1	1	1	1	0	0
1	1	0	0	1	1

Figure 1.2: An example of a linear code. The right code is a $(6, 2, 4)$ linear code. The left code is not a $(6, 2, 4)$ linear code because: a) the first and third vectors have Hamming distance $3 \leq 4$, b) it is not a subspace as the sum of the first and third vectors is not in the code.

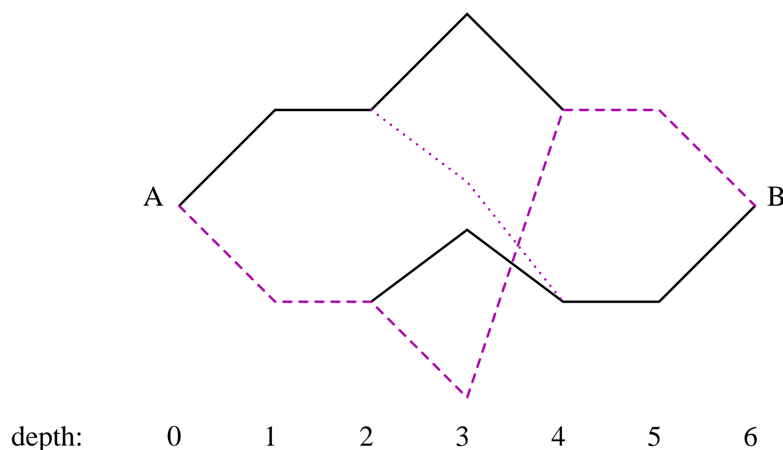

Figure 1.3: A minimal trellis of the $(6, 2, 4)$ linear code in Figure 1.2. Solid lines indicate transition on a “1” and dotted lines indicate transition on a “0”. The transition through any path from time 0 to time 6 represents one code word.

Chapter 2

Relaxed Minimum Distance Constraints

Traditionally, the minimum distance of a code has been used as the measure of a code's performance because a code with minimum distance d can correct $\lfloor (d-1)/2 \rfloor$ channel errors for any input. However, as this chapter shows, it is sometimes better to focus error-correction on only a certain fraction of inputs in favor of getting a better probability of error.

Section 2.1 begins with an explanation and intuition for the various communications models on which this work is based. In Section 2.2 we develop an expression for the probability of decoding error of a code under the models explained in Section 2.1. This expression will be used as an objective measure for a code's performance in the subsequent sections. Section 2.3 shows that perfect codes have an optimal probability of decoding error. In Section 2.4 we apply the results of Section 2.2 to develop an algorithm that optimizes a code with respect to the probability of decoding error. We investigate a similar algorithm in Section 2.5 that generates non-linear codes with good parameters. Finally, in Section 2.6 we explore the visualization and interactive development of algebraic codes, as related to the ideas in the previous sections.

2.1 Maximum Likelihood Decoding

Many channel models exist in the literature [11, Chapter 8]. Though none of the models are completely accurate, the Binary Symmetric Channel (henceforth, BSC) is a fairly good approximation to how noise interferes with transmission. Figure 2.1 shows the BSC model, in which a transmitted bit has a probability $1-p$ of being received correctly, and a probability p of being garbled in transmission, where p is called the *crossover probability*.

A decoder that minimizes its probability of decoding error conditional on the message it received from a channel is called a maximum likelihood decoder. Over a BSC, maximum likelihood decoding is achieved by minimum distance decoding [31].

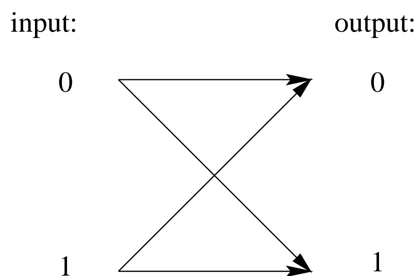

Figure 2.1: A Binary Symmetric Channel with crossover probability p.

That is to say, given a vector r received over a BSC, the optimal strategy for decoding is to decode to the closest codeword (in terms of Hamming distance) to r. Consider the following example:

Example 2 *Consider the* $(6, 3, 4)$ *non-linear code from Figure 1.1 with the input "1 2 3." We trace the steps of encoding and then minimum distance decoding:*

1. *Encode the input according to the code as the sequence of blocks: "0 0 0 0 0 0   0 0 1 1 1 1   1 1 1 1 0 0."*

2. *Send the message over the Binary Symmetric Channel. Let us assume this channel has crossover probability* $3/18$ *so that we may expect three errors in the transmission of the* 18 *bits above. One possible received sequence would then be "0 0 0 0 0 0   **1** 0 1 1 1 1   **0** 1 1 1 **1** 0" where bold face indicates an error.*

3. *Decode the received sequence by finding the closest codewords and translating back into a message word. Specifically,*

   0 0 0 0 0 0 is itself a codeword, and so is translated directly to 1.

   1 0 1 1 1 1 is clearly closest to 0 0 1 1 1 1 and is thus translated to 2.

   0 1 1 1 1 0 is of distance 2 from both codewords "1 1 1 1 0 0" and "0 0 1 1 1 1", indicating a decoding failure.

4. *The decoder outputs:* 1 2 (failure).

2.2 Probability of Error

The minimum distance of a code is often a good measure for how well the code will function under minimum distance decoding over a BSC. However, we will show that it is not an optimal measure of a code's effectiveness by showing that, by relaxing minimum distance constraints, we can increase the probability of correct decoding of a code which is, after all, the ultimate measure of a code's effectiveness. The following lemma appears in similar form, but different proof, in [31][p. 10].

Lemma 2.2.1 *Consider an (n, M, d) code $\mathbb{C}$ over a field $\mathbb{F}$ being sent over a binary symmetric channel with crossover probability p. The probability of correct decoding using minimum distance decoding (ties broken arbitrarily), where all codewords in $\mathbb{C}$ are equally probable is given by:*

$$Pr_{cd}^{\mathbb{C}} = \frac{1}{M} * \sum_{v \in \mathbb{F}^n} p^{\delta(v,\mathbb{C})} (1-p)^{n-\delta(v,\mathbb{C})} \tag{2.1}$$

$$\text{where } \delta(v, \mathbb{C}) = \min_{c \in \mathbb{C}} (v, c) \tag{2.2}$$

$\delta(v, \mathbb{C})$ is essentially the distance between v and the code $\mathbb{C}$, and, where clear from context, will be referred to as $\delta(v)$. It is one of the key factor in this probability.

Proof Consider a (one-to-one) encoding function $E : \mathbb{M} \longrightarrow \mathbb{C}$ mapping each message in the set $\mathbb{M} \hat{=} \{1 \ldots M\}$ onto a respective codeword. Similarly consider the maximum likelihood decoding function $D : \mathbb{F}^n \longrightarrow \mathbb{M}$ that hypothesizes a sent message based on a received word. Finally, consider the error e caused by the random noise through the BSC.

We can also assign to each codeword $c \in \mathbb{C}$ a ball $\mathbf{B}_c$ of all the vectors that are closer to c than to any other codeword, with ties broken according to the decoding rule. In other words, every vector in $\mathbf{B}_c$ will be decoded to c and $\mathbf{B}_c$'s partition $\mathbb{F}^n$ over all $c \in \mathbb{C}$.

Since the encoding function E is one-to-one and we are using minimum distance decoding, we may rewrite the definition of probability of correct decoding as so:

$$
\begin{aligned}
Pr_{cd}^{\mathbb{C}} &\hat{=} \sum_{m \in \mathbb{M}} \Pr\left(D\left(E(m)+e\right)=m \mid m \text{ is the message}\right) * \Pr(m \text{ is the message}) \\
&= \sum_{c \in \mathbb{C}} \Pr\left(c+e \in \mathbf{B}_c \mid c = \text{the encoded message}\right) * \Pr(c = \text{the encoded message}) \\
&= \frac{1}{M} * \sum_{c \in \mathbb{C}} \Pr\left(c+e \in \mathbf{B}_c\right)
\end{aligned}
$$

Since $\mathbf{B}_c$'s partition $\mathbb{F}^n$, we may unambiguously define a function $\bar{c}(v)$ such that $v \in \mathbf{B}_{\bar{c}(v)}$ and rewrite the summation as:

$$
= \frac{1}{M} * \sum_{v \in \mathbb{F}^n} \Pr\left(\bar{c}(v) + e' = v\right)
$$

and, finally, we are using minimum distance decoding over a BSC, so:

$$
\begin{aligned}
&= \frac{1}{M} * \sum_{v \in \mathbb{F}^n} p^{\delta(\bar{c}(v),v)} (1-p)^{n-d(\bar{c}(v),v)} \\
&= \frac{1}{M} * \sum_{v \in \mathbb{F}^n} p^{\delta(v)} (1-p)n - \delta(v)
\end{aligned}
$$

□

The requirement that all codewords in $\mathbb{C}$ should be equally probable in Lemma 2.2.1 is not very restrictive because it is the basis for the optimality of minimum distance decoding [31, p. 10].

An interesting corollary of Lemma 2.2.1 supplies us with the probability of error for decoding with a *linear* code, which is found in many textbooks on the matter, such as [32, p. 18]:

Corollary 2.2.1.1 *The probability of error of an (n,k,d) linear code $\mathbb{C}$ with w_i coset leaders of Hamming weight i is given by:*

$$
Pr_{err}^{\mathbb{C}} = 1 - \sum_{i=0}^{n} w_i p^i (1-p)^{n-i}
$$

Proof It is observed that all elements of the same coset have the same distance from $\mathbb{C}$. Thus, by Lemma 2.2.1:

$$\begin{aligned} Pr_{err}^{\mathbb{C}} &= 1 - \frac{1}{2^k} * \sum_{v \in \mathbb{F}^n} p^{\delta(v)}(1-p)^{n-\delta(v)} \\ &= 1 - \frac{1}{2^k} * \sum_{\text{coset leader } l \text{ of } C} 2^k p^{\delta}(l)(1-p)^{n-\delta(l)} \\ &= 1 - \sum_{i=0}^{n} w_i p^i (1-p)^{n-i} \end{aligned}$$

□

A non-linear form of Corollary 2.2.1.1 can similarly be deduced, with a trivial proof:

Corollary 2.2.1.2 *The probability of error of an arbitrary (n, M, d) code, where w_i indicates the number of vectors in $\mathbb{F}^n$ of distance i from the code, is given by:*

$$Pr_{err}^{\mathbb{C}} = 1 - \frac{1}{M} \sum_{i=0}^{n} w_i p^i (1-p)^{n-1}$$

To fully understand the implications of Lemma 2.2.1 we supply a simple example.

Example 3 *Consider the linear $(6, 2, 4)$ code in Figure 1.2. Through exhaustive enumeration, we can seen the following information about the vectors in $\mathbb{F}^n$ and their contribution to the probability of correct decoding for a BSC with crossover probability $p = 0.1$:*

i	w_i	$\frac{w_i}{M} p^i (1-p)^{n-i}$
0	*4*	*0.53144*
1	*24*	*0.35429*
2	*28*	*0.04592*
3	*8*	*0.01312*
total:	*64*	*0.93312*

The probability of error of this code is $1 - 0.93312 \approx 0.06688$. The vectors that are most influential in reducing this probability of error are those that are closest to the code.

2.3 Optimality of Perfect Codes

Example 1.1 seems to suggest that the optimal code, in terms of decoding error, will be as close as possible to all the vectors in the decoding sphere. This, in fact, can be made rigorous with the following lemma:

Lemma 2.3.1 *For* $0 \leq p < \frac{1}{2}$,

$$a < b \Longleftrightarrow p^a(1-p)^{n-a} > p^b(1-p)^{n-b}$$

The proof is immediate from the fact that $0 \leq p < \frac{1}{2} \quad \Longleftarrow \quad p < 1-p$.

□

Because of Lemma 2.3.1 we see that the optimal code, in terms of decoding error, will spread its codewords out as far as possible from each other, so that there will be few vectors in $\mathbb{F}^n$ that will be close to the code. In other words, w_i will be as small as possible for any i. There are at least $\binom{n}{i}$ vectors of distance i from any particular codeword. Among the linear codes, it is the Perfect Codes [4][p.43] which achieve this number with equality and will be the optimal codes for any fixed length n and number of codewords M.

2.4 Improving Known Codes

Lemma 2.2.1 gives us a better standard of a code's efficiency than the traditional minimum distance. It is true that the minimum distance of a non-trivial code[1] ultimately bounds $\delta(v)$, combining with Lemma 2.3.1 to give us the following crude bound on the probability of correct decoding:

$$Pr_{cd}^{\mathbb{C}} \leq \frac{2^n}{M} p^d(1-p)^{n-d}. \tag{2.3}$$

Nevertheless, Lemma 2.2.1.2 gives us the counter-intuitive realization that, at a certain point, for a fixed n and M, the lower the minimum distance between some codewords, the higher the probability of correct decoding. In other words, it is sometimes better to be able to decode a certain region of the the coded space very well, at the expense of other regions, in order to ultimately improve the probability of correct decoding. We can see this idea more clearly in the following example:

---

[1] By a non-trivial code we mean a code whose covering radius is not greater than its the minimum distance.

Example 4 *Consider the following non-linear* $(6,4,1)$ *code:*

0	*0*	*1*	*0*	*0*	*0*
1	*1*	*0*	*1*	*1*	*0*
0	*1*	*1*	*1*	*0*	*1*
1	*0*	*0*	*0*	*1*	*1*

Using Lemma 2.2.1 we can see that this code has probability of error ≈ 0.055216, *which is less than the probability of error of the* $(6,2,4)$ *code in Figure 1.2. Thus, despite the lower minimum distance, the code in Example 4 has the lower probability of error (and, hence, the higher probability of correct decoding). The improvement of the code in 4 can be seen in Figure 2.2*

We have implemented a simple algorithm, similar to a genetic algorithm, that, through random perturbations, tries to improve the probability of error of a code. Some results are given in Table 2.4. It is particularly important to note that the amount of improvement achieved by this algorithm is very sensitive to the crossover probability of the channel. Figure 2.3 illustrates how the improvement in probability of error of the modified $(12,4,6)$ lexicode versus the standard lexicode depends on crossover probability.

Base Code	Base Pr_{err}	Improved Pr_{err}	Gain
(6,2,4), linear (Figure 1.2)	$8.1 * 10^{-4}$	$6.2 * 10^{-4}$	23.5%
(7,8,4), non-linear[32, p. 51]	$1.366 * 10^{-3}$	$1.283 * 10^{-3}$	6.1%
(12,4,6), lexicode [9]	$9.908 * 10^{-5}$	$9.328 * 10^{-5}$	5.9%

Table 2.1: Displayed are improvements to several known codes. The improvements are the result of implementing a perturbation algorithm based on relaxing minimum distance in favor of lowering the probability of error of several known codes. The crossover probability used is $p = 0.01$

2.5 Augmenting Non-Linear Codes

The idea of random perturbations expressed in Section 2.4 can also be used to develop generate good codes. Specifically, starting with a code of fixed minimum distance d, one may randomly perturb codewords in an effort to create a hole where extra vectors can be added into the code.

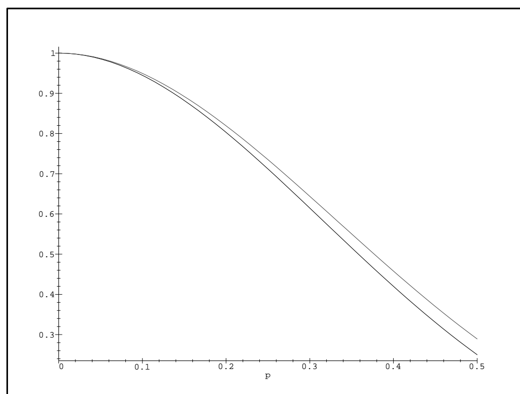

Figure 2.2: This graph compares the probability of error of a code and its improvement. Specifically, the probability of error of the non-linear $(6, 4, 1)$ code in example 4 (higher) and linear $(6, 2, 4)$ code of Figure 1.2 (lower) are plotted against the crossover probability, p, of the channel.

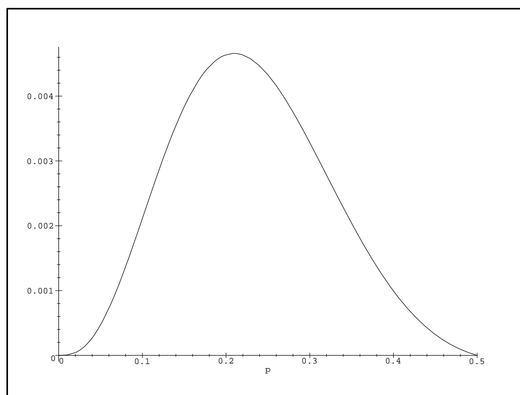

Figure 2.3: Code improvement as a function of crossover probability. The improvement in the probability of correct decoding achieved by relaxing minimum distance is plotted against the crossover probability of the channel.

Example 5 *Consider the following two codes with length* 4 *and minimum distance* 2*:*

0	*0*	*0*	*0*
0	1	1	1
1	*0*	*1*	*1*
1	*1*	*0*	*0*

Code 1

0	*0*	*0*	*0*
0	1	1	0
0	1	0	1
1	*0*	*1*	*1*
1	*1*	*0*	*0*

Code 2

No more vectors can be added to Code 1 without lowering its minimum distance of 2 *because it has a covering radius* $1 < 2$ *and is, hence, non-trivial. However, by perturbing the emphasized vector in Code 1 and changing it from* 0111 *to* 0110*, we have changed the covering radius to* 2*, so that the vector* 0101 *may be added to Code 1 to get Code 2, a code with the same minimum distance and length, but containing a larger number of vectors.*

The simplistic algorithm demonstrated in Example 5 is actually remarkably effective in generating some of the best known non-linear codes, and is also a good way of finding non-linear improvements to good linear codes. Table 2.5 shows some of the results of running this algorithm on various codes.

Length	Min. Dist.	Number of Vectors in Base Code	Number of Vectors in Augmented Code	Comments
11	4	0	72	Best known to date
12	4	0	127	Best known has 144-152 vectors
12	6	16	24	Best known to date; based on (12,4,6) Lexicode
17	8	0	32	Best known has 36-37 vectors

Table 2.2: Augmented codes generated by the algorithm in Section 2.5.

2.6 Visualization

The algorithms in Sections 2.5 and 2.4 involve a random search that does not consider the structure of a code. Though these algorithms seem to be effective, it is sometimes important to be able to visualize a code to better understand its structure and establish an intuition about its workings. However, algebraic codes tend to reside in

many dimensions, and it is necessary to project them into three dimensions so that they may be visualized.

One conventional method for viewing n-dimensional binary data involves splitting up a three-dimensional viewing area into $\binom{n}{3}$ half-spaces (polytopes) in such a way that each of the sectors formed shows the relationships among three different dimensions of the data. The problem with this technique is that it will enable viewing of only a small fraction of the the number of direction relationships[2] between vectors in $\mathbb{F}_2^n$. Specifically, since each cube has 12 edges, the number of direct relationships we will be able to view is:

$$12\binom{n}{3} = \frac{1}{2}n(n-1)(n-2) \tag{2.4}$$

However, since each vector in $\mathbb{F}_2^n$ is next to $n-1$ other vectors in the space, there are altogether $2^n(n-1)$ direct relations among vectors, a number exponentially bigger than what we view with this method. In addition, it is quite difficult to view $\binom{n}{3}$ different cubes, much less to understand the data they represent in such a fashion.

There is a way to view more of the direct relations among vectors in a a vector space, and it concerns a particular generalization of Gray code. Gray code is the sequence of vectors starting from the zero vector that is in one-to-one correspondence with the set of all binary vectors and has the distinguishing property that consecutive vectors differ in exactly one bit. Thus, the following would be a subsequence of Gray code:

$$0, 1, 10, 11, 110, 111, 101, 100, \ldots \tag{2.5}$$

Given an integer i, it is well known [40, p.886ff] that the i'th vector in the Gray code, $G(i)$, is precisely $(\text{bin}(i) \ \text{XOR} \ \text{bin}(\lfloor i/2 \rfloor))$, where $\text{bin}(i)$ refers to the binary representation of the integer i. Since Gray code is in one-to-one correspondence with all binary vectors, the inverse function $G^{-1}(v)$ is well-defined.

We now define a mapping from $\mathbb{F}_2^{n/3} \times \mathbb{F}_2^{n/3} \times \mathbb{F}_2^{n/3}$ (isomorphic to $\mathbb{F}_2^n$) to $\mathbb{Z}_{n/3}^3$ that will be the basis of our new projection. In this mapping we use $a|b$ to indicate the concatenation of a and b (e.g. $111|010 = 111010$). We map:

$$f : (x, y, z) \longrightarrow \left(G^{-1}(x), G^{-1}(y), G^{-1}(z)\right) \tag{2.6}$$

Thus, for example, $f(1, 10, 11) = (1, 2, 3)$ following the ordering in 2.5.

[2]Two vectors are directly related if they are of Hamming distance 1 from each other.

Clearly, Mapping 2.6 has the property that neighbors in the projection correspond to direct relations in $\mathbb{F}_2^n$ because of the definition of Gray code. Moreover, for n a multiple of 3, the projection of this mapping is merely a cube of length $\ell \hat{=} 2^{n/3}$ filled with length 1 cubes; Figure 2.4 shows this image as rendered by an interactive visualizer that we have developed using OpenGL. Based on our understanding of the image, we can count the number of edges in it, which corresponds to the number of visible direct relations.

Specifically, if we for any of the ℓ spaces between successive planes $x = i$ and $x = i + 1$ we can count a total of $(\ell + 1)^2$ edges. If we repeat this for the disjoint spaces between planes $y = i$ and $y = i+1$ and again for the planes $z = i$ and $z = i+1$, we get the following formula for the number of edges in the projection:

$$3\ell(\ell + 1)^2 = 3\left(2^{n/3}\right)\left(\left(2^{n/3}\right) + 1\right) \tag{2.7}$$

In the case that n is not a multiple of 3, we may merely pad n to the next multiple of 3 to get an appropriate bound. The specific fraction of all the direct relations that are visible with this method is:

$$\begin{aligned}\text{visible fraction of relations} &= \frac{3\left(2^{n/3}\right)\left(\left(2^{n/3}\right)+1\right)}{2^n(n-1)} \\ &\underset{n\longrightarrow\infty}{\longrightarrow} \frac{1}{3(n-1)}\end{aligned}$$

We can see a marked improvement in the number of direct relations that we can view with this method over the conventional method. Though we still lose a lot of information, the fractional loss is linearly bounded, rather than exponentially bounded as it is in the conventional case. Moreover, the relations are presented in a more intuitive and visible form using this method.

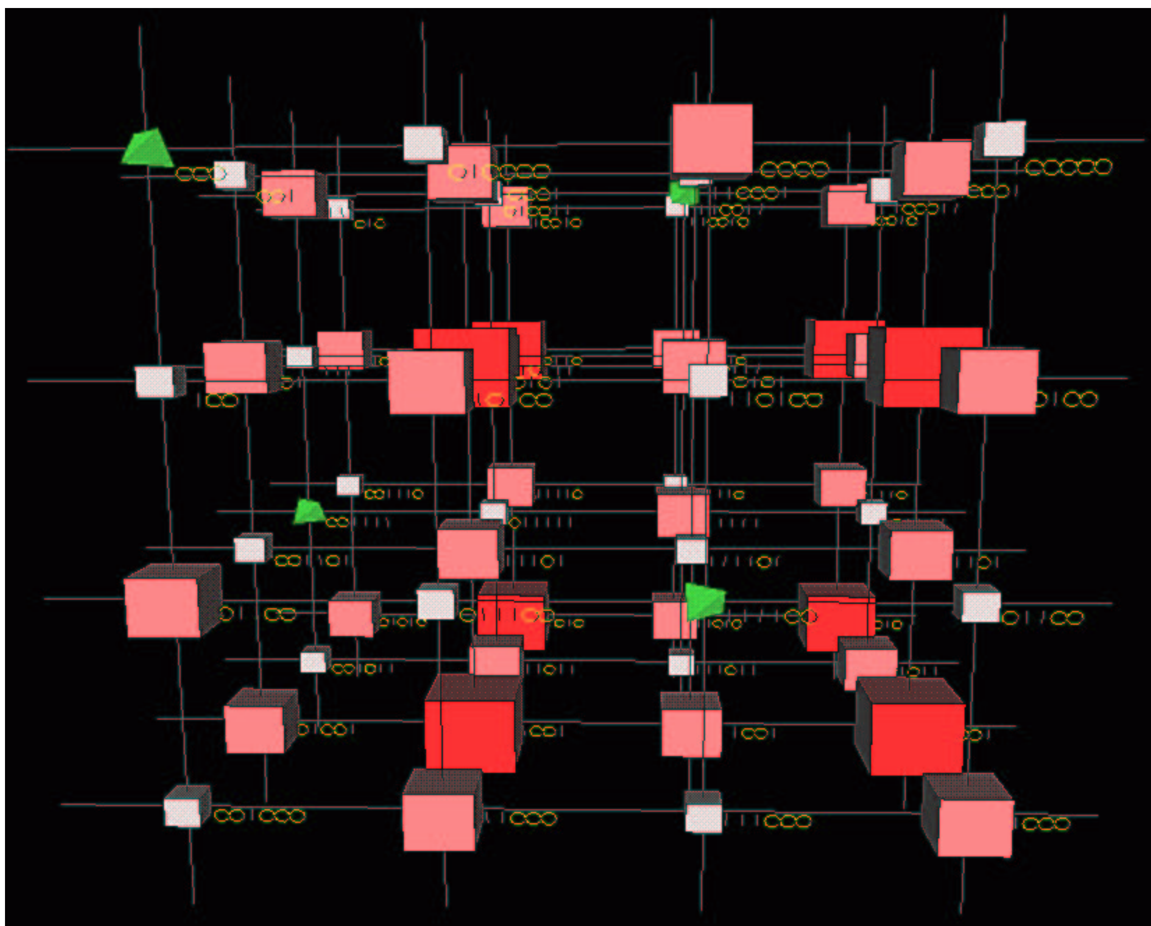

Figure 2.4: A visualization of the $(6, 2, 4)$ linear code of Figure 1.2. Visualization is based on the method described in Section 2.6.

Chapter 3

The Lexicographic Construction

We now shift our attention to a linear codes, where structure can be utilized to understand the properties of a code. Specifically, we shall concern ourselves with lexicographic codes, or lexicodes for short, and their generalization through the lexicographic construction.

In Section 3.2, we prove that the lexicographic construction does, indeed, produce lexicodes when iteratively applied to the **0** code. In addition, we establish a relationship between subsequent codes in the lexicographic construction that will be employed in subsequent sections and in Chapter 4 as the basis for a fast algorithm for generating the lexicographic construction. Based on this relationship, we develop a few bounds for the parameters of the lexicodes in Section 3.3; one of these bounds will give us a closed-form expression for the parameters of all lexicodes of dimensions 1, 2, and 3. These bounds come together to give a bound on the rate of lexicodes which is asymptotically better than that found in the similar work by Brualdi and Pless [6].

3.1 Lexicodes

Lexicodes were introduced by Conway and Sloane in [8, 9] as algebraic codes with enigmatically good code parameters. Lexicodes include, among other famous optimal codes, the Golay code, the Hamming codes and some extended Quadratic Residue codes [9, 32]. [6, 9, 30] also showed that lexicodes are linear. Trial comparison [9] to optimal linear codes with corresponding length and dimension parameters show that lexicodes are within one of the optimal minimum distance, and, hence, they are quite good approximations to optimal codes.

Lexicodes are formed by iteratively adding the lexicographically earliest vector[1]

---

[1] i.e. the earliest vector in dictionary order. Thus 01111 would come before 10000.

of distance at least d from the vectors already added. For example, to generate the binary lexicographic code of length 3 and minimum distance $d = 2$, we would set up the following lexicographically ordered table. A $\bullet$ in the table indicates that the vector is in the code.

000	001	010	011	100	101	110	111
$\bullet$			$\bullet$		$\bullet$	$\bullet$	

Figure 3.1: A simple $(n = 3, M = 4, d = 2)$ binary lexicode.

We will prove in Section 3.2 that, since lexicodes are linear, they may be produced alternatively by a more general method called the *lexicographic construction* which iteratively adds non-zero basis vectors in lexicographic order. In this construction, one is given a code $\mathbb{C}$ with parameters (n, k, d) and covering radius ρ, and adds to it the first vector in the lexicographic order that is of distance ρ from $\mathbb{C}$, padding the vector with $d - \rho$ extra ones on the left. In this way, a code with parameters $(n+d-\rho, k+1, d)$ is produced. For example, if $d = 2$ then the first two basis vectors produced by the lexicographic construction will be $\langle 011, 101 \rangle$, which generate the lexicode in Figure 3.1.

As an example, consider the $(8, 4, 4)$ code $\mathbb{C}$, with covering radius $\rho = 2$, generated by the vectors in Table 3.1. We can see that the vector 00000011 is the lexicographically earliest vector of distance $\rho(= 2)$ from $\mathbb{C}$. Hence, we pad it with $d - \rho = 2$ ones to get the new codeword 1100000011. With this new vector, we have a $(10, 5, 4)$ code. Note that the generality of this construction comes from the fact that it can generate not only ordinary lexicodes, as happens when we apply it to the $\mathbf{0}$ code, but also lexicographic extensions of any other code whose generator matrix is known.

1111
110011
1010101
11000011

Table 3.1: Generator matrix for the $(8, 4, 4)$ lexicode.

3.2 Theoretical Underpinnings

In this section we introduce some of the theoretical underpinnings of the lexicographic construction. In Subsection 3.2.1 we prove that the lexicographic construction produces exactly all the lexicodes. This will enable us to apply our results of the analysis of the lexicographic construction to lexicodes. In Subsection 3.2.2, we develop an explicit, iterative construction of the coset leaders in each coset of the code under addition over $\mathbb{F}_2$. The knowledge of these coset leaders will be the basis, in Chapter 4, of a fast algorithm for generating the lexicographic construction.

3.2.1 Equivalence of Lexicodes and Lexicographic Construction

We will prove an equivalence between lexicodes and the lexicographic construction in two parts. In the first part, we present an inductive proof that the codes produced by the lexicographic construction starting at the **0** code are, in fact, the lexicodes. We will use the notation $\mathcal{L}_k^d(\mathbb{C})$ to denote the k'th code with minimum distance d produced by the lexicographic construction starting with the code $\mathbb{C}$; where d is obvious from context, we will abbreviate to $\mathcal{L}_k(\mathbb{C})$, and we will omit $\mathbb{C}$ when we desire the zero code to be the starting code of the construction. In the second part, we show, conversely, that every lexicode can be lexicographically constructed from the **0** code.

Theorem 3.1 *An (n,k,d) code $\mathbb{C}$ is a lexicode* if and only if $\mathbb{C} = \mathcal{L}_k^d$.

Proof We first prove that $\mathcal{L}_k^d$ is always a lexicode, by induction on k for a fixed d. It should be noted that, by construction, a lexicode is uniquely determined by its parameters.

Base Case For $k = 1$ it is clear that $\mathcal{L}_k = \mathcal{L}_1 = \langle 1^d \rangle = \left\{0^d, 1^d\right\}$ where 1^d has the usual meaning of d successive 1's. $\mathcal{L}_1$ is clearly a $(d,1,d)$ lexicode because 1^d is the smallest vector of weight greater than or equal to d in $\mathbb{F}_2^d$, and any other vector in $\mathbb{F}_2^d$ will be of minimum distance less than d from either 0^d or 1^d.

Inductive Hypothesis Assume that $\mathcal{L}_k$ is equal to the (n,k,d) lexicode, $L(n,k,d)$ (or, in context, just L_k).

Induction From the definition of the lexicographic construction, $\mathcal{L}_{k+1}$ has length $n+\rho$, where ρ is the covering radius of $\mathcal{L}_k$, so that it has code parameters$(n+\rho, k+1, d)$.

Consider an $(n+\rho, k', d)$ lexicode $L_{k'}$. Such a code can be constructed by picking lexicographically-earliest vectors in $\mathbb{F}_2^{n+\rho}$. Clearly, $L_k \subset L_{k'}$ because, when we are constructing $L_{k'}$, any vector in $\mathbb{F}_2^n$ will come before any other vector in $\mathbb{F}_2^{n+\rho} - \mathbb{F}_2^n$ so that we will be relegated to first including all vectors in L_k. Consequently, from our inductive assumption, $\mathcal{L}_k \subseteq L_{k'}$. This also implies that $k' \geq k$, but equality cannot happen because $L_{k'}$ will always contain the vector $(1^\rho|x)$ that is not in $\mathcal{L}_k$ (x is the vector of distance ρ from $\mathcal{L}_k$). Thus, we conclude that $k' > k$.

When building $L_{k'}$, after we have included all the vectors of $\mathcal{L}_k$ as stated above, we will next have to include the lexicographically first vector that is of distance d from $\mathcal{L}_k$. Assume, without loss of generality,[2] that we add the vector $v = (u|t)$, where t denotes the n least significant bits of v and has distance w from $\mathcal{L}_k$, and u denotes the remaining bits of v and will have distance $d-w$ from $\mathcal{L}_k$. Since $\mathcal{L}_k$ has length n, it has only zeroes outside of the first n bits, implying that v will have $d-w$ ones after the first n bits. Table 3.2 shows the relationship between these variables graphically.

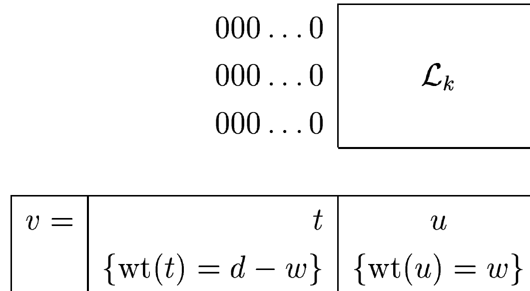

Table 3.2: A graphical diagram relating to the inductive proof of Theorem 3.1. v is shown with the corresponding weights of its two sub-vectors.

Since the definition of lexicodes requires us to pick the lexicographically earliest v to add, we must pick $u = 1^{(d-\max(w))}$, as any other u will lead to a lexicographically later vector $v = (u|t)$, no matter what t is. However, by definition of covering radius, $\max(w) = \rho$ so that we will pick $u = 1^{(d-\rho)}$. Then, given this u, we pick

[2] We have shown above that such a vector must always exist

the lexicographically earliest t of distance ρ from L_k to maintain the definition of lexicodes. One can see that we have added the same vector v as would be added by the lexicographic construction. Since lexicodes are known to be linear [9, 6], it must be that $L_{k'}$ will include the entire subspace spanned by L_k and this new vector v, so that $\mathcal{L}_{k+1} \subset L_{k'}$.

However, any vector $v' \in \mathbb{F}_2^{n+\rho}$ that we would add to $\mathcal{L}_{k+1}$ would have its n least significant bits of distance less than or equal to ρ from $\mathcal{L}_{k+1}$, and its other bits will of distance less than or equal to $\lceil (d-\rho)/2 \rceil$ from $\mathcal{L}_{k+1}$. Since $\rho < d$ (by definition, we add vectors to the lexicode until this condition is fulfilled), we may apply the triangle inequality to get that v' would be of distance less than $\rho + \lceil (d-\rho)/2 \rceil = \lceil (d+\rho)/2 \rceil < d$ from $\mathcal{L}_{k+1}$ so that it could not be added to $\mathcal{L}_{k+1}$ without violating the definition of lexicodes. Thus, it must be that $\mathcal{L}_{k+1} = L_{k'}$.

◇

For the second part of the proof, assume that we have an (n, k, d) lexicode $L(n, k, d)$. Suppose that $\mathcal{L}_k^d$ has length n'. From the first part of the proof, we know that $\mathcal{L}_k^d$ is a lexicode. Thus, if $n' > n$, then $L(n, k, d) \subset \mathcal{L}_k^d$ and, otherwise, $\mathcal{L}_k^d \subset L(n, k, d)$ by the same rationale as in the first part of the proof. Either way, since $\mathcal{L}_k^d$ and L both have dimension k, it follows that they must be equal [1, p. 93].

□

Thus, we have shown the equivalence of lexicodes and the lexicographic construction. Because of this equivalence, we will concern ourselves mostly with the lexicographic construction, which is easier to handle, and apply the results to lexicodes.

3.2.2 Relation among Subsequent Lexicodes

The relationship between the coset leaders of subsequent codes in the lexicographic construction will be the basis of the improved performance of Algorithm 4.1 on page 46, Chapter 4. Upon further research, we have already discovered that Brualdi and Pless [6, Thm 2.2] have noticed this same relationship, but in a different form which is less conducive to our results. Before we establish this relationship, however, we must introduce some terminology.

Definition 3.1 *Given:*

- *a linear code $\mathbb{C}$ with parameters (n, k, d)*
- *a coset leader l for a coset of $\mathbb{C}$*
- *a vector $v \in \mathbf{F}_2^n$*

the buddy *of l is the coset leader of the coset containing $l+v$, and is denoted $\beta_v(l)$ or just $\beta(l)$ when v is clear from context. We will also denote the coset containing $l+v$ with the notation $\kappa(l+v)$.*

As an example, Table 3.3 shows the coset leaders and buddies for the $(4,1,4)$ lexicode generated by $\langle 1111 \rangle$ together with the vector $v = 0101$. The bold buddies denote the instances when $l+v$ is not itself a coset leader and some computation is required to determine $\beta(l)$. We can also see from Table 3.3 that the coset buddies are just a permutation of the coset leaders. This is, in fact, true in general, as the following lemma will show.

Coset leader l	$l+v$	$\beta(l)$
0000	0101	0101
0001	0100	0100
0010	0111	**1000**
0011	0110	0110
0100	0001	0001
0101	0000	0000
0110	0011	0011
1000	1101	**0010**

Table 3.3: Coset leaders and buddies for the $(4,1,4)$ lexicode. The $(4,1,4)$ lexicode is generated by $\langle 1111 \rangle$

Lemma 3.2.1 *For a code $\mathbb{C}$ with a fixed set of coset leaders $\mathcal{S}$,*

$$\mathcal{S}' \triangleq \{\beta(l) : l \in \mathcal{S}\} = \mathcal{S}.$$

Proof: We show that $\beta(\cdot)$ is a one-to-one correspondence.

Indeed, if $l, l' \in \mathcal{S}$ and $\beta(l) = \beta(l')$, then, by definition, $l+v$ and $l'+v$ must be in the same coset, implying (by linearity) that l and l' are in the same coset, so

that $l = l'$ since our coset leaders are fixed for each coset. This shows that $\beta(\cdot)$ is an injection.

It is obvious that $\beta(\cdot)$ is a surjection from the definition of $\mathcal{S}'$. Hence, $\beta(\cdot)$ is a one-to-one mapping and the lemma follows.

□

Now we have the background for the main theorem of this section, upon which our lexicographic construction algorithm in Chapter 4 will be based. This theorem can also be seen as a generalization of Lemma 3.2.1. Note that the notation $\bar{a}$ is used to denote the binary complement of a (i.e. $\bar{a} = 1^{|a|} + a$).

Theorem 3.2 *Consider an (n,k,d) code with a fixed set of coset leaders $\mathcal{S}$, and the code $\mathbb{C}'$ obtained by adding a generator $g = (1^{\Delta}|v)$ for $v \in \mathbf{F}_2^n$ and $\Delta \in \mathbb{Z}$. Then the set $\mathcal{S}'$ containing the coset leaders of $\mathbb{C}'$ can be obtained from $\mathcal{S}$ according to the following bijective mapping:*

$$f : l \in \mathcal{S} \longmapsto l' \in \mathcal{S}' \text{ if, for any } a \in \mathbb{F}_2^{\Delta},$$

$$l' = \begin{cases} a|l & \text{if } \mathrm{wt}(a|l) \leq \mathrm{wt}\left(\bar{a}|\beta_v(l)\right), \\ \bar{a}|\beta_v(l) & \text{if } \mathrm{wt}(a|l) > \mathrm{wt}\left(\bar{a}|\beta_v(l)\right). \end{cases} \tag{3.1}$$

Table 3.4 shows a direct correspondence between some of the vectors $l' \in \mathcal{S}'$ and the coset leaders in the $(6,2,4)$ lexicode (s.f.r. Table 3.3) to aid with the intuition of this proof.

Proof: For reasons of simplification, let us denote

$$\mathrm{f}(a,l) \hat{=} (a \mid l) \text{ and } \mathrm{h}(a,l) \hat{=} (\bar{a} \mid \beta(l)) \tag{3.2}$$

This proof rests upon two observations.

The *first observation* is that $\mathrm{f}(a,l)$ and $\mathrm{h}(a,l)$ are always in the same coset in the new code $\mathbb{C}'$. This is clear because:

$$\begin{aligned} \mathrm{f}(a,l) + \mathrm{h}(a,l) &= a + \bar{a} \mid [l + \beta(l)] \\ &= 1^{\Delta} \mid [l + (l + v + c)], \text{ for some } c \in \mathbb{C} \\ &= \left(1^{\Delta} \mid v\right) + \left(0^{\Delta} \mid c\right) \end{aligned}$$

$a \mid l$	$\bar{a} \mid \beta_v(l)$	*corresponding* $l' \in \mathcal{S}'$
00 0000	11 0101	00 0000
01 0000	10 0101	01 0000
10 0000	01 0101	10 0000
11 0000	00 0101	11 0000
00 0001	11 0100	00 0001
01 0001	10 0100	01 0001
10 0001	01 0100	10 0001
11 0001	00 0100	00 0100
00 0010	11 1000	00 0010
01 0010	10 1000	01 0010
10 0010	01 1000	10 0010
11 0010	00 1000	00 1000
00 0011	11 0110	00 0011
01 0011	10 0110	01 0011
10 0011	01 0110	10 0011
11 0011	00 0110	00 0110
$\ddots$	$\ddots$	$\ddots$

Table 3.4: A partial table of coset leaders of the $(6,2,4)$ lexicode. It is derived from the (4,1,4) lexicode, using Theorem 3.2 with $v = 0101$ and $\Delta = 2$. Spaces between bits are used to highlight which vectors are being concatenated.

Since $(0^\Delta \mid c) \in (0^\Delta|\mathbb{C})$ and $g = (1^\Delta|v)$, it follows that $\mathrm{f}(a,l)+\mathrm{h}(a,l)$ is an element of $\big(g + (0^\Delta \mid \mathbb{C})\big) \subset \mathbb{C}'$ implying that $\mathrm{f}(a,l)$ and $\mathrm{h}(a,l)$ are in the same coset in $\mathbb{C}'$.

The *second observation* on which this proof is based is that, for any two different coset leaders l and l' of $\mathbb{C}$ and for all $a, a' \in \mathbb{F}_2^\Delta$, $\mathrm{f}(a,l)$ and $\mathrm{f}(a',l')$ are not in the same coset. To prove this observation we need consider two cases.

If $\mathbf{a} \neq \mathbf{a}'$ then the observation is trivial because $\mathrm{f}(a,l) + \mathrm{f}(a',l')$ will have a leading bits that are different from 0^Δ and 1^Δ, but all the codewords in $\mathbb{C}'$ must have leading bits 0^Δ or 1^Δ (the leading bits are determined by the added generator $g = 1^\Delta \mid v$).

If a = a′ then $f(a,l) + f(a',l') = 0^\Delta \mid l + l'$. Since $l + l'$ was given to be not in $\mathbb{C}$, $f(a,l)$ and $f(a',l')$ are not in the same coset. This is because all codewords that are in $\mathbb{C}'$ but not in $\mathbb{C}$ must start with 1^Δ.

Based on the above two observations, we see that f and h represent the same one-to-one correspondence between pairs (a,l) and the cosets of $\mathbb{C}'$. Thus, in order to determine the coset leaders of $\mathbb{C}'$, we are left merely with the task of determining this correspondence between pairs (a,l) and the coset leaders of the corresponding code.

By inspection, we see that for each $a \in \mathbb{F}_2^n$ and coset leader l of $\mathbb{C}$, $\kappa\left(f(a,l)\right)$ is comprised of two types of vectors:

- those of the form $(a \mid c)$, for c in the same coset as l
- and those of the form $(\bar{a} \mid (v + c'))$, for $v + c'$ in the same coset as $\beta(l)$

In addition, $f(a,l)$ cannot have weight greater than the weight of $(a \mid c)$ because $\mathrm{wt}(a \mid c) - \mathrm{wt}(f(a,l)) = \mathrm{wt}(c) - \mathrm{wt}(l) \geq 0$ as l is a coset leader of the $\kappa(c)$.

Similarly, $h(a,l)$ cannot have weight greater than $(\bar{a} \mid (v + c'))$ because $\mathrm{wt}\left(\bar{a} \mid (v + c')\right) - \mathrm{wt}(h(a,l)) = \mathrm{wt}(v + c') - \mathrm{wt}(\beta(l)) \geq 0$ because $\beta(l)$ is the coset leader of $\kappa(v + c')$.

Thus, all the vectors in $\kappa\left(f(a,l)\right)$ will have weight not greater than $\min\{\mathrm{wt}(f(a,l)), \mathrm{wt}(h(a,l))\}$ so that we may arbitrarily choose as a coset leader that vector $f(a,l)$ or $h(a,l)$ which has minimum weight. Moreover, since this mapping gives us a $\|\mathbb{F}_2^\Delta\| = 2^\Delta$-fold increase in the number of coset leaders, which corresponds to the total number of coset leaders $\mathbb{C}'$ should have, this mapping must, in fact, give us all the coset leaders of $\mathbb{C}'$, and the theorem is proved.

$\square$

3.3 Bounds on Code Parameters

Theorem 3.2 in section 3.2, together with Lemma 3.2.1, give us a description of a the coset leaders of $\mathcal{L}_m$ in terms of its the coset leaders of its predecessor in the lexicographic construction, $\mathcal{L}_{m-1}$. The maximum weight coset leader of $\mathcal{L}_m$, in turn, determines the covering radius of a lexicode and, hence, the code parameters of the subsequent lexicode in the construction.

However, the maximum weight coset leader actually depends on the permutation $l \longmapsto \beta(l)$ of coset leaders, as in Lemma 3.2.1, that a lexicode imposes on its predecessor. In this section we will consider some counting arguments that bound the maximum weight coset leader in each subsequent code, and, thus, impose a bound on the parameters of lexicodes. Derivation of these bounds will also give us a closed-form expression for the parameters of all lexicodes of dimensions 1, 2, and 3.

First we derive a simple upper bound on the covering radius, ρ_m, of $\mathcal{L}_m^d$.

Lemma 3.3.1 $\rho_m \le \left\lfloor \frac{d+\rho_{m-1}}{2} \right\rfloor$

Proof: Consider the $m-1$'th lexicode $\mathcal{L}_{m-1}$, and its coset leaders, being used to construct the m'th lexicode. Given any coset leader l of $\mathcal{L}_{m-1}$, $\beta_v(l)$ is also a coset leader of $\mathcal{L}_{m-1}$. Thus, in such a case, both l and $\beta(l)$ must have weight at most ρ_{m-1}. This, in turn, means that, for all coset leaders l of $\mathbb{C}$ and for all $a \in \mathbb{F}_2^{\Delta}$ (where $\Delta = d - \rho_{m-1}$ as per the lexicographic construction) we have:

$$\begin{aligned} &\min\{\mathrm{wt}(a \mid l), \mathrm{wt}\,(\bar{a} \mid \beta_v(l))\} \\ &\qquad \le \min\{\mathrm{wt}(a) + \rho_{m-1}, \mathrm{wt}(\bar{a}) + \rho_{m-1}\} \\ &\qquad \le \lfloor \Delta/2 \rfloor + \rho_{m-1} \\ &\qquad\qquad \{max.isata = 0^{\lceil \Delta/2 \rceil} 1^{\lfloor \Delta/2 \rfloor}\} \end{aligned}$$

From Theorem 3.2 we know that each coset leader l' of $\mathcal{L}_m$ has weight not greater than $\lfloor \Delta/2 \rfloor + \rho_{m-1} = \left\lfloor \frac{d-\rho_{m-1}}{2} \right\rfloor + \rho_{m-1} = \left\lfloor \frac{d+\rho_{m-1}}{2} \right\rfloor$ and the lemma is proved.

□

Clearly Lemma 3.3.1 implies that $\rho_m < d$ since $\rho_0 = 0$, so that $\mathcal{L}_m$ cannot be trivially improved with the addition of another codeword at distance d, a fact that we also saw in Theorem 3.1.

Now that we have a simple upper bound on ρ_m, we also develop a lower bound on ρ_m by considering the worst possible permutation between coset leaders and buddies. We use the notation n_m to denote the length of the m'th lexicode.

Theorem 3.3 *For $m \ge 3$,*

$$\rho_m \ge \left\lfloor \frac{d-1}{2} \right\rfloor + \left\lfloor \frac{d-\rho_{m-1}}{2} \right\rfloor \tag{3.3}$$

In the case of $m = 1$, $\mathcal{L}_m$ consists only of the basis vector $\langle 1^d \rangle$, so that

$$\rho_1 = \lfloor d/2 \rfloor \tag{3.4}$$

In the case of $m = 2$, $\mathcal{L}_m$ consists of the basis vectors $\langle 1^d, \left(1^{\lceil d/2 \rceil} \mid 0^{\lceil d/2 \rceil} \mid 1^{\lfloor d/2 \rfloor}\right)\rangle$. By careful inspection, we can see that $\rho_2 = \rho_1 + \left\lfloor \left\lceil \frac{d}{2} \right\rceil /2 \right\rfloor$, which is attained by a vector that is of distance ρ_1 from $\mathcal{L}_1$ in its first d bits,[3] and of distance $\left\lfloor \left\lceil \frac{d}{2} \right\rceil /2 \right\rfloor$ from the remaining $\left\lceil \frac{d}{2} \right\rceil$ bits. Thus,

$$\rho_2 = \lfloor d/2 \rfloor + \left\lfloor \frac{\lceil d/2 \rceil}{2} \right\rfloor \tag{3.5}$$

Since the parameters of a lexicode are determined by the covering radius of the preceding codes in the lexicographic construction, we may combine equations 3.4 and 3.5 to give a closed form for the parameters of lexicodes of dimensions 1, 2, and 3:

Dimension	Parameters
1	$(d, 1, d)$
2	$(d + \lceil d/2 \rceil, 2, d)$
3	$\left(d + 2\lceil d/2 \rceil - \left\lfloor \frac{\lceil d/2 \rceil}{2} \right\rfloor, 3, d\right)$

Figure 3.2 plots the length of the dimension 3 lexicodes versus their minimum distance. All the lengths plotted seem to be those of optimal linear codes.

Before we continue with the proof, we introduce a convenient combinatorial lemma.

Lemma 3.3.2 *If $a > 3b > 0$ then*

$$\binom{a}{b} > \sum_{i=0}^{b-1} \binom{a}{i} \tag{3.6}$$

---

[3] One such vector would be $000\ldots0111\ldots101$, but any similar type of permutation will do, as long as it does not complement the less significant bits of the second generator of $\mathcal{L}_2$.

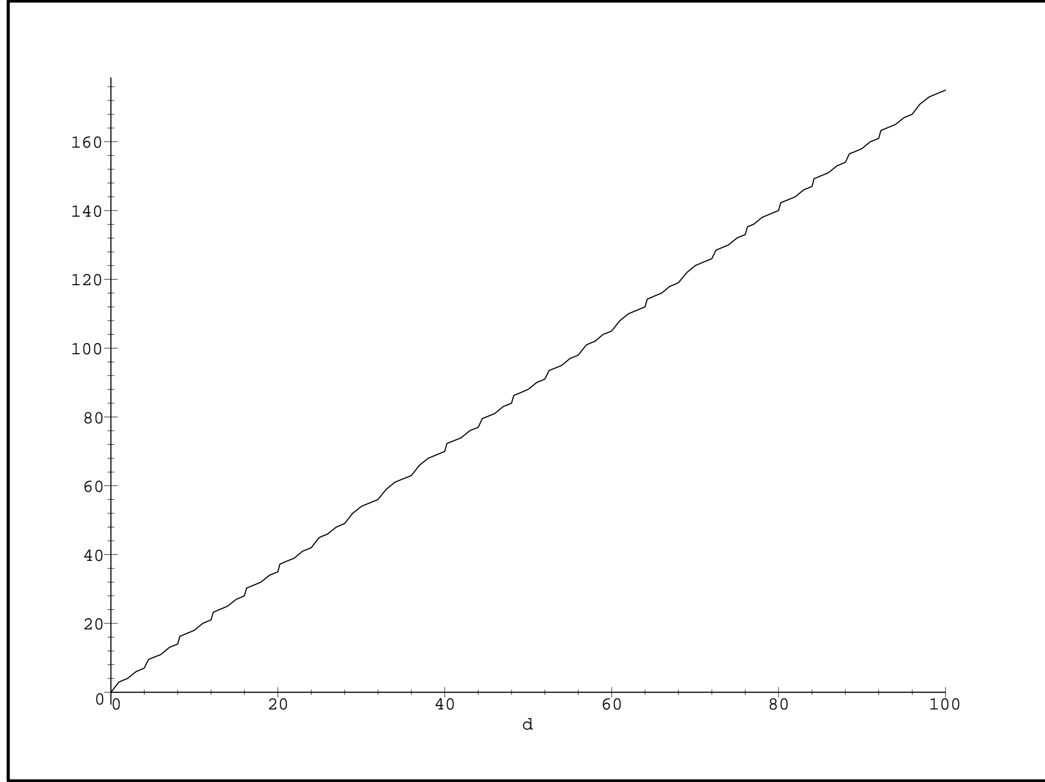

Figure 3.2: The length of the dimension 3 lexicodes plotted versus their minimum distance d. The slope of the line tends towards $\frac{7}{4} = 1.75$.

Proof It is well known (s.f.r. [7, p. 122]) for binomial distributions that, for $0 < b < ap$:

$$\sum_{i=0}^{b-1} \binom{a}{i} p^i (1-p)^{a-i} < \frac{b(1-p)}{ap-b} \binom{a}{b} p^b (1-p)^{a-b} \tag{3.7}$$

Setting $p = 1/2$ in equation (3.7) gives us the following relation:

$$\binom{a}{b} > \left(\frac{a}{b} - 2\right) \sum_{i=0}^{b-1} \binom{a}{b}$$

Since we assumed $a > 3b$, we have that $\frac{a}{b} - 2 > 1$ and the lemma is proved.

□

We now have the necessary tools for the proof of the theorem.

Proof of Theorem 3.3: Since we always have $\rho_0 = 0$ and $\rho_1 = \left\lfloor \frac{d}{2} \right\rfloor$ (from equation (3.4)), with corresponding $n_0 = 0$, $n_1 = d$, $n_2 = d + \left\lceil \frac{d}{2} \right\rceil = \left\lceil \frac{3d}{2} \right\rceil$, it is sufficient for us to consider only those cases where $n_m > n_2 = \left\lceil \frac{3d}{2} \right\rceil$, where the theorem is not trivial.

Now, from basic coding theory, we know that each vector in $\mathbb{F}_2^{n_{m-1}}$ of weight $\leq t \triangleq \left\lfloor \frac{d-1}{2} \right\rfloor$ must be a unique coset leader for $\mathcal{L}_{m-1}$. Furthermore, there are $\binom{n_{m-1}}{t}$ vectors of weight t, and $\sum_{i=0}^{t-1} \binom{n_{m-1}}{i}$ vectors of weight $< t$ in $\mathcal{L}_{m-1}$.

We now apply Lemma 3.3.2 under the substitutions $a \longrightarrow n_{m-1}$ and $b \longrightarrow t$ to get:

$$\binom{n_{m-1}}{t} > \sum_{i=0}^{t-1} \binom{n_{m-1}}{i}$$

Thus, by the pigeon-hole principle applied to Lemma 3.2.1, there must be at least one coset leader $l \in \mathcal{L}_{m-1}$ with weight t, whose buddy, $\beta(l)$ also has weight at least t. For this pair, we may pick $a = 0^{\lfloor \Delta/2 \rfloor} 1^{\lceil \Delta/2 \rceil}$ (where, in this case, $\Delta = d - \rho_{m-1}$), to get:

$$\begin{aligned} \rho_m &\geq \min\left\{ wt(a \mid l), \mathrm{wt}\left(\bar{a} \mid \beta_v(l)\right) \right\} \\ &= \left\lfloor \frac{d - \rho_{m-1}}{2} \right\rfloor + t \\ &= \left\lfloor \frac{d - \rho_{m-1}}{2} \right\rfloor + \left\lfloor \frac{d-1}{2} \right\rfloor \end{aligned}$$

□

Inequality (3.3) is a bit unwieldy, so it is helpful to prove some corollaries that better explain its usefulness.

The following corollary is an obvious simplification of inequality (3.3):

Corollary 3.3.1

$$\begin{aligned} \rho_m &\geq \left\lfloor \frac{d-1}{2} \right\rfloor + \frac{d}{2} - \frac{\rho_{m-1}}{2} - 1 \qquad (3.8) \\ &\geq d - \frac{\rho_{m-1}}{2} - 2 \qquad (3.9) \end{aligned}$$

We may also use Theorem 3.3 to attain a bound on the lengths of lexicodes.

Corollary 3.3.2

$$n_k \leq \frac{kd}{3} + \frac{4}{3} \qquad (3.10)$$

Proof: Consider summing inequality (3.9) over all $m = 1 \dots k$:

$$\sum_{m=1}^{k} \rho_m \geq \sum_{m=1}^{k} \left(d - \frac{\rho_{m-1}}{2} - 2\right)$$
$$\geq k(d-2) - \frac{1}{2} \sum_{m=0}^{k-1} \rho_m$$

Noting that $\rho_0 = 0$ and $\rho_k \geq 0$, we may bring the right-hand summation over to the left hand side to get:

$$\frac{3}{2} \sum_{m=1}^{k-1} \rho_m \geq k(d-2) - \rho_0 + \rho_k \tag{3.11}$$

Furthermore, according to the lexicographic construction,

$$n_k = \sum_{i=0}^{k} (d - \rho_i) = kd - \sum_{i=1}^{k} \rho_i \tag{3.12}$$

so that we may now conclude the proof:

$$n_k \leq kd - \sum_{i=1}^{k} \rho_i$$
$$\leq kd - \frac{2}{3} k(d-2)$$
$$\leq \frac{k}{3}(d+4)$$

□

Note that, in the case that d is odd, $\left\lfloor \frac{d-1}{2} \right\rfloor = \frac{d-1}{2}$ so that we can attain a trivial improvement:

$$n_k \leq \frac{k}{3}(d+1)$$

Also, we may rewrite Corollary 3.3.2 in terms of the dimension:

$$k \geq \frac{3n}{d+1} \tag{3.13}$$

and see that this bound is asymptotically better than that of Theorem 3.5 in [6]:

$$k \geq \begin{cases} n - 2 - \lfloor \log_2(n-1) \rfloor & \text{if } d = 4, \\ \left\lfloor \frac{4n-d-12}{2d-4} \right\rfloor & \text{if } d \equiv 0 \pmod 4, d \neq 4, 8, \\ \left\lfloor \frac{n}{3} \right\rfloor, & \text{if } d = 8,\ n > 18, \\ \left\lfloor \frac{4n-d-14}{2d-4} \right\rfloor & \text{if } d \equiv 2 \pmod 4. \end{cases} \tag{3.14}$$

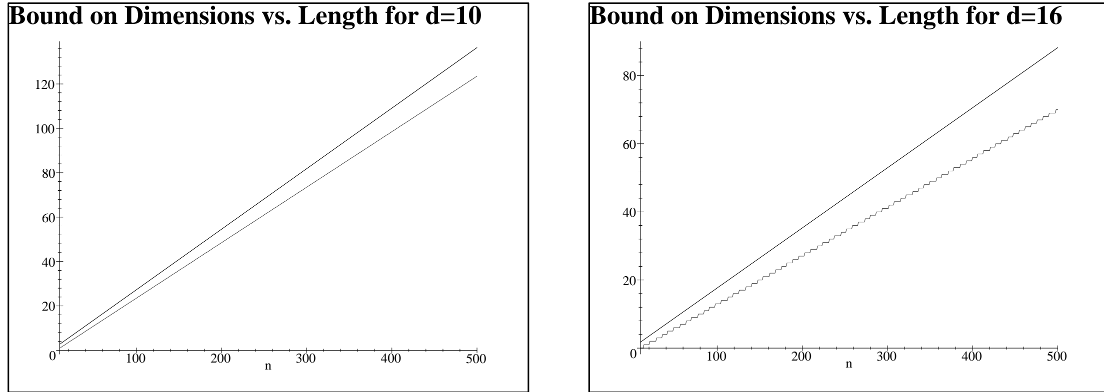

Figure 3.3: Lower bounds on lexicode dimension. The higher line depicts the bound in Corollary 3.3.2 and the lower line depicts the bound 3.14 for the representative distances $d = 10$ and $d = 16$. Clearly the bound in Corollary 3.3.2 is better.

Bound (3.10) asymptotically binds $k \geq 3\frac{n}{d}$ whereas Bound (3.14) binds $k \geq 2\frac{n}{d}$. In fact, a close analysis of the two inequalities shows that Bound (3.10) is, overall, better than Bound 3.14, and this improvement can be seen graphically in Figure 3.3.

Thus, we have attained both upper and lower bounds for the parameters of lexicographic codes based on the analysis of the permutation dictated by Lemma 3.2.1. In fact, it should not be hard to generalize this bound to non-binary lexicographic codes. Also it should not be difficult to improve the bound with a more careful analysis that takes into consideration the structure of the permutation imposed by adding a new vector to the code.

Chapter 4

Generalized Lexicographic Construction

In Chapter 3 we showed that the lexicographic construction can be used to generate the lexicodes. However, many of the theorems and lemmas we proved in association with the lexicographic construction (especially Theorem 3.2) are not limited to the lexicodes and can be incorporated as part of a Generalized Lexicographic Construction (GLC), which we rigorously define and explain in Section 4.1 along with several other important definitions. In fact, it has come to our attention since the completion of this work that the GLC codes seem to produce the $\mathcal{B}$-greedy codes described in [6].

In Section 4.2 we analyze how to compute three families of codes that result from various restrictions of the GLC: standard lexicodes, trellis-oriented GLC codes, and locally optimal GLC codes. The trellis-oriented GLC codes are of particular interest because they are (locally) optimized to reduce decoding complexity.

In Section 4.3 we develop a relatively fast algorithm that runs in time and space $O(2^{n-k+1})$ (where n and k are the length and dimension of the code respectively), which computes subsequent codes in the GLC for the families of codes described in Section 4.2. In the specific case of $d = 4$, based on the work in [6], this algorithm actually runs in linear time. Again, we have discovered that this algorithm bares an uncanny resemblance to the parity-check matrix generator found in [6, p.16].

In Section 4.4 we show how the algorithm in Section 4.3 can be easily modified to compute GLC codes with a bounded trellis state complexity, which are important for real implementations with physical constraints.

In Section 4.5 we compile some results of computations carried out using the algorithms developed in this chapter. We extend the table of constructed lexicodes found in [9], demonstrate the improvements of trellis-oriented GLC codes, and construct codes with various trellis-state complexity bounds. In addition, we show how trellis-oriented GLC codes can also improve on the greedy algorithm of [24].

4.1 Terminology

We begin with a definition of the Generalized Lexicographic Construction.

Definition 4.1 *Given:*

- *an* (n, k, d) *code* $\mathbb{C}$*;*
- $r(\cdot)$ *mapping each code to an integer between* 0 *and the code's covering radius;*
- $w(\cdot)$ *mapping each code* $\mathbb{C}'$ *to a vector of Hamming distance* $r(\mathbb{C}')$ *from the code.*

The Generalized Lexicographic Construction over r and w is the family of codes

$$\left\{\mathrm{GLC}^i\,(r, w, \mathbb{C}) : i \in \mathbb{Z}\right\}$$

where $\mathrm{GLC}^i\,(r, w, \mathbb{C})$ (or, GLC^i in context) is a function that returns a linear code as follows:

- GLC^0 trivially returns the code $\mathbb{C}$;
- GLC^i returns the code obtained by adding the generator
$$v = \left(1^{n-r(\mathrm{GLC}^{i-1})} \quad | \quad w(\mathrm{GLC}^{i-1})\right)$$
to the code GLC^{i-1}.

We may think of $r(\cdot)$ as the effective covering radius of the code, and $w(\cdot)$ is a vector that is as far as possible from the code within the effective covering radius. In context, we may omit any of the parameters of $\mathrm{GLC}^i\,(r, w, \mathbb{C})$ for sake of clarity.

The *GLC* is quite general, and, in order to make it more understandable, we present the following examples.

Example 6 *Consider the following functions* r_l *and* w_l*:*

$$r_l(\mathbb{C}) = \textit{the covering radius of } \mathbb{C} \tag{4.1}$$

$$w_l(\mathbb{C}) = \textit{the lexicographically earliest vector of} \tag{4.2}$$

$$\textit{distance } r(\mathbb{C}) \textit{ from } \mathbb{C} \tag{4.3}$$

One can readily recognize, for the seed code $\mathbb{S} = \{0^d, 1^d\}$*, that* $GLC^i\,(r_l, w_l, \mathbb{S})$ *is precisely the lexicographic construction which generates the lexicodes, so that* $GLC^i\,(r_l, w_l, \mathbb{S}) = \mathcal{L}_i^d$.

A simpler example is:

Example 7 *Consider the following inputs:*

$$r_e(\mathbb{C}) = 0$$
$$w_e(\mathbb{C}) = 1^n \text{ where } n \text{ is the length of } \mathbb{C}$$
$$\mathbb{S} = 000, 111$$

Then $GLC^i(r_t, w_e, \mathbb{S})$ will construct the family of codes generated by generator matrices with the following form:

$$\begin{bmatrix} 111 & 000 & 000 & \dots \\ 111 & 111 & 000 & \dots \\ 111 & 111 & 111 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} = \begin{bmatrix} 111 & 000 & 000 & \dots \\ 000 & 111 & 000 & \dots \\ 000 & 000 & 111 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

In addition to the GLC, we will also make use of the following definitions in this chapter.

We now define three similar ideas:

Definition 4.2 *The* location *of a bit b in a bit sequence is the number of bits in a sequence that are to the right of b, including the bit b.*

The least significant location *(also denoted* lsl*) of a bit sequence is the location of the least significant bit in the sequence, which is the rightmost 1-bit.*

The most significant location *(*msl*) is the leftmost 1-bit in the sequence.*

Thus, $\text{lsl}(11011100) = 3$ whereas $\text{msl}(11011100) = 8$.

Definition 4.3 ([33], p. 28) *The* span *of x, denoted* $\text{Span}(x)$*, is the set:*

$$\{\text{lsl}(x), \text{lsl}(x) + 1, \text{lsl}(x) + 2, \dots, \text{msl}(x)\}$$

Thus, $\text{Span}(11011100) = 3, 4, 5, 6, 7, 8$.

Definition 4.4 ([33], p. 31) *The* Minimum Span Generator Matrix *of a code (hereby denoted* MSGM*) is a generator matrix of the code in which no two generators have either a least significant location or a most significant location in common.*

For example, the generator matrix in Table 3.1 is *not* in *MSGM* form because all the vectors share the same least significant location. However, [33] describes a method that computes an *MSGM* from any given generator matrix and, when applied to Table 3.1, the following *MSGM* for the same code results:

1111
111100
1011010
11110000

Table 4.1: *MSGM* for the $(8,4,4)$ code in Table 3.1.

4.2 Choosing Subsequent Vectors

In this section we analyze the effect of using the r_l function of equation 4.1 on the GLC. In such a case, the main bottleneck becomes the computation of the w function used in the GLC, which will have to compute vectors at the maximum distance from a code (i.e. the covering radius). We develop algorithms that efficiently compute $\mathrm{GLC}^i(r_l, w, \mathbb{C})$ for three different functions w that generate three families of codes with particularly useful properties.

Method 4.2.1 in Subsection 4.2.1 puts forth the basic algorithm for computing w_l; that is to say, it describes the computation of the lexicographically earliest vector of maximal distance from the code. Method 4.2.2 in Subsection 4.2.2, calculates a function w_t so as to generate a family of codes whose generator matrices have locally minimal trellis complexity; we would like to thank G.D. Forney for directing us to this particular avenue of research. We analyze Methods 4.2.1 and 4.2.2 with a few corollaries in Subsection 4.2.3. Finally, Method 4.2.3 in Subsection 4.2.4 computes a function w_o that generates a family of codes that has locally optimal parameters.

4.2.1 Standard Lexicographic Construction

We showed in Example 6 that for the seed code $\mathbb{S} = \{0^d, 1^d\}$, $\mathrm{GLC}^i(r_l, w_l, \mathbb{S})$ generates the lexicodes, but we never showed how to calculate either of the component functions r_l or w_l. We now state a method for calculating w_l for an arbitrary code $\mathbb{C}$ with parameters (n,k,d) and MSGM G, that depends on being given a set of vectors $\mathcal{V} \subset \mathbf{F}_2^n$ of distance ρ from $\mathbb{C}$. Determining this set $\mathcal{V}$ is the bottleneck in computational

implementation of the *GLC*, and it is addressed in Section 4.3 together with the calculation of r_l.

Method 4.2.1 Given a set of vectors $\mathcal{V} \subset \mathbf{F}_2^n$ of distance ρ from $\mathbb{C}$, which is generated by an *MSGM* G, the following method returns the lexicographically earliest vector among the cosets represented in $\mathcal{V}$. In the case that $\mathcal{V}$ contains all the vectors of distance ρ from $\mathbb{C}$, the result of this method is precisely $w_l(\mathbb{C})$.

> **for** $v \in \mathcal{V}$ **do**
>
>   **while** the *msl* of some row G_i of G equals the location of any 1-bit of v
>
>   **do** $v \leftarrow v + G_i$; {step *}
>
> **store** the modified v;
>
> **among** all stored v, return the lexicographically earliest

It is important to note that this method is different from computing the matrix multiplication $v * G$ because of its iterative nature.

Proof of the method: We will prove that, for each $v \in \mathcal{V}$, the **while** loop computes the lexicographically earliest vector in the same coset as v. Thus, the lexicographically earliest vector among all the cosets represented by $\mathcal{V}$ must be among the **store**d vectors, so that the **among** line in the method will return the lexicographically earliest vector among all the cosets represented.

To see that the **while** iteration does as we claim, we must first note that it actually stops at some point (implying that the method halts). This is true because, the *msl* of the modified v (after execution of *step* *) must be to the right of the *msl* of the pre-modified v (before execution of *step* *); we are adding two vectors with the same *msl* in *step* *, so their sum must have its *msl* at a lower location. Since each bit sequence has a finite number of bits n, the **while** iteration may not iterate more than n times.

Now we progress with the meat of the proof, where we assume the method has chosen some $v \in \mathcal{V}$ and intend to show that the **while** loop computes the lexicographically earliest vector in the same coset as v. From [33, Thm 6.11 and Lemma

6.7] we know that the rows of G have the *predictable support property*, meaning that:

$$\text{Span}\left(\sum_{j\in J} G_j\right) = \bigcup_{j\in J} \text{Span}(G_j) \tag{4.4}$$

for every subset $J \subseteq \{1, 2, 3, ..., n\}$. Now suppose v_{best} is the lexicographically earliest vector in the same coset as v, but that v' is the vector **store**d by the method upon exit from the **while** loop. Let us denote the difference between these two vectors, which we wish to be 0, as follows:

$$v_{\text{diff}} \hat{=} (v_{\text{best}} - v') \tag{4.5}$$

One should note that v, v_{best}, and v' are necessarily in the same coset of $\mathbb{C}$ because v and v_{best} are defined so, and v' is derived from v only by adding code vectors (specifically, the generators of G) keeping it in the same coset as v. Moreover, since v and v' are in the same coset, v_{diff} must be a code word of $\mathbb{C}$.

Since $v_{\text{diff}} \in \mathbb{C}$, we can express v_{diff} as the sum over some subset $J_{\text{diff}} \subseteq \{1, 2, 3, ..., n\}$ of the rows of G:

$$v_{\text{diff}} = \sum_{j\in J_{\text{diff}}} G_j \tag{4.6}$$

Then, by the *predictable span property* of G,

$$\text{Span}(v_{\text{diff}}) = \bigcup_{j\in J_{\text{diff}}} \text{Span}(G_j) \tag{4.7}$$

We are now in a position to show that v_{diff} cannot have any significant bits, implying that it must be 0. This, in turn, would imply that $v_{\text{best}} = v'$ which proves the theorem.

Assume (for sake of contradiction) that the most significant location of v_{diff} is k. By equation (4.5), there are only two ways of getting a 1 in this location of v_{diff}: either v_{best} has a 0 and v' a 1 in the k'th location or vice versa. The latter case is clearly impossible under these conditions because it would imply that v' comes lexicographically before v_{best}[1], a contradiction of the definition of v_{best}. Hence, it must be that v_{best} has a 0 and v' a 1 in the k'th location.

Using equation (4.7) we also see that there must be some generator vector G_j whose most significant location is the same the most significant location of v_{diff},

[1] Since k is the most significant location of v_{diff}, locations k and higher of v_{diff} must be 0 implying that v_{best} and v' coincide at those locations. Thus, the bit at location k would determine the lexicographic order of v' and v_{best}.

which is at location k. However, by the previous conclusion we know that v' *also* has a 1 at location k, which is a **contradiction** of the terminating condition of the method.

Thus, our assumption above must have been wrong, and the most significant location of v_{diff} could not possibly be at location k, for any k, so that v_{diff} must be 0. Therefore, we have shown that the **while** iteration returns the lexicographically earliest vector in the same coset as v so that, by the discussion at the beginning of this proof, the method acts as claimed.

□

4.2.2 Trellis-Oriented

The Viterbi algorithm was invented for the purpose of fast decoding of convolutional codes and also linear block codes [2]. Forney [16] introduced trellises to simplify the Viterbi algorithm. In addition, as we can see in [33], the number of edges and vertices in a trellis determine the decoding complexity of the Viterbi algorithm, and the *BCJR* trellis [2] minimizes this decoding complexity.

We can modify Method 4.2.1 to construct a family of codes whose generator matrices have locally minimal trellis complexity. In order to understand what it means for a generator matrix to have locally minimal trellis complexity, we make use of the following relation between the numbers of vertices and edges in the *BCJR* trellis on the one hand, and the form of the code's *MSGM* on the other hand. This theorem is slightly modified from its original to suit the purposes of this paper.

Theorem 4.1 ([33]: Thm's 4.7, 6.1) *Consider using the* BCJR *trellis to represent a code of dimension k and Minimum Span Generator Matrix G. Then, for this trellis, the number of vertices at depth i, $|V_i|$, and the number of edges linking depths i and $i+1$, $|E_{i,i+1}|$, are given by the following formulae:*

$$|V_i| = 2^{k-p_i-f_i}$$
$$|E_{i,i+1}| = 2^{k-p_i-f_i+1}$$

where,

$$p_i = |\,\{j : \mathrm{lsl}(G_j) \leq i\}\,|$$
$$f_i = |\,\{j : \mathrm{msl}(G_j) \geq i+1\}\,|$$

Thus, given a code $\mathbb{C}$ to which we would like to add a codeword c using the $GLC(r_l)$, we note that $\text{msl}(c)$ is fixed by the covering radius of $\mathbb{C}$. Thus, in order to minimize decoding complexity on a local scale, we must pick c to have the smallest possible span, which in this case means that $\text{lsl}(c)$ will be as high as possible. If we pick c as the lexicographically latest vector, we are assured that it will minimize trellis complexity, though not necessarily uniquely. Note also that we actually have to pick the lexicographically latest *vector*, and not just the coset leader in which it resides or the lexicographically latest coset leader, in order to get the minimum span desired. Thus, $\text{GLC}^i\,(r_l, w_t, \mathbb{C})$ has locally minimal decoding complexity if w_t is defined as follows:

$$w_t = \text{the lexicographically } \textit{latest} \text{ vector of distance } r_l(\mathbb{C}) \text{ from } \mathbb{C} \qquad (4.8)$$

In fact, Method 4.2.1 can be easily modified to compute w_t.

Method 4.2.2 Given a set of vectors $\mathcal{V} \subset \mathbf{F}_2^n$ of distance ρ from a code $\mathbb{C}$ which is generated by an *MSGM* G, the following method returns the lexicographically latest vector among the cosets represented in $\mathcal{V}$.

for $v \in \mathcal{V}$ **do**

  while the lsl of some row G_i of G equals the location of any 1-bit of v

    do $v \leftarrow v + G_i$;

store the modified v;

among all stored v, return the lexicographically earliest

Notice that the only difference between this method and Method 4.2.1 is that we are using least significant locations rather than most significant locations in the **while** condition.

Proof of the method: The proof of this method is very closely linked to the proof of Method 4.2.1. Let us modify the definition of the *location* of a bit as so:

Definition 4.5 *The* reverse location *a bit b in a bit sequence is the number of bits in a sequence that are to the* left *of b, including the bit b.*

Now, we can just replace "location" in the proof of method 4.2.1 with "reverse location" and method 4.2.2 is proved.

□

4.2.3 Corollaries

We now analyze Methods 4.2.1 and 4.2.2 by means of a few corollaries that relate their effectiveness.

Corollary 4.2.1 *If the codeword c returned by method 4.2.2 is added to the* MSGM *G, the new generator matrix formed is also in an* MSGM.

That no two codewords share a most significant location is evident from their construction method. That no two codewords share a least significant location derives from the fact that **while** loop in method 4.2.2 is actually just a specific version of the greedy method described in [33, p. 32] for finding a minimal span generator matrix.

It is also of some interest to note that both method 4.2.1 and method 4.2.2 run in relatively small time and space.

Corollary 4.2.2 *For an input set $\mathcal{V}$ of cardinality $|V|$ containing codewords drawn from a k-dimensional code, methods 4.2.1 and 4.2.2 require time $O(k|V|)$ and space $O(|V|)$.*

The space bound is clear because exactly one lexicographically earliest vector is stored for each vector $v \in \mathcal{V}$, and determination of the lexicographically earliest vector can be done in place. The time bound derives from the remark at the beginning of the proof of method 4.2.1. Specifically, the **while** iteration in these methods takes, at most, time k (one iteration for each row of G) if done in left-to-right order. The **for** iteration iterates the **while** clause $|V|$ times. Finally, the **among** line is merely a search for minimality, which naively requires time $O(|V|)$. Thus, the overall time is $O(k) * |V| + O(|V|) \subset O(k|V|)$.

The Methods 4.2.1 and 4.2.2 will be applied, in Section 4.3, to the set of coset leaders that are of distance ρ from the code $\mathbb{C}$. In this way, they will determine how to extend a code both by the standard and by the trellis-oriented version of the lexicographic construction.

4.2.4 Locally Optimal Codes

Given a code $\mathbb{C}$, Theorem 3.2 describes all the coset leaders of the subsequent code in the Generalized Lexicographic Construction, GLC[1] $(r_l, w, \mathbb{C})$. This, in turn, enables

us to pick $\mathbb{C}$ in such a manner as to maximize the covering radius of GLC^1, so that GLC^2 would have the best possible parameters achievable among all functions w used in the *GLC*. In other words, GLC^2 will be locally optimal over r_l. This idea can be extended to any size of locality (i.e. to make GLC^m optimal), and the respective method follows.

Method 4.2.3 Given any set $\mathcal{V} \subset \mathbf{F}_2^n$ of all vectors of distance ρ from $\mathbb{C}$ together with the coset leaders of $\mathbb{C}$, the following method returns the vector v such that there exists some function w_o, with $w_o(\mathbb{C}) = v$ that produces the optimal length, dimension, and covering radius for $\mathrm{GLC}^m(w_o)$. Applied iteratively, this method will, indeed, determine the value of this function w_o on all subsequent codes in the *GLC*.

`Temporary Code` $\longleftarrow \mathcal{L}_m$;

if $i > 1$ **then**

for each $v \in \mathcal{V}$ **do**

add v to the `Temporary Code` and **decrement** i.

compute all the coset leaders in `Temporary Code` by using Theorem 3.2, and determine those coset leaders of distance ρ from the code;

recursively run method 4.2.3 on `Temporary Code` and **store** the returned covering radius if it s bigger than any previously returned covering radius;

delete v from `Temporary Code` and **increment** i.

return the **store**d covering radius;

else **for** each $v \in \mathcal{V}$ **do**

add v to the `Temporary Code`.

compute all the coset leaders of `Temporary Code` using the Theorem 3.2 and store a leader of greatest weight

return the leader of greatest weight and its corresponding weight (which is the covering radius of the generated code)

Proof: This method is really just an exhaustive search over all maximum-distance coset leaders by means of Theorem 3.2. Though Theorem 3.2 greatly simplifies the method over a straight-forward brute force approach, method 4.2.3 is, nevertheless, unwieldy for large i or d.

4.3 The Algorithm

We may now combine the results of Section 4.2 and Section 3.2 in Chapter 3 to describe an algorithm for computing all aspects of the GLC for the standard lexicographic construction (i.e. $\mathrm{GLC}^i\left(r_l, w_l, \mathbb{C}\right)$), the trellis-oriented GLC codes (i.e. $\mathrm{GLC}^i\left(r_l, w_t, \mathbb{C}\right)$), and the locally optimal GLC codes (i.e. $\mathrm{GLC}^i\left(r_l, w_o, \mathbb{C}\right)$).

Algorithm 4.1 One may generate $GLC^i\left(r_l, w, \mathbb{C}\right)$ for w being w_l, w_t, or w_o by using the following algorithm. We assume that $\mathbb{C}$ has minimum distance d and that its coset leaders and their respective syndromes are given.

1. GLC^0 is trivially the code $\mathbb{C}$; we store its cosets and their syndromes, as is given in the input to the algorithm, in the array `COSETS`;
2. Set v to some maximum weight coset leader in `COSETS`
3. **for** i going from 1 to $m-1$ **do**
   (a) **for** each coset $l \in$ `COSETS` **do**
      compute $l+v$ and scan all syndromes stored in `COSETS` to determine the coset leader of the coset containing $l+v$. This is the **buddy** $\beta(l)$;
      store both l and $\beta(l)$ are in a temporary array `BUDDIES`;
   (b) $\{$*Now, we have each coset and its buddy stored in the array* BUDDIES, *so we proceed to generate the set of cosets for the* $i+1$*'th code*$\}$
   (c) **delete** the contents of the array `COSETS`;
   (d) **for** each coset $l \in$ `COSETS` **do**
      for each length-$\Delta \left\{= (d-\rho)\right\}$ bit-sequence a whose complement, $\bar{a}$ has not yet been iterated **do**
         if $\mathrm{wt}(a \mid l) < \mathrm{wt}\left[\bar{a} \mid \beta_v(l)\right]$ **then**

add $[a \mid l]$ to the array `COSETS`;

else

add $[\bar{a} \mid \beta_v(l)]$ to the array `COSETS`;

endif

(e) *{Now we have the set of new coset leaders for the $i+1$'th code}*

(f) **set** $\rho \longleftarrow$ the maximum weight of coset leaders in the array `COSETS`;

(g) **search** through `COSETS` and record each coset leader of with ρ in an array `RHOS`

(h) **run** method 4.2.1 or method 4.2.2 or method 4.2.3 to determine which coset leader in $c \in$ `RHOS` to add to the code constructed so far

(i) Add $v \longleftarrow (c|1^{d-\rho})$ (unless $d - \rho \leq 0$, in which case $v \longleftarrow c$) to the previously generated GLC^i to create GLC^{i+1};

4. Return GLC^m

Proof of the algorithm: Step 3a merely computes the buddy of each coset as it is defined in Definition 3.1. Then, step 3d merely creates the set $\mathcal{S}'$ as per Theorem 3.2. Finally, steps 3f, 3g, and 3h pick a vector to add to the code consistent with either the standard lexicographic construction of method 4.2.1 or the trellis-oriented construction of method 4.2.2 or the locally optimal construction of method 4.2.3.

□

As promised, we can also bound the time and space complexity of Algorithm 4.1.

Corollary 4.1.1 *Algorithm 4.1 runs in time $O(2^{n_m-m+1})$ and space $O(2^{n_m-m})$, if method 4.2.1 or 4.2.2 is chosen in step 3h.*

Proof: To understand the space bound, we note that there are two arrays stored: `COSETS`, `BUDDIES`. In the j'th iteration of step 3, each of these contains *O(# of cosets in the code $\mathcal{L}_m$)* $= O(2^{n_j-j})$. Since they are rewritten at each iteration, the overall space bound is $O(2^{(n_m-m)})$. The time bound is slightly more complicated. During any iteration j of step 3, we have the following breakup of time (note that d is constant for any particular run of the algorithm):

Step 3a: *O(# of cosets in* $\mathcal{L}_j$*)*= $O(2^{n_j-j})$.

Step 3d: *O(# of cosets in* $\mathcal{L}_j$*)* $*2^{d-\rho} = O(2^{n_j-j} * 2^d) = O(2^{n_j-j})$.

Steps 3f,3g, and 3h: *O(# of cosets in* $\mathcal{L}_j$*)*3* $= O(2^{n_j-j})$.

The bound for step 3h is based on corollaries 4.2.1 and 4.2.2. Thus, all together, the algorithm requires time $O(2^{n_j} - j)$ for each iteration. Thus, summing for $j = 2 \ldots m$, we see a total time $O(2^{n_m-m+1})$ for the algorithm.

□

This bound is particularly good for $\mathrm{GLC}^i\,(r_l, w_l, \mathbb{C})$ with $d = 4$.

Corollary 4.1.2 *For $d = 4$, Algorithm 4.1 over r_l and w_l requires linear time and space.*

Proof: [6, Theorem 3.5] derives the following bound for the binary lexicodes with minimum distance $d = 4$:

$$k = n - 2 - \lfloor \log_2(n-1) \rfloor \tag{4.9}$$

Under this bound, $n - k = O(log(n))$ so that Algorithm 4.1 requires time and space $O(2^{n-k+1}) = O(n)$.

□

This algorithm has computed lexicodes well beyond those in the tables of [9], for small d. It's main weakness is that, as d gets large, the number of cosets in the code grows very fast, thereby requiring a lot of memory and time for algorithmic computations. The computations of this algorithm can be found in the appendices, along with the trellis complexities of the resultant codes, both using the standard construction and the trellis-oriented construction. In the examples present, the standard and trellis-oriented construction yielded mostly the same code parameters, which is quite a mystery. Nevertheless, the trellis-oriented construction does seem to provide codes with a significantly better trellis complexity than their standard counterparts.

4.4 State Bounded Codes

There is a modification of Algorithm 4.1 that is of practical significance. Namely, it is sometimes useful to bound the state complexity of the codes generated so that they can be handled by a real system with real complexity constraints. In fact, we can modify steps 3f, 3g, and 3h of algorithm 4.1 as follows in order to generate codes with state complexity bound β:

Modification 4.1.1 for *each r from ρ down to 1* ***do***

> **run** *method 4.2.2 on each coset leader c of distance r from $\mathbb{C}$ (found in* ***COSETS****), computing the resulting state complexity of the linear code determined by $\mathbb{C} \cup \{1^{n-r}|c\}$ directly from the generator matrix formed (as per Theorem 4.1)*

pick *the first c with resultant state complexity $s \leq \beta$;*

set *ρ to the* effective covering radius $\mathrm{wt}(c)$*;*

Modification 4.1.1 thus generates $\mathrm{GLC}^i(r_b, w_n, \mathbb{C})$ where:

$r_b(\mathbb{C})$ = the maximum distance from v to $\mathbb{C}$ over all v such that the linear code determined by $\mathbb{C} \cup \{1^{n-\delta(v,\mathbb{C})}|v\}$ has maximum state complexity β

$w_b(\mathbb{C})$ = the lexicographically latest vector v of distance $r(\mathbb{C})$ from $\mathbb{C}$ with the property that the linear code determined by $\mathbb{C} \cup \{1^{n-\delta(v,\mathbb{C})}|v\}$ has maximum state complexity β

By making $w_b(\cdot)$ the lexicographically latest vector, we ensure that the added generator will have minimum span, so that by the discussion in Subsection 4.2.2, the returned code will have locally optimal decoding complexity. It is also of value to note that $r_b \leq r_l$ so it is very likely that the codes constructed will have different parameters from the standard lexicodes. Nevertheless, it is clear that Modification 4.1.1 is an exhaustive search for the highest weight coset leader (which will, in turn, form the lowest length constructed code) that maintains the state complexity bound β. One of the side-effects of this modification is that it requires a search through each of the coset leaders in `COSETS` and slows down the overall running time of algorithm 4.1;

however, a careful analysis will show that this slowdown does not affect the *order* of the running time of the algorithm by more than a factor of the length of the code.

4.5 Computations

4.5.1 Data

We have computed several parameters of the specific classes of the *GLC* studied. First, we have computed the lengths and dimensions of the *GLC* codes constructed using Methods 4.2.1 and 4.2.2. We have also computed two characteristics of the *BCJR* trellis attained from these constructed codes: the decoding complexity of the trellis and the maximum number of states in any of its levels. The decoding complexity is a measure of the number of steps needed for decoding with the Viterbi algorithm, and hence is a good characteristic of trellis complexity as shown in [33, 44]. As mentioned before, [33] demonstrates that, for any trellis with edges E and states V, the decoding complexity is $2|E| - |V| + 1$. The maximum number of states in the trellis is an important measure of complexity because it strongly affects the decoding complexity of the trellis; the more states a trellis has, the more places there are for bifurcations, and, in general, the higher the decoding time.

In all cases, the decoding complexity and the maximum number of states in the corresponding trellis is computed using Theorem 4.1. For the trellis-oriented *GLC* codes this is particularly easy because Corollary 4.2.1 tells us that the computed generator matrix will be in *MSGM* form. As an example, the $(64, 50, 6)$ trellis-oriented lexicode is shown in Figure A.1; it contains as subcodes all the trellis-oriented lexicodes of dimension less than 50, where the i'th trellis-oriented lexicode can be deduced by restriction to the first i generators. The p_i's and f_i's of Theorem 4.1 can be read from from the generator matrix.

We have also implemented Modification 4.1.1 to Algorithm 4.1 to compute trellis-oriented *GLC* codes with bounds on the state complexity of the decoding trellis. Tables B.4, B.5 and B.6 show that the state complexity bound has a substantive effect on the quality of the codes achieved. Furthermore, these tables show that, in some cases, this *GLC*-based construction is better than the state-bounded codes computed in [48] using a different technique.

4.5.2 Analysis

Actual parameters of some lexicographic codes can be seen in Tables B.1 to C.4. For the minimum distance $d = 4$, the standard lexicodes have exactly the same parameters as the trellis-oriented *GLC* codes (for all the codes that we have calculated). Since Algorithm 4.1 runs in linear time and space for $d = 4$ (by Corollary 4.1.2), Table B.1 represents a very small part of the table that can actually be computed. One should be able to compute codes up to dimensions $2^{16} = 65536$ with even a home computer.

The minimum distance $d = 6$ codes become more interesting, because they present a visible differences in trellis complexity between the standard lexicode construction and the greedy, trellis-oriented construction. Table B.2 shows the computed parameters of these codes. The difference between their trellis structures can be most visibly seen in Figure C.1 which relates the decoding complexity for the two construction methods. There is a "bouncing" relationship where at some points the trellis-oriented construction produces much better trellis complexities than the standard lexicodes, yet, at other points, it produces the same (or slightly worse) trellis complexities. It is conceivable that this relationship continues as length increases.

The minimum distance $d = 8$ codes show similar characteristics to the $d = 6$ codes. Their parameters are shown in Table B.3 with the corresponding figures C.2 and C.4 depicting the decoding complexities and maximum trellis states respectively.

Figure 4.1 shows that the $(32, 16, 8)$ trellis-oriented *GLC* code achieves a better state complexity than the $(32, 16, 8)$ extended BCH code heuristically minimized in [24] as is predicted in that paper. This is also true of the $(31, 16, 7)$ trellis-oriented lexicode as compared to the $(31, 16, 7)$ BCH code, as seen in Figure 4.2.

Trelli:	0-1-2-3-4-5-6-7-6-7-8-9-8-9-8-7-6-7-7-7-6-6-5-4-3-4-4-4-3-3-2-1-0
ext BCH:	0-1-2-3-4-5-6-7-6-7-8-9-8-9-8-7-6-7-8-9-8-9-8-7-6-7-6-5-4-3-2-1-0

Figure 4.1: State-complexity comparison of a trellis-oriented code and extended BCH code. Specifically, we compare the $(32, 16, 8)$ trellis-oriented *GLC* code (Trelli) to the $(32, 16, 8)$ extended BCH code (ext BCH).

In all, the trellis-oriented *GLC* codes produce quite good *BCJR* trellises, some of which are better than those produced by other heuristics. Moreover, the trellis-oriented codes enjoy the advantage of being close to the good code parameters that

Trelli:	0-1-2-3-4-5-6-6-7-8-9-8-9-8-7-6-7-6-6-6-5-5-4-3-4-4-4-3-3-2-1-0
BCH:	0-1-2-3-4-5-6-6-7-8-9-8-9-8-7-6-7-8-9-8-9-8-7-6-7-6-5-4-3-2-1-0

Figure 4.2: State-complexity comparison of trellis-oriented code and regular BCH code. Specifically, we compare the $(31, 16, 7)$trellis-oriented lexicode (Trelli) and the $(31, 16, 7)$ BCH code (BCH).

are produced by the lexicographic construction. Finally, the Algorithm 4.1 provides a method for generating the code parameters of the lexicodes well beyond the length 44 lexicodes that were computed in [9], as can be seen in Appendix 3.2.

Chapter 5

Conclusion

5.1 Summary

We have studied several important areas in algebraic coding theory from a computational perspective, offering efficient algorithms and computed results where possible. Our two major areas of research concerned the optimization of non-linear codes, and the improvement of a specific class of good linear codes.

Our particular method of non-linear code optimization involved relaxation of traditional minimum distance constraints. We developed an expression for the probability of decoding error of a code, which served as a better measure of a code's performance than the minimum distance of the code. In addition, we have demonstrated the optimality of the Perfect Codes using this performance measure. We have also developed and implemented algorithms for optimizing and augmenting codes for better performance in error-correction over a channel. Finally, we have developed and implemented an interactive visualization program for manually improving code performance.

The basis of our work in the study of linear codes was presented in Theorem 3.2, which supplied us with a fundamental understanding of the coset structure of codes created by the Generalized Lexicographic Construction. With this understanding we have improved upon the bounds of [6] on the very good code parameters of lexicodes. We have also developed a relatively fast algorithm for generating the Generalized Lexicographic Construction, which we have later discovered to be very similar to the algorithm in [6, p.16]. We have studied how this algorithm can be directed to produce standard lexicodes and locally-optimal codes. In addition, we have adapted this algorithm to design trellis-oriented codes and codes with bounded trellis-state dimensions for use in fast decoding.

Our implementation of this algorithm was used both to extend the list of known lexicodes first published in [9], and to show how our greedy heuristic involved in mak-

ing trellis-oriented codes can significantly effect the trellis-complexity of the resulting code, even improving on the greedy algorithm in [24] as predicted in that paper. Moreover, our computations of trellis-state bounded codes show some improvement over the similarly designed codes in [48].

5.2 Future Directions

Many possibilities for improvements remain, and many questions remain unanswered.

The code optimization algorithms in Chapter 2 can probably be improved by real genetic algorithms and their implementation speeded up by making use of their inherent locality for massive parallelization. These algorithms can also be guided heuristically by the probability of error expression we developed, which naturally bounds which codes can and cannot be easily improved. Finally, the implementation of the visualization in the same chapter becomes too cumbersome and slow for codes of greater than 15 dimensions, so that improvements are needed.

As concerns Chapter 3, we would like to see the lexicographic construction (and the GLC) generalized to non-binary fields, with Theorem 3.2 updated accordingly; we do not believe that this would be too difficult a task, and, in fact, it seems that [6] has already researched this idea. Finally, based on empirical data and the generality of our analysis, we strongly suspect that the bounds of Section 3.3 can be greatly improved by taking into account the specific structure of the cosets of a lexicode, or by a more sophisticated count of the worst case coset-buddy pairings, and we have initiated work in this direction.

Chapter 4 poses the most interesting questions of this paper. First of all, there is a peculiar resemblance between the code parameters of the standard lexicodes and the trellis-oriented GLC codes; in the case of $d = 4$ it seems that these two families of codes are exactly the same. However, for higher distances, the parameters only seem to differ for a small set of codes. It is also of interest to see if Algorithm 4.1 can be modified to prevent the exponential memory usage which is the bottleneck of current implementations. Of special consideration would be to determine if there are any coset leaders that could never affect the covering radius (in which case they need not be stored in memory). Finally, all the algorithms in this chapter can be applied from an arbitrary start point, that is from any given code; as [10] points out, lexicodes are very similar to laminated lattices, and lexicographic extension can

be seen as the counterpart of lamination, which produces some of the best known lattices. It would be interesting to run these algorithms on known, optimal codes and to see what parameters of extended codes will be achieved.

Appendix A

Sample Algorithm Output

The following is a sample output from our implementation of Algorithm 4.1, where method 4.2.2 is used to construct the generator matrix of a trellis-oriented GLC code.

```
                                                                              111 111
                                                                          111 111 000
                                                                       11 110 100 100
                                                                      110 101 010 010
                                                                   11 111 100 000 000
                                                                  110 110 011 000 000
                                                                1 100 111 000 100 000
                                                               11 000 101 010 010 000
                                                              111 010 110 000 000 000
                                                           11 101 110 000 000 000 000
                                                          110 110 101 000 000 000 000
                                                        1 100 101 000 101 000 000 000
                                                      111 111 000 000 000 000 000 000
                                                    1 101 000 111 000 000 000 000 000
                                                   11 111 010 100 100 000 000 000 000
                                                  101 111 100 100 000 100 000 000 000
                                                1 101 011 010 000 000 000 000 000 000
                                               11 110 111 100 100 000 000 010 000 000
                                            1 111 110 000 000 000 000 000 000 000 000
                                           11 011 001 100 000 000 000 000 000 000 000
                                          111 110 000 001 000 000 000 000 000 000 000
                                        1 100 011 101 000 000 000 000 000 000 000 000
                                       10 101 101 000 010 000 000 000 000 000 000 000
                                      101 010 100 100 000 100 000 000 000 000 000 000
                                    1 110 101 110 000 000 000 100 000 000 000 000 000
                                   10 010 001 110 000 000 000 010 000 000 000 000 000
                                  111 111 101 010 100 000 000 000 000 000 000 000 000
                               11 111 100 000 000 000 000 000 000 000 000 000 000 000
                              110 110 011 000 000 000 000 000 000 000 000 000 000 000
                            1 111 000 010 100 000 000 000 000 000 000 000 000 000 000
                           11 110 101 000 000 000 000 000 000 000 000 000 000 000 000
                          110 000 110 010 001 000 000 000 000 000 000 000 000 000 000
                        1 010 110 010 000 000 100 000 000 000 000 000 000 000 000 000
                       11 111 111 100 000 010 000 000 000 000 000 000 000 000 000 000
                      100 011 001 100 000 000 010 000 000 000 000 000 000 000 000 000
                    1 110 110 100 000 000 000 000 000 000 000 000 000 000 000 000 000
                   11 000 010 001 110 000 000 000 000 000 000 000 000 000 000 000 000
                  110 110 100 010 100 000 000 001 000 000 000 000 000 000 000 000 000
               11 111 010 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
              110 110 101 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
            1 001 101 100 100 000 000 000 000 000 000 000 000 000 000 000 000 000 000
           10 111 100 000 001 000 000 000 000 000 000 000 000 000 000 000 000 000 000
          111 111 010 100 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
        1 101 100 011 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
       10 010 011 010 000 010 000 000 000 000 000 000 000 000 000 000 000 000 000 000
      110 001 111 100 000 000 010 000 000 000 000 000 000 000 000 000 000 000 000 000
    1 110 000 010 100 000 000 001 000 000 000 000 000 000 000 000 000 000 000 000 000
   11 001 011 111 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
  110 100 010 000 100 000 000 000 100 000 000 000 000 000 000 000 000 000 000 000 000
1 111 011 101 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
```

Figure A.1: The $(64, 50, 6)$ trellis-oriented *GLC* code.

Appendix B

GLC Code Parameters

We compute constructions of the various *GLC* codes studied in the paper.

B.1 Lexicodes and Trellis-Oriented *GLC* codes

The following tables contain information about the lexicodes and the trellis-oriented *GLC* codes generated by Algorithm 4.1 for distances $d = 4$, $d = 6$, and $d = 8$. For all of these distances, we were able to compute lexicodes well beyond the length 44 boundary of the codes computed in [9]. For each constructed code, we computed:

- the length of the code;
- the dimension of the code;
- the maximum number of states found at any depth of the *BCJR* trellis representing the code;
- and the value $2|E| - |V| + 1$ which represents the decoding complexity of the Viterbi algorithm when run on the *BCJR* trellis representing the code.

Where compared parameter values are equal between the two types of codes, we have combined their columns into one column, with a shared value centered within it.

Table B.1: Parameters of $\mathbf{d=4}$ codes. We examine the following parameters for lexicodes and trellis-oriented *GLC* codes: length, dimension, maximum number of states and the number of steps required for Viterbi decoding $(2|E|-|V|+1)$ using the BCJR trellis.

<table>
<tr><th rowspan="2">Dim-ension</th><th>Lexicodes</th><th>Trellis-oriented</th><th>Lexicodes</th><th>Trellis-oriented</th><th>Lexicodes</th><th>Trellis-oriented</th></tr>
<tr><th>Length</th><th>Length</th><th>States</th><th>States</th><th>$2|E|-|V|+1$</th><th>$2|E|-|V|+1$</th></tr>
<tr><td>1</td><td colspan="2">4</td><td colspan="2">1</td><td colspan="2">9</td></tr>
<tr><td>2</td><td colspan="2">6</td><td colspan="2">2</td><td colspan="2">19</td></tr>
<tr><td>3</td><td colspan="2">7</td><td colspan="2">3</td><td colspan="2">39</td></tr>
<tr><td>4</td><td colspan="2">8</td><td colspan="2">3</td><td colspan="2">55</td></tr>
<tr><td>5</td><td colspan="2">10</td><td colspan="2">3</td><td colspan="2">69</td></tr>
<tr><td>6</td><td colspan="2">11</td><td colspan="2">3</td><td colspan="2">107</td></tr>
<tr><td>7</td><td colspan="2">12</td><td colspan="2">3</td><td colspan="2">123</td></tr>
<tr><td>8</td><td colspan="2">13</td><td colspan="2">4</td><td colspan="2">235</td></tr>
<tr><td>9</td><td colspan="2">14</td><td colspan="2">4</td><td colspan="2">259</td></tr>
<tr><td>10</td><td colspan="2">15</td><td colspan="2">4</td><td colspan="2">331</td></tr>
<tr><td>11</td><td colspan="2">16</td><td colspan="2">4</td><td colspan="2">355</td></tr>
<tr><td>12</td><td colspan="2">18</td><td colspan="2">4</td><td colspan="2">369</td></tr>
<tr><td>13</td><td colspan="2">19</td><td colspan="2">4</td><td colspan="2">407</td></tr>
<tr><td>14</td><td colspan="2">20</td><td colspan="2">4</td><td colspan="2">423</td></tr>
<tr><td>15</td><td colspan="2">21</td><td colspan="2">4</td><td colspan="2">563</td></tr>
<tr><td>16</td><td colspan="2">22</td><td colspan="2">4</td><td colspan="2">587</td></tr>
<tr><td>17</td><td colspan="2">23</td><td colspan="2">4</td><td colspan="2">659</td></tr>
<tr><td>18</td><td colspan="2">24</td><td colspan="2">4</td><td colspan="2">683</td></tr>
<tr><td>19</td><td colspan="2">25</td><td colspan="2">5</td><td colspan="2">1,219</td></tr>
<tr><td>20</td><td colspan="2">26</td><td colspan="2">5</td><td colspan="2">1,243</td></tr>
<tr><td>21</td><td colspan="2">27</td><td colspan="2">5</td><td colspan="2">1,315</td></tr>
<tr><td>22</td><td colspan="2">28</td><td colspan="2">5</td><td colspan="2">1,339</td></tr>
<tr><td>23</td><td colspan="2">29</td><td colspan="2">5</td><td colspan="2">1,603</td></tr>
<tr><td>24</td><td colspan="2">30</td><td colspan="2">5</td><td colspan="2">1,627</td></tr>
<tr><td>25</td><td colspan="2">31</td><td colspan="2">5</td><td colspan="2">1,699</td></tr>
<tr><td>26</td><td colspan="2">32</td><td colspan="2">5</td><td colspan="2">1,723</td></tr>
<tr><td>27</td><td colspan="2">34</td><td colspan="2">5</td><td colspan="2">1,737</td></tr>
<tr><td>28</td><td colspan="2">35</td><td colspan="2">5</td><td colspan="2">1,775</td></tr>
<tr><td>29</td><td colspan="2">36</td><td colspan="2">5</td><td colspan="2">1,791</td></tr>
<tr><td>30</td><td colspan="2">37</td><td colspan="2">5</td><td colspan="2">1,931</td></tr>
<tr><td>31</td><td colspan="2">38</td><td colspan="2">5</td><td colspan="2">1,955</td></tr>
<tr><td>32</td><td colspan="2">39</td><td colspan="2">5</td><td colspan="2">2,027</td></tr>
<tr><td>33</td><td colspan="2">40</td><td colspan="2">5</td><td colspan="2">2,051</td></tr>
<tr><td>34</td><td colspan="2">41</td><td colspan="2">5</td><td colspan="2">2,643</td></tr>
<tr><td>35</td><td colspan="2">42</td><td colspan="2">5</td><td colspan="2">2,667</td></tr>
<tr><td>36</td><td colspan="2">43</td><td colspan="2">5</td><td colspan="2">2,739</td></tr>
<tr><td>37</td><td colspan="2">44</td><td colspan="2">5</td><td colspan="2">2,763</td></tr>
<tr><td>38</td><td colspan="2">45</td><td colspan="2">5</td><td colspan="2">3,027</td></tr>
</table>

continued on next page...

(continued from previous page) Parameters of the $\mathbf{d = 4}$ codes

Dimension	*Lexicodes*	*Trellis-oriented*	*Lexicodes*	*Trellis-oriented*	*Lexicodes*	*Trellis-oriented*
	Length	*Length*	*States*	*States*	$2\|E\| - \|V\| + 1$	$2 * \|E\| - \|V\| + 1$
39	46		5		3,051	
40	47		5		3,123	
41	48		5		3,147	
42	49		6		5,491	
43	50		6		5,515	
44	51		6		5,587	
45	52		6		5,611	
46	53		6		5,875	
47	54		6		5,899	
48	55		6		5,971	
49	56		6		5,995	
50	57		6		7,027	
51	58		6		7,051	
52	59		6		7,123	
53	60		6		7,147	
54	61		6		7,411	
55	62		6		7,435	
56	63		6		7,507	
57	64		6		7,531	
58	66		6		7,545	
59	67		6		7,583	
60	68		6		7,599	
61	69		6		7,739	
62	70		6		7,763	
63	71		6		7,835	
64	72		6		7,859	
65	73		6		8,451	
66	74		6		8,475	
67	75		6		8,547	
68	76		6		8,571	
69	77		6		8,835	
70	78		6		8,859	
71	79		6		8,931	
72	80		6		8,955	
73	81		6		11,411	
74	82		6		11,435	
75	83		6		11,507	
76	84		6		11,531	
77	85		6		11,795	
78	86		6		11,819	
79	87		6		11,891	
80	88		6		11,915	
81	89		6		12,947	

continued on next page...

(continued from previous page) Parameters of the $\mathbf{d = 4}$ codes

Dimension	*Lexicodes* Length	*Trellis-oriented* Length	*Lexicodes* States	*Trellis-oriented* States	*Lexicodes* $2\|E\| - \|V\| + 1$	*Trellis-oriented* $2 * \|E\| - \|V\| + 1$
82	90		6		12,971	
83	91		6		13,043	
84	92		6		13,067	
85	93		6		13,331	
86	94		6		13,355	
87	95		6		13,427	
88	96		6		13,451	
89	97		7		23,251	
90	98		7		23,275	
91	99		7		23,347	
92	100		7		23,371	
93	101		7		23,635	
94	102		7		23,659	
95	103		7		23,731	
96	104		7		23,755	
97	105		7		24,787	
98	106		7		24,811	
99	107		7		24,883	
100	108		7		24,907	
101	109		7		25,171	
102	110		7		25,195	
103	111		7		25,267	
104	112		7		25,291	
105	113		7		29,395	
106	114		7		29,419	
107	115		7		29,491	
108	116		7		29,515	
109	117		7		29,779	
110	118		7		29,803	
111	119		7		29,875	
112	120		7		29,899	
113	121		7		30,931	
114	122		7		30,955	
115	123		7		31,027	
116	124		7		31,051	
117	125		7		31,315	
118	126		7		31,339	
119	127		7		31,411	
120	128		7		31,435	

Table B.2: Parameters of $\mathbf{d = 6}$ codes. We examine the following parameters for lexicodes and trellis-oriented *GLC* codes: length, dimension, maximum number of states and the number of steps required for Viterbi decoding ($2|E| - |V| + 1$) using the BCJR trellis.

<table>
<tr><th rowspan="2">Dim-ension</th><th>Lexicode</th><th>Trellis-oriented</th><th>Lexicode</th><th>Trellis-oriented</th><th>Lexicode</th><th>Trellis-oriented</th></tr>
<tr><th>Length</th><th>Length</th><th>States</th><th>States</th><th>$2|E| - |V| + 1$</th><th>$2|E| - |V| + 1$</th></tr>
<tr><td>1</td><td colspan="2">6</td><td colspan="2">1</td><td colspan="2">13</td></tr>
<tr><td>2</td><td colspan="2">9</td><td colspan="2">2</td><td colspan="2">27</td></tr>
<tr><td>3</td><td colspan="2">11</td><td colspan="2">3</td><td colspan="2">55</td></tr>
<tr><td>4</td><td colspan="2">12</td><td colspan="2">4</td><td colspan="2">111</td></tr>
<tr><td>5</td><td colspan="2">14</td><td colspan="2">4</td><td>181</td><td>141</td></tr>
<tr><td>6</td><td colspan="2">15</td><td>5</td><td>4</td><td>335</td><td>215</td></tr>
<tr><td>7</td><td colspan="2">16</td><td colspan="2">5</td><td>451</td><td>387</td></tr>
<tr><td>8</td><td colspan="2">17</td><td colspan="2">6</td><td colspan="2">747</td></tr>
<tr><td>9</td><td colspan="2">18</td><td>7</td><td>6</td><td>1,275</td><td>955</td></tr>
<tr><td>10</td><td colspan="2">20</td><td>8</td><td>6</td><td>1,821</td><td>1,053</td></tr>
<tr><td>11</td><td colspan="2">21</td><td>8</td><td>6</td><td>2,655</td><td>1,279</td></tr>
<tr><td>12</td><td colspan="2">22</td><td>8</td><td>7</td><td>3,555</td><td>2,083</td></tr>
<tr><td>13</td><td colspan="2">24</td><td>8</td><td>7</td><td>4,333</td><td>2,117</td></tr>
<tr><td>14</td><td colspan="2">25</td><td>9</td><td>7</td><td>6,143</td><td>2,351</td></tr>
<tr><td>15</td><td colspan="2">26</td><td>9</td><td>7</td><td>7,715</td><td>3,011</td></tr>
<tr><td>16</td><td colspan="2">27</td><td>9</td><td>8</td><td>9,323</td><td>5,035</td></tr>
<tr><td>17</td><td colspan="2">28</td><td>9</td><td>8</td><td>10,235</td><td>5,435</td></tr>
<tr><td>18</td><td colspan="2">29</td><td colspan="2">9</td><td>12,827</td><td>10,651</td></tr>
<tr><td>19</td><td>30</td><td>31</td><td>10</td><td>9</td><td>14,939</td><td>10,701</td></tr>
<tr><td>20</td><td colspan="2">32</td><td>10</td><td>9</td><td>18,141</td><td>10,943</td></tr>
<tr><td>21</td><td colspan="2">33</td><td>10</td><td>9</td><td>19,167</td><td>11,683</td></tr>
<tr><td>22</td><td colspan="2">34</td><td>10</td><td>9</td><td>23,523</td><td>12,267</td></tr>
<tr><td>23</td><td colspan="2">35</td><td>10</td><td>9</td><td>25,067</td><td>13,947</td></tr>
<tr><td>24</td><td colspan="2">36</td><td>10</td><td>9</td><td>29,691</td><td>18,075</td></tr>
<tr><td>25</td><td colspan="2">37</td><td colspan="2">10</td><td>31,259</td><td>27,739</td></tr>
<tr><td>26</td><td colspan="2">38</td><td colspan="2">11</td><td>44,635</td><td>47,835</td></tr>
<tr><td>27</td><td colspan="2">39</td><td colspan="2">11</td><td>53,979</td><td>54,235</td></tr>
<tr><td>28</td><td colspan="2">41</td><td colspan="2">11</td><td>55,005</td><td>54,285</td></tr>
<tr><td>29</td><td colspan="2">42</td><td colspan="2">11</td><td>62,431</td><td>54,527</td></tr>
<tr><td>30</td><td colspan="2">43</td><td colspan="2">11</td><td>71,139</td><td>55,203</td></tr>
<tr><td>31</td><td colspan="2">44</td><td colspan="2">11</td><td>77,291</td><td>55,403</td></tr>
<tr><td>32</td><td colspan="2">45</td><td colspan="2">11</td><td>83,451</td><td>58,107</td></tr>
<tr><td>33</td><td colspan="2">46</td><td colspan="2">11</td><td>89,627</td><td>64,027</td></tr>
<tr><td>34</td><td colspan="2">47</td><td colspan="2">11</td><td>95,835</td><td>71,771</td></tr>
<tr><td>35</td><td colspan="2">48</td><td colspan="2">11</td><td>102,107</td><td>92,891</td></tr>
<tr><td>36</td><td colspan="2">49</td><td>12</td><td>11</td><td>144,347</td><td>93,275</td></tr>
<tr><td>37</td><td colspan="2">50</td><td>12</td><td>11</td><td>157,147</td><td>96,987</td></tr>
<tr><td>38</td><td colspan="2">51</td><td colspan="2">12</td><td>170,459</td><td>146,395</td></tr>
</table>

continued on next page...

(continued from previous page) Parameters of the **d = 6** lexicodes

Dimension	*Lexicode Length*	*Trellis-oriented Length*	*Lexicode States*	*Trellis-oriented States*	*Lexicode* $2\|E\|-\|V\|+1$	*Trellis-oriented* $2*\|E\|-\|V\|+1$
39	52	*53*	12		178,651	*146,493*
40	53	*54*	12		190,939	*146,783*
41	55		12		203,229	*147,555*
42	56		12		215,519	*150,251*
43	57		12		221,667	*151,035*
44	58		12		240,107	*151,835*
45	59		12		252,411	*157,275*
46	60		12		253,979	*172,763*
47	61		12		267,867	*204,763*
48	62		12		277,211	*206,811*
49	63		12		302,043	*275,931*
50	64		12		308,699	*277,467*
51	65		12		328,155	*279,003*
52	66		12		336,347	*299,483*
53	67		12		348,635	*301,019*
54	68		13		541,147	*488,923*
55	69		13		565,723	*498,139*
56	70	*71*	13		571,867	*498,189*
57	72		13		602,589	*498,431*
58	73		13		639,455	*499,107*
59	74		13		664,035	*500,843*
60	75		13		688,619	*505,083*
61	76		13		700,923	*505,883*
62	77		13		704,027	*516,699*
63	78		13		704,859	*518,363*
64	79		13		707,291	*526,299*
65	80		13		750,555	*560,603*
66	81		13		787,931	*564,699*
67	82		13		791,003	*668,123*
68	83		13		819,675	*671,195*
69	84		13	*14*	834,011	*913,883*
70	85		13	*14*	930,267	*916,955*
71	86		13	*14*	954,843	*938,459*
72	87		13	*14*	979,419	*956,891*
73	88		14		1,487,323	*1,030,619*
74	89		14		1,490,395	*1,055,195*
75	90	*91*	14		1,518,043	*1,055,293*
76	91	*92*	14		1,548,763	*1,055,583*
77	92	*93*	14		1,659,355	*1,056,355*
78	93	*94*	14		1,683,931	*1,056,459*
79	95		14		1,782,237	*1,058,491*
80	96		14		1,806,815	*1,059,483*
81	97		14		1,880,547	*1,066,843*

continued on next page...

(continued from previous page) Parameters of the **d = 6** lexicodes

Dimension	*Lexicode Length*	*Trellis-oriented Length*	*Lexicode States*	*Trellis-oriented States*	*Lexicode* $2\|E\| - \|V\| + 1$	*Trellis-oriented* $2 * \|E\| - \|V\| + 1$
82	98		14		1,929,707	*1,067,355*
83	99		14		1,978,875	*1,079,515*
84	100		14		2,003,483	*1,109,979*
85	101		14		2,077,275	*1,121,755*
86	102		14		2,101,979	*1,127,899*
87	103		14		2,108,379	*1,212,891*
88	104		14		2,139,611	*1,225,179*
89	105		14		2,177,499	*1,419,739*
90	106		14		2,183,643	*1,976,795*
91	107		14		2,216,411	*1,979,867*
92	108		14		2,228,699	*1,989,083*
93	109		14		2,429,403	*2,013,659*
94	110		14		2,511,323	*2,025,947*
95	111		14		2,560,475	*2,038,235*
96	112		14	*15*	2,609,627	*3,607,003*
97	113		14	*15*	2,658,779	*3,613,147*
98	114		15		4,067,803	*3,631,579*
99	115		15		4,092,379	*3,828,187*
100	116		15		4,215,259	*3,831,259*

Table B.3: Parameters of $\mathbf{d = 8}$ codes. We examine the following parameters for lexicodes and trellis-oriented *GLC* codes: length, dimension, max. number of states and Viterbi decoding complexity with the BCJR trellis.

<table>
<tr><th rowspan="2">Dim-ension</th><th>Standard</th><th>Trellis-oriented</th><th>Standard</th><th>Trellis-oriented</th><th>Standard</th><th>Trellis-oriented</th></tr>
<tr><th>Length</th><th>Length</th><th>States</th><th>States</th><th>$2|E|-|V|+1$</th><th>$2|E|-|V|+1$</th></tr>
<tr><td>1</td><td colspan="2">8</td><td colspan="2">1</td><td colspan="2">17</td></tr>
<tr><td>2</td><td colspan="2">12</td><td colspan="2">2</td><td colspan="2">35</td></tr>
<tr><td>3</td><td colspan="2">14</td><td colspan="2">3</td><td colspan="2">71</td></tr>
<tr><td>4</td><td colspan="2">15</td><td colspan="2">4</td><td colspan="2">143</td></tr>
<tr><td>5</td><td colspan="2">16</td><td colspan="2">4</td><td colspan="2">195</td></tr>
<tr><td>6</td><td colspan="2">18</td><td colspan="2">5</td><td colspan="2">341</td></tr>
<tr><td>7</td><td colspan="2">19</td><td colspan="2">6</td><td colspan="2">647</td></tr>
<tr><td>8</td><td colspan="2">20</td><td colspan="2">6</td><td colspan="2">779</td></tr>
<tr><td>9</td><td colspan="2">21</td><td colspan="2">7</td><td colspan="2">1,547</td></tr>
<tr><td>10</td><td colspan="2">22</td><td colspan="2">8</td><td colspan="2">2,395</td></tr>
<tr><td>11</td><td colspan="2">23</td><td colspan="2">9</td><td colspan="2">4,219</td></tr>
<tr><td>12</td><td colspan="2">24</td><td colspan="2">9</td><td colspan="2">4,475</td></tr>
<tr><td>13</td><td colspan="2">28</td><td colspan="2">9</td><td colspan="2">4,529</td></tr>
<tr><td>14</td><td colspan="2">30</td><td colspan="2">9</td><td colspan="2">4,777</td></tr>
<tr><td>15</td><td colspan="2">31</td><td colspan="2">9</td><td colspan="2">5,463</td></tr>
<tr><td>16</td><td colspan="2">32</td><td colspan="2">9</td><td colspan="2">5,515</td></tr>
<tr><td>17</td><td colspan="2">34</td><td colspan="2">9</td><td>7,645</td><td>5,693</td></tr>
<tr><td>18</td><td colspan="2">35</td><td colspan="2">9</td><td>12,671</td><td>6,143</td></tr>
<tr><td>19</td><td colspan="2">36</td><td colspan="2">9</td><td>12,803</td><td>6,275</td></tr>
<tr><td>20</td><td colspan="2">37</td><td>10</td><td>9</td><td>24,267</td><td>7,691</td></tr>
<tr><td>21</td><td colspan="2">38</td><td>10</td><td>9</td><td>25,115</td><td>8,539</td></tr>
<tr><td>22</td><td colspan="2">39</td><td>10</td><td>9</td><td>26,939</td><td>10,363</td></tr>
<tr><td>23</td><td colspan="2">40</td><td>10</td><td>9</td><td>27,195</td><td>10,619</td></tr>
<tr><td>24</td><td colspan="2">42</td><td colspan="2">10</td><td>31,805</td><td>17,853</td></tr>
<tr><td>25</td><td colspan="2">43</td><td colspan="2">11</td><td>41,791</td><td>33,087</td></tr>
<tr><td>26</td><td colspan="2">44</td><td colspan="2">11</td><td>41,987</td><td>33,283</td></tr>
<tr><td>27</td><td colspan="2">45</td><td colspan="2">12</td><td colspan="2">65,227</td></tr>
<tr><td>28</td><td colspan="2">46</td><td colspan="2">12</td><td colspan="2">67,163</td></tr>
<tr><td>29</td><td colspan="2">47</td><td colspan="2">12</td><td>72,571</td><td>78,203</td></tr>
<tr><td>30</td><td colspan="2">48</td><td colspan="2">12</td><td>72,827</td><td>78,459</td></tr>
<tr><td>31</td><td>49</td><td>50</td><td>13</td><td>12</td><td>135,611</td><td>80,317</td></tr>
<tr><td>32</td><td>50</td><td>51</td><td>13</td><td>12</td><td>169,019</td><td>87,103</td></tr>
<tr><td>33</td><td>51</td><td>52</td><td>13</td><td>12</td><td>248,123</td><td>88,643</td></tr>
<tr><td>34</td><td>52</td><td>53</td><td colspan="2">13</td><td>248,507</td><td>137,547</td></tr>
<tr><td>35</td><td>53</td><td>54</td><td>14</td><td>13</td><td>427,579</td><td>138,331</td></tr>
<tr><td>36</td><td>54</td><td>55</td><td>14</td><td>13</td><td>431,419</td><td>142,203</td></tr>
<tr><td>37</td><td>55</td><td>56</td><td>14</td><td>13</td><td>442,171</td><td>142,459</td></tr>
<tr><td>38</td><td>56</td><td>57</td><td colspan="2">14</td><td>442,555</td><td>274,875</td></tr>
<tr><td>39</td><td colspan="2">58</td><td colspan="2">14</td><td>487,997</td><td>308,283</td></tr>
</table>

continued on next page...

(continued from previous page) Parameters of the $\mathbf{d} = \mathbf{8}$ lexicodes

<table>
<tr><th rowspan="2">Dim-
ension</th><th>Standard</th><th>Trellis-
oriented</th><th>Standard</th><th>Trellis-
oriented</th><th>Standard</th><th>Trellis-
oriented</th></tr>
<tr><th>Length</th><th>Length</th><th>States</th><th>States</th><th>$2|E| - |V| + 1$</th><th>$2 * |E| - |V| + 1$</th></tr>
<tr><td>40</td><td colspan="2">59</td><td colspan="2">14</td><td>628,031</td><td>457,019</td></tr>
<tr><td>41</td><td colspan="2">60</td><td colspan="2">14</td><td>628,227</td><td>460,091</td></tr>
<tr><td>42</td><td colspan="2">62</td><td colspan="2">14</td><td>629,197</td><td>460,861</td></tr>
<tr><td>43</td><td colspan="2">63</td><td colspan="2">14</td><td>631,903</td><td>464,703</td></tr>
<tr><td>44</td><td colspan="2">64</td><td colspan="2">14</td><td>632,035</td><td>464,899</td></tr>
<tr><td>45</td><td colspan="2">65</td><td>15</td><td>14</td><td>1,263,819</td><td>581,835</td></tr>
<tr><td>46</td><td colspan="2">66</td><td colspan="2">15</td><td>1,287,195</td><td>1,053,275</td></tr>
<tr><td>47</td><td colspan="2">67</td><td>15</td><td>16</td><td>1,346,235</td><td>2,106,299</td></tr>
<tr><td>48</td><td>68</td><td>69</td><td>15</td><td>16</td><td>1,701,883</td><td>2,106,557</td></tr>
</table>

B.2 State Bounded *GLC* Codes

The following tables contain the code parameters for state-bounded *GLC* codes with log-state bounds of 4 (i.e. 16 states maximum), 5 (i.e. 32 states maximum), and 6 (i.e. 64 states maximum) respectively. For each of these bounds, we list, in order of code dimension, the length of the generated code for each of the minimum distances 4 through 8.

Dimension	*d=4*	*d=5*	*d=6*	*d=7*	*d=8*
1	4	5	6	7	8
2	6	8	9	11	12
3	7	10	11	13	14
4	8	11	12	14	15
5	10	13	14	15	16
6	11	14	15	18	20
7	12	16	17	22	24
8	13	18	19	24	26
9	14	20	21	25	27
10	15	21	23	26	28
11	16	23	25	29	32
12	18	24	27	33	36
13	19	26	29	35	
14	20	28	31		
15	21	30	33		
16	22	31	35		
17	23	33	37		
18	24	34	39		
19	26	36			
20	27	38			
21	28	40			
22	29	41			
23	30	43			
24	31				
25	32				
26	34				
27	35				
28	36				
29	37				
30	38				
31	39				
32	40				
33	42				
34	40				
35	41				
36	43				
37	44				
38	45				
39	47				
40	48				
41	49				
42	50				
43	51				
44	52				
45	53				
46	55				
47	56				
48	57				
49	58				
50	59				

Table B.4: Codes with a trellis log-state bound of 4. Displayed are lengths of the codes generated by Modification 4.1.1 to Algorithm 4.1 under a *maximum log-state bound of 4* (i.e. 16 states) for a given dimension and minimum distance.

Dimension	*d=4*	*d=5*	*d=6*	*d=7*	*d=8*
1	4	5	6	7	8
2	6	8	9	11	12
3	7	10	11	13	14
4	8	11	12	14	15
5	10	13	14	14	16
6	11	14	15	17	18
7	12	15	16	19	20
8	13	17	18	22	24
9	14	18	19	24	26
10	15	20	21	25	27
11	16	21	23	27	28
12	18	23	24	29	30
13	19	24	26	31	32
14	20	25	27	33	36
15	21	27	28	35	
16	22	28	30	36	
17	23	30	31		
18	24	31	33		
19	25	33	35		
20	26	34	36		
21	27	35	38		
22	28	37	39		
23	29	38	40		
24	30	40	42		
25	31	41	43		
26	32	43			
27	34	44			
28	35	46			
29	36	47			
30	37	48			
31	38	50			
32	39	51			
33	40	53			
34	41	54			
35	42	56			
36	43	57			
37	44	58			
38	45				
39	46				
40	47				
41	48				
42	50				
43	51				
44	52				
45	53				
46	54				
47	55				
48	56				
49	57				
50	58				

Table B.5: Codes with a trellis log-state bound of 5. Displayed are lengths of the codes generated by Modification 4.1.1 to Algorithm 4.1 under a *maximum state bound of 5* (i.e. 32 states) for a given dimension and minimum distance.

Dimension	*d=4*	*d=5*	*d=6*	*d=7*	*d=8*
1	4	5	6	7	8
2	6	8	9	11	12
3	7	10	11	13	14
4	8	11	12	14	15
5	10	13	14	15	16
6	11	14	15	17	18
7	12	15	16	18	19
8	13	16	17	19	20
9	14	17	18	22	24
10	15	19	20	24	26
11	16	20	21	25	27
12	18	22	23	27	28
13	19	23	25	29	30
14	20	24	26	30	31
15	21	26	27	32	32
16	22	27	29	33	36
17	23	28	30	35	38
18	24	29	31	37	39
19	25	31	33	39	40
20	26	32	34	40	42
21	27	33	36	42	43
22	28	35	37		
23	29	36	39		
24	30	37	40		
25	31	38	41		
26	32	40	43		
27	34	41	44		
28	35	42	45		
29	36	44	47		
30	37	45	48		
31	38	46	50		
32	39	48	51		
33	40	49	53		
34	41	50	54		
35	42	51			
36	43	53			
37	44	54			
38	45	55			
39	46	57			
40	47	58			
41	48	59			
42	49	61			
43	50	62			
44	51	63			
45	52	64			
46	53				
47	54				
48	55				
49	56				
50	57				

Table B.6: Codes with a trellis log-state bound of 6. Displayed are lengths of the codes generated by Modification 4.1.1 to Algorithm 4.1 under a *maximum state bound of 6* (i.e. 64 states) for a given dimension and minimum distance.

Appendix C

Comparisons

The following figures display comparisons of various parameters between the standard lexicodes and the trellis-oriented GLC codes. Since comparisons are made only between constructed codes with equal code parameters, these figures demonstrate that, for equal code parameters, the trellis-oriented GLC codes have lower trellis complexity than the plain lexicodes.

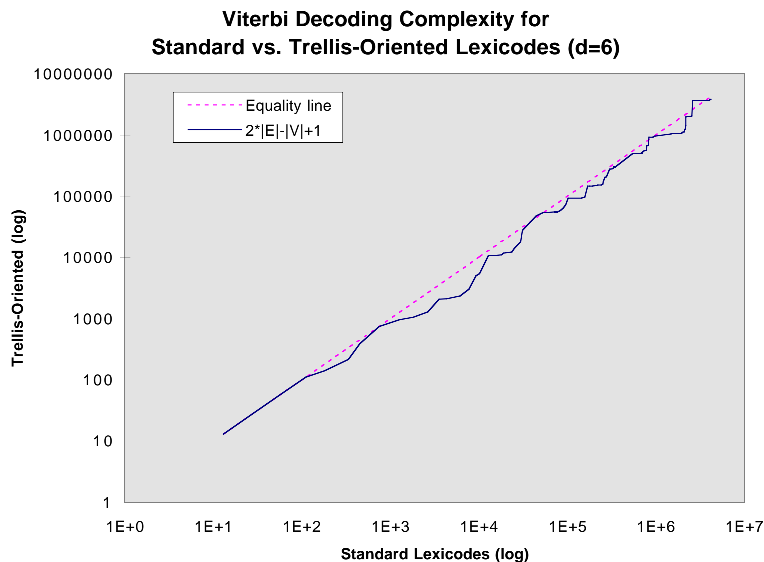

Figure C.1: Decoding complexity comparison of $d = 6$ codes. A comparison of Viterbi decoding complexity for the lexicodes and the trellis-oriented GLC codes with **minimum distance 6**.

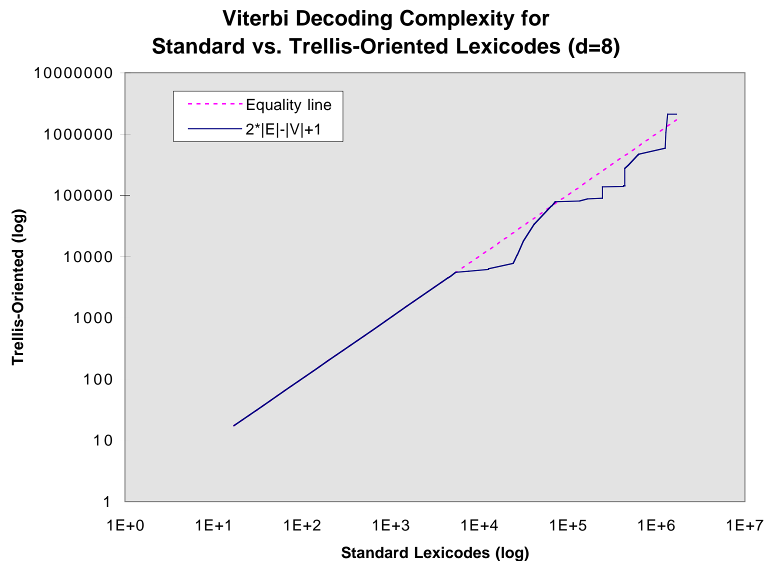

Figure C.2: Decoding complexity comparison of $d = 8$ codes. A comparison of Viterbi decoding complexity for the lexicodes and the trellis-oriented GLC codes with **minimum distance 8**.

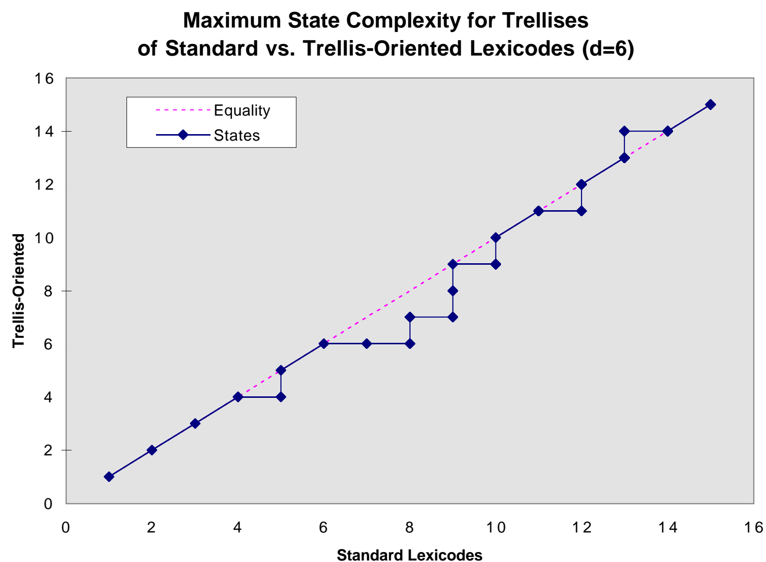

Figure C.3: Maximum state comparison of $d = 6$ codes. A comparison of the maximum number of states in the *BCJR* trellis for the lexicodes and the trellis-oriented *GLC* codes with **minimum distance 6**.

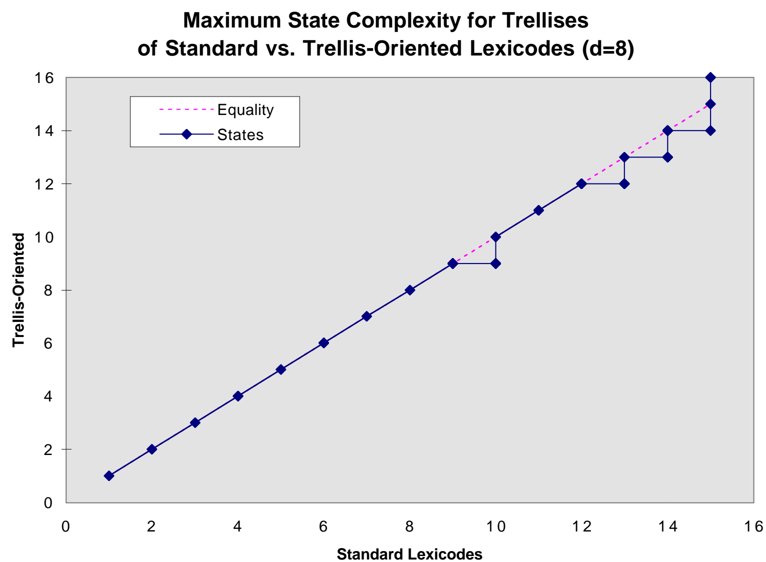

Figure C.4: Maximum state comparison of $d = 8$ codes. A comparison of the maximum number of states in the *BCJR* trellis for the lexicodes and the trellis-oriented *GLC* codes with **minimum distance 8**.

Bibliography

[1] M. Artin, *Algebra*, Prentice Hall 1991.

[2] L.R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inform. Theory*, vol. 20, pp. 284-287, 1974.

[3] A.M. Barg, "Some new NP-complete coding problems," *Problems of Information Transmission*, vol. 30, pp. 23–28, 1994.

[4] R.E. Blahut, *Theory and Practice of Data Transmission Codes* 2nd edition, 1994.

[5] E.R. Berlekamp, R.J. McEliece, and H.C.A. van Tilborg, "On the inherent intractability of certain coding problems," *IEEE Trans. Inform. Theory*, vol. 24, pp. 384–386, 1978.

[6] R.A. Brualdi and V.S. Pless, Greedy codes, *Journal of Comb. Th. (A)*, Sept., 1993.

[7] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, MIT Press 1990.

[8] J.H. Conway, Integral lexicographic codes, *Discrete Math*, 83, 1990, 219-235.

[9] J.H. Conway and N.J.A. Sloane, Lexicographic codes: Error-correcting codes from game theory, *IEEE Trans. Inform. Theory*, vol. IT-32, 1986, 337-348.

[10] J.H. Conway and N.J.A. Sloane, *Sphere Packings, Lattices and Groups*, Springer-Verlag 1993.

[11] T.M. Cover and J.A. Thomas, *Elements of Information Theory*, John Wiley & Sons, Inc., New York 1991.

[12] S. Dolinar, L. Ekroot, A.B. Kiely, R.J. McEliece, and W. Lin, "The permutation trellis complexity of linear block codes," in *Proc. 32-nd Allerton Conference on Comm., Control, and Computing*,Monticello, IL., pp. 60–74, September 1994.

[13] S. Dolinar, L. Ekroot, A.B. Kiely, R.J. McEliece, and W. Lin, "The permutation trellis complexity of linear block codes," *Proc. 32nd Allerton Conference on Comm., Control, and Computing*, Monticello, IL., pp 60-74, September 28-30, 1994.

[14] A.A. El-Gamal, L.A. Hemachandra, I. Shperling, and V.K. Wei, "Using simulated annealing to design good codes," *IEEE Trans. Inform. Theory*, vol. 33, pp. 125-138, 1987.

[15] J. Feigenbaum, "The use of coding theory in computational complexity," preprint.

[16] G.D. Forney, Jr., "Final report on a coding system design for advanced solar missions," Contract NAS2–3637, NASA Ames Research Center, Moffet Field, CA, December 1967.

[17] G.D. Forney, Jr., "The Viterbi algorithm," *Proc. IEEE*, vol. 61, pp. 268–278, 1973.

[18] G.D. Forney, Jr., "Coset codes II: Binary lattices and related codes," *IEEE Trans. Inform. Theory*, vol. 34, pp. 1152–1187, 1988.

[19] G.D. Forney, Jr., "Dimension/length profiles and trellis complexity of linear block codes," *IEEE Trans. Inform. Theory*, vol. 40, pp. 1741–1752, 1994.

[20] G.D. Forney, Jr. and M.D. Trott, "The dynamics of group codes: state spaces, trellis diagrams and canonical encoders," *IEEE Trans. Inform. Theory*, vol. 39, pp. 1491–1513, 1993.

[21] D.S. Herscovici, "Minimal Distance Lexicographic Codes Over an Infinite Alphabet." *IEEE Transactions on Information Theory*, vol. 37, No. 5, September 1991, p. 1366-1368.

[22] K.A.S. Immink, *Coding Techniques for Digital Recorders,* New York: Prentice Hall, 1991.

[23] G.B. Horn and F.R. Kschischang, "Another inherently intractable problem in coding theory," *IEEE Trans. Inform. Theory*, to appear.

[24] F.R. Kschischang and G.B. Horn, "A Heureistic for Ordering a Linear Block Code to Minimize Trellis State Complexity," *32nd Annual Allerton Conference on Communications, Control and Computing*, Monticello, IL., p.75-84, September 28-30, 1994.

[25] F.R. Kschischang and V. Sorokine, "On the trellis structure of block codes," *IEEE Trans. Inform. Theory*, to appear.

[26] A. Lafourcade and A. Vardy, "On trellis complexity of good codes," in *Proc. 28-th Annual Conference on Information Sciences and Systems*, Princeton, NJ., March 1994.

[27] A. Lafourcade and A. Vardy, "Asymptotically good codes have infinite trellis complexity," *IEEE Trans. Inform. Theory*, vol. 41, pp. 555–559, 1995.

[28] A. Lafourcade and A. Vardy, "Lower bounds on trellis complexity of block codes," *IEEE Trans. Inform. Theory*, vol. 41, No. 6, November 1995, to appear.

[29] G.R. Lang and F.M. Longstaff, "A Leech lattice modem," *IEEE J. Select. Areas Comm.*, vol. 7, pp. 968–973, 1989.

[30] V.I. Levenštein, A class of systematic codes, *Soviet Math. Dokl.*, 1:1, 1960, pp. 368-371

[31] S. Lin and D.J. Costello Jr., *Error Control Coding: Fundamentals and Applications*, Prentice-Hall, Inc., New Jersey, 1983.

[32] F.J. MacWilliams and N.J.A. Sloane, *The Theory of Error-Correcting Codes*, North-Holland Publishing Company, New York, 1977.

[33] R.J. McEliece, "On the BCJR trellis for linear block codes." Submitted to *IEEE Trans. on Inform. Theory*, 1994.

[34] C.L. Liu, B.G. Ong, and G.R. Ruth, "A construction scheme for linear and nonlinear codes," *Discrete Math.*, vol. 4, pp. 171–184, 1973.

[35] K.Y. Liu and J.J. Lee, "Recent results on the use of concatenated Reed-Solomon/Viterbi channel coding for space communications," *IEEE Trans. Comm.*, vol. 32, pp.456–471, 1984.

[36] F.J. MacWilliams and N.J.A. Sloane, *The Theory of Error-Correcting Codes*, New York: North-Holland, 1977.

[37] J.L. Massey, "Foundation and methods of channel encoding," *Proc. Int. Conf. Information Theory and Systems*, NTG-Fachberichte, Berlin, 1978.

[38] R.J. McEliece, "On the BCJR trellis for linear block codes," preprint.

[39] D.J. Muder, "Minimal trellises for block codes," *IEEE Trans. Inform. Theory*, vol. 34, pp. 1049–1053, 1988.

[40] W.H. Press, B.P. Flanner, S.A. Teukolsky, *Numerical Recipes in FORTRAN*, Cambridge University Press, 1992.

[41] C.E. Shannon, "A mathematical theory of communication," *Bell Syst. Tech. J.*, vol. 27, pp. 379–423 and pp. 623–656, 1948.

[42] R.J.M. Vaessens, E.H.L. Aarts, and J.H. van Lint, "Genetic algorithms in coding theory: A table of $A_3(n, d)$," preprint.

[43] A. Vardy and Y. Be'ery, "Maximum-likelihood soft decision decoding of BCH codes," *IEEE Trans. Inform. Theory*, vol. 40, pp. 546–554, 1994.

[44] A. Vardy and F.R. Kschischang, "Proof of a Conjecture of McEliece Regarding the Expansion Index of the Minimal Trellis", *IEEE Trans. Inform. Theory*, vol. 42, No. 6, November 1996, to appear.

[45] A.J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Trans. Inform. Theory*, vol. 13, pp. 260–269, 1967.

[46] T.J. Wagner, "A search technique for quasi-perfect codes," *Inform. and Control*, vol. 9, pp. 94–99, 1966.

[47] J.K. Wolf, "Efficient maximum-likelihood decoding of linear block codes," *IEEE Trans. Inform. Theory*, vol. 24, pp. 76–80, 1978.

[48] S. Zhang, "Design of Linear Block Codes with Fixed State Complexity", *Master's Thesis*: University of Toronto, 1996.