ERROR-CORRECTING CODES ON GRAPHS: LEXICODES, TRELLISES AND FACTOR GRAPHS

BY

ARI TRACHTENBERG

S.B., Massachusetts Institute of Technology, 1994
M.S., University of Illinois, 1996

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2000

Urbana, Illinois

Abstract

We consider two graph-based techniques for correcting the errors introduced into a transmission by a noisy channel. In the first technique, which is based on decoding with a layered graph known as a trellis, we examine generalization of heuristically good codes called lexicodes. We propose and analyze a method for designing generalized lexicodes that minimize or constrain various trellis decoding parameters. The second technique is based on decoding with factor graphs, which are smaller than trellis representations and have enjoyed much attention in the recent literature. We look at a specific type of factor graphs known as Tanner graphs, for which a decoding algorithm is known. This decoding algorithm is proven to converge always to a maximum-likelihood decoding if and only if the corresponding Tanner graph is cycle-free. However, we show that cycle-free Tanner graphs cannot support good codes, because of their weak structure. This weakness has a direct affect on codes of Tanner graphs with cycles, which we analyze as well.

"כל אלה חברתי בעזר ה׳ יתברך החונן לאדם דעת."

"לֹא הַבַּיְשָׁן לָמֵד, וְלֹא הַקַּפְּדָן מְלַמֵּד." -פרקי אבות ב:ו

To my wife and my whole family

for that which words cannot convey.

Acknowledgments

No matter how this thesis is attributed, its contents are actually the fortunate product of a concerted effort by a large, multi-national team of mentors over a twenty-six year period. Often their work involved significant personal sacrifice and little material reward. Though I can acknowledge but a fraction of the members of this team, I am indebted to them all.

First, I would like to thank the computer science department and the communication group at the University of Illinois for their support and environment. In addition, the Computational Science and Engineering program provided the computational power for many of the simulation results in this thesis. Drawn from these groups, my thesis committee provided invaluable advice and suggestions throughout this protracted process, and I would like to individually thank Professors Bruce Hajek, Leonard Pitt, Edward Reingold, and Alexander Vardy.

I was particularly fortunate to have as advisors Professors Alexander Vardy and Edward Reingold. Professor Vardy is clearly a master of his craft and I have benefited tremendously from observing him and conducting research with him. Just as importantly, he permitted me the flexibility of a post-doctorate student in choosing and pursuing research ideas from his ever-abundant list of suggested problems. Professor Reingold also served as a true mentor by providing knowledgeable advise on all aspects of academic life. He expertly oiled bureaucratic wheels, when necessary, and patiently listened and supported the research in this thesis.

I would also like to thank my co-authors, which included Professor Vardy, Professor Tuvi Etzion from the Technion, and Yaron Minsky from Cornell University. In addition, G. David Forney, Jr. and Frank Kschischang provided me with inspiration and background for the lexicode chapter of this thesis, and Jeff Erickson, Amir Khandani, and Ralf Kötter provided stimulating discussions for the Tanner graph chapter of this thesis.

I also benefited greatly from my many colleagues here at the University of Illinois. Dakshi

Agrawal served as my model, being half a year ahead of me; Bill Weeks and Weishi Feng served as my encyclopedic references; and Tanya Berger-Wolf, Mitch Harris, Shripad Thite, and Greg Harfst were my sounding boards for new ideas.

Needless to say, this entire exercise would be at most a dream were it not for my loving family. To my parents, Lazar and Tania Trachtenberg, education has been as important as food in my development, and my father has diligently provided critical feedback during the writing of this work. My two sisters, Dalia and Danna, have provided invaluable emotional support, as have my extended family Ted, Adriane, and Scott Moss.

Finally, I dearly thank my wife Felicia, whose love and caring made this entire endeavor worthwhile.

Table of Contents

List of Tables

Notation

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 1.1a on page 2
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 1.1b on page 2
Page 11	$\mathcal{L}_k^d$	the k-th code with minimum distance d produced by the lexicographic construction	$\mathcal{L}_4^3$ is given in Figure 2.2 on page 11
Page 11	$\langle a\|b\rangle$	the concatenation of a and b	$\langle 111 \mid 010\rangle = 111010$
Page 11	a^i	the concatenation of a with itself i times	$(01)^3 = 010101$
Page 11	f_{lexi}, f_{trelli}, f_{state}, f_{decoding}	generating mappings for lexicodes, trellis-oriented codes, state-bounded codes, and decoding-bounded codes respectively	See Section 2.2

continued on next page...

Notation (continued)

Introduced	*Expression*	*Denotes*	*Example*
Page 13	$\mathfrak{G}_i(f, \mathbb{C})$ or $\mathfrak{G}_i$ or $\mathfrak{G}$	the Generalized Lexicographic Construction; if arguments are supplied, they refer to the i-th iteration of the construction seeded by code $\mathbb{C}$ using the generating mapping $f(\cdot)$	$\mathfrak{G}_3(f_{\text{lexi}}, \mathbb{S}_3)$ is the linear code in Table 2.2 on page 11
Page 14	$\mathbb{S}_d$	the seed code $\{0^d, 1^d\}$ for a particular generalized lexicographic construction	$S_3 = \{000, 111\}$
Page 15	$R(v)$, $\mathrm{L}(v)$	the rightmost and leftmost (respectively) significant bit of a bit sequence $v = (v_1, v_2, v_3, \ldots, v_n)$	$R(0101) = 4$, $\mathrm{L}(0101) = 2$
Page 22	$\kappa_v(l)$ or $\kappa(l)$	the companion of coset leader l under v; i.e. the coset leader of the coset containing $l+v$; v is omitted in context	$\kappa_{0110}(0101) = 0011$ for code $\mathbb{C}$ in Figure 1.1b (page 2)
Page 22	$\bar{a}$	the binary complement of a	$\overline{01010} = 10101$
Pages 25,28	n_m, ρ_m	the length and covering radius (respectively) of k-th code in a $\mathfrak{G}$-family	the $(7,3,4)$ lexicode has length n_3=7 and covering radius $\rho_3 = 3$
Page 41	T(H)	the Tanner graph corresponding to the parity-check matrix H	See Example 3.1
Page 42	$\omega(\mathcal{G})$	the number of connected components in $\mathcal{G}$	For any tree, $\omega(\mathcal{G}) = 1$
Page 42	$\mathrm{wt}(M)$	the Hamming weight (i.e. number of nonzero entries) of a matrix M	$\mathrm{wt}(I_n) = n$ for the rank n identity matrix I_n
Page 43	$\mathcal{E}_n$	the even-weight $(n, n-1, 2)$ code, whose parity-check matrixs consists of a single all-1 vector	$\mathcal{E}_2$ has vectors 00 and 11

Chapter 1

Introduction

Coding theory is a branch of communications concerned with the design and evaluation of efficient signaling schemes for reliable data transmission and storage. Notable examples of the application of coding theory include: telephone-line modems [37], where increasing transmission speeds introduce high levels of noise; compact-disk recorders [31], in which error is inherent in the production process; and deep-space probes [38], in which large lag times confound the problems associated with transmission error. All these applications are based on a vast body of basic research in coding theory, see for instance [5, 7, 42] and over 3000 references therein.

It was shown by Shannon [49] that it is possible to transmit information over a noisy channel with arbitrarily small probability of error, at rates up to the capacity of the channel. Reliable transmission of information over noisy channels requires the use of error-correcting codes which encode input in such a way that errors can be detected and corrected at the receiving site.

Thus one of the major problems in coding theory is the design of error-correcting codes that minimize the probability of error on one hand, while maximizing the information rate on the other hand. Another problem of primary importance in applications is the design of efficient decoding algorithms which, given an error-corrupted sequence observed at the output of a noisy channel, quickly reconstruct the most likely transmitted codeword. Both problems are known to be inherently difficult [61] and in fact the latter problem is NP-hard [5] in the general case.

Our approach to both of these problems involves the study of decoding techniques based on certain graphs. We first develop codes which have a low probability of error and a reasonably high information rate, and can be decoded efficiently by means of a *trellis*. The trellis may be thought of as a constrained finite-state automaton; it was originally introduced by Forney [32] in 1967 to

0	0	0	0	0	0
0	0	1	2	1	2
1	2	1	2	0	0

(a)

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

(b)

Figure 1.1: A sample $[n = 6, M = 3, d = 4]$ ternary non-linear and a $(n = 6, k = 2, d = 4)$ binary linear code.

explain the Viterbi decoding algorithm. We then consider a generalization of trellises, known as Tanner graphs [55, 65], in an attempt to further understand the efficiency of iterative decoding.

1.1 Background

Formally, a q-ary $[n, M, d]$ error-correcting code is a set of M vectors over the finite field $\mathbb{F}_q^n$ with the property that any two vectors in the code are separated by a Hamming distance of no less than d. The Hamming distance between two vectors is simply the number of positions in which the vectors differ. For instance, the vectors 1011 and 1002 differ in their two last positions, and are therefore separated by a Hamming distance of 2.

It is often useful to restrict our attention to vectors which form a linear subspace of $\mathbb{F}_q^n$. Thus, a q-ary (n, k, d) linear code is a subspace of $\mathbb{F}_q^n$ of dimension k with the property that any two vectors in the subspace are separated by a Hamming distance no less than d. Figure 1.1 depicts a ternary non-linear code and a binary linear code.

We will typically specify a linear code by its set of basis vectors (also known as *generators*), which form the rows of a generator matrix G. A parity-check matrix for a linear code is a matrix H with the property that $GH^T = 0$ for some generator matrix G of the code. For example, the linear code in Figure 1.1 has parity-check matrix

$$H = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{1.1}$$

as can be verified explicitly.

Linear codes impose a natural coset-structure on their ambient vector spaces. Specifically, given a linear code $\mathbb{C} \subseteq \mathbb{F}_q^n$, we can say that two vectors $v_1, v_2 \in \mathbb{F}_q^n$ are in the same coset if and only if $v_1 - v_2 \in \mathbb{C}$. Moreover, we can represent each coset by a *coset leader* which is arbitrarily chosen among the lowest weight vectors in a coset. For example, some cosets of the linear code in Figure 1.1 are:

$$\begin{aligned} C_0 &= \{\mathbf{000000}, 001111, 111100, 110011\} \\ C_1 &= \{\mathbf{000001}, 001110, 111101, 110010\} \\ C_2 &= \{\mathbf{000011}, 001100, 111111, 110000\} \end{aligned}$$

The coset leaders are shown in bold face. In general, for a linear code of dimension k, there will be 2^{n-k} coset leaders. These cosets can be used in decoding, because e is the coset leader of a corrupted vector $c + e$ whenever c is a codeword and the Hamming weight of e is below the correctable error rate of the code. More importantly, however, they will be used to generate and study lexicodes in Chapter 2.

To use an (n, k, d) binary linear code over a communications channel, a user would split his input into blocks of k bits each and encode each block b by multiplying it by the generator matrix G. The resulting n-bit vector bG is sent over the channel giving a transmission rate of k/n. The receiver then decodes the received error-corrupted vector $bG + e$ to the nearest vector v in the code. The vector v will be equal to bG as long as the error e contains no more than $\lfloor \frac{d-1}{2} \rfloor$ bits. Figure 1.2 depicts this transmission model. The type of decoding we have described is also known as maximum-likelihood decoding because it decodes to the most likely transmitted input. Several descriptions of this communications model may be found in [42], [7], and [4].

Often when we talk about a family of linear error-correcting codes, we shall be interested in the asymptotic tendency of the code. Specifically, we would like that, as the length n increases, the error-correction capability of the code likewise increases. It has been proven that there exist

$$\boxed{10 \quad 11} \longrightarrow \boxed{\begin{matrix} [1\ \ 0]\begin{bmatrix}0&0&1&1&1&1\\1&1&1&1&0&0\end{bmatrix} = [0\ \ 0\ \ 1\ \ 1\ \ 1\ \ 1] \\ \\ [1\ \ 1]\begin{bmatrix}0&0&1&1&1&1\\1&1&1&1&0&0\end{bmatrix} = [1\ \ 1\ \ 0\ \ 0\ \ 1\ \ 1] \end{matrix}}$$

$$\xrightarrow{\text{noise}} \boxed{\begin{matrix}[\mathbf{1}\ \ 0\ \ 1\ \ 1\ \ 1\ \ 1]\\ [1\ \ 1\ \ 0\ \ \mathbf{1}\ \ 1\ \ 1]\end{matrix}} \longrightarrow \boxed{10 \quad 11}$$

Figure 1.2: The transmission model: an input message 1011 is split into blocks of 2 bits each, and then encoded by the generator matrix of the linear code in Figure 1.1. It is transmitted over a noisy channel (errors shown in bold face) and then decoded to the nearest codeword and back into the message word 1011.

families of linear codes which achieve the Gilbert-Varshamov bound [42], namely

$$\frac{k}{n} \geq 1 - H_2\left(\frac{d}{n}\right). \tag{1.2}$$

Here $H_2(\cdot)$ refers to the binary entropy function

$$H_2(p) = -p\log(p) - (1-p)\log(1-p).$$

On the other hand, the McEliece, Rodemich, Rumsey, and Welch [4] provide a well known upper bound on the parameters of linear codes

$$\frac{k}{n} \lesssim H_2\left(\frac{1}{2} - \sqrt{\frac{d}{n}\left(1 - \frac{d}{n}\right)}\right). \tag{1.3}$$

Thus, the asymptotically optimal codes are somewhere in the region between these two bounds, as depicted in Figure 1.3. For our purposes, we shall consider a linear code *asymptotically good* if it asymptotically bounds the relative minimum distance d/n away from 0, for information rates $k/n < 1$ and length n going to infinity. Our definition of "asymptotically good" is deliberately weak, yet we shall show that certain families of codes cannot even achieve this weak definition.

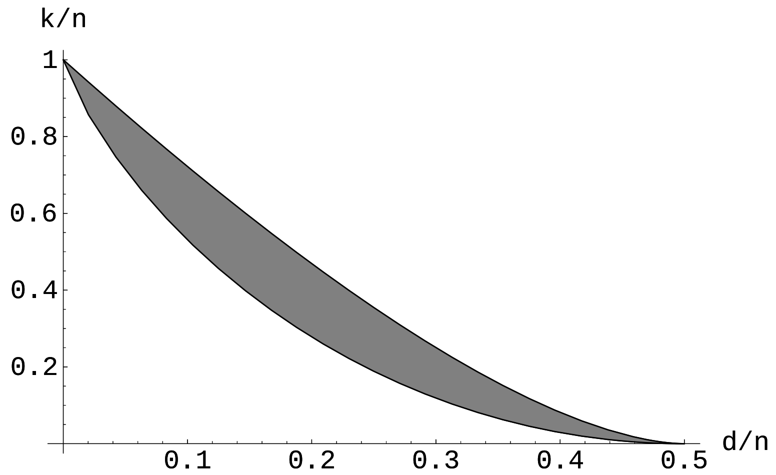


Figure 1.3: A plot of the McEliece-Rodemich-Rumsey-Welch bound (upper) and the Gilbert-Varshamov bound (lower) on the parameters of linear codes. The information rate $R = k/n$ is plotted versus the relative minimum distance $\delta = d/n$, for asymptotically large n. The gray area marks the gap between the known upper and lower bounds.

1.1.1 Trellis decoding

Maximum-likelihood decoding is an obvious computational bottleneck for error-correction. One manner of implementing this decoding involves the use of a trellis, which is a time-indexed graph that represents a given linear code. More formally, a trellis $T = (V, E)$ of depth n is a finite, directed, edge-labeled graph with the following properties:

1. Each vertex $v \in V$ has an associated depth $d_v \in \{0, 1, 2, 3, \ldots n\}$.

2. Each edge $e \in E$ connects a vertex at depth i to a vertex at depth $i + 1$, for some i.

3. There is one vertex, called the *root*, at depth 0 and one vertex, called the *toor*, at depth n.

A trellis for a code establishes a one-to-one correspondence between codewords and paths from the root to the toor. One may consider a trellis to be a definite finite automaton with one start state and one finish state and no loops, so that a code represented by the trellis is precisely the language of the trellis. Figure 1.4 gives an example of a trellis for the linear code in Figure 1.1.

Given a received, possibly error-corrupted word, we may associate bit-error probabilities with weights on each edge in the trellis so that the problem of maximum-likelihood decoding reduces to the problem of finding the minimum-weight path from the root to the toor in the trellis. This can

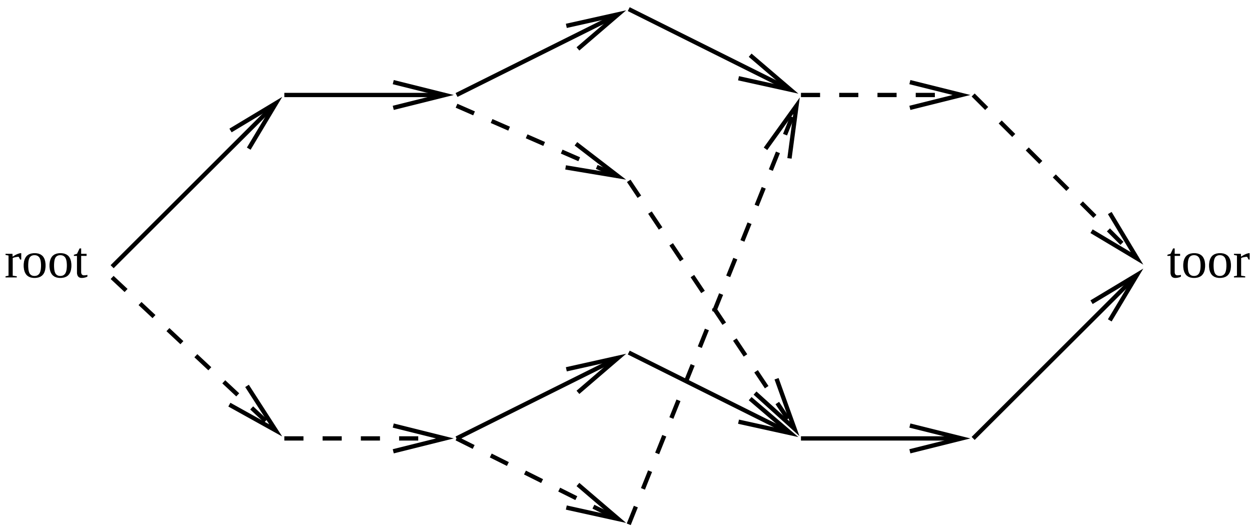


Figure 1.4: A trellis for the linear code in Figure 1.1. Solid lines represent transition on 1 bits and dashed lines represent transitions on 0 bits. Any path from the root to the toor spells out a codeword.

be accomplished by a dynamic-programming algorithm known as the Viterbi algorithm [63] which iteratively finds the most likely prefixes up to depth i.

The Viterbi algorithm requires $|E|$ multiplications and $|E|-|V|+1$ additions for the maximum-likelihood decoding of a given vector. By making use of the depth-structure of the trellis, the Viterbi algorithm is able to easily beat even the $O(|E| + |V| \log |V|)$ Fibonacci-heap implementation of Dijkstra's algorithm and the DAG-Shortest-Path algorithm for the same task [15]. Moreover, like the Floyd-Warshall algorithm [15] but faster, the Viterbi algorithm is fairly general and can be applied to any label alphabet with a semiring structure [62].

Though trellises are found in many common decoding implementations, Lafourcade and Vardy [36] have shown that their decoding complexity grows exponentially in the length of the code for asymptotically good codes. For this reason, alternative graph structures have been proposed for decoding, such as Tanner graphs [55] and their generalization to factor graphs [24].

1.1.2 Tanner graphs

A Tanner graph for a binary (n, k, d) code $\mathbb{C}$ is a bipartite graph whose adjacency matrix is the parity-check matrix of $\mathbb{C}$. The left part of the graph consists of *site nodes* which correspond to symbol bits of the code, and the right part of the graph consists of *check nodes* which correspond

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

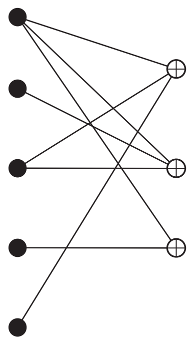


Figure 1.5: H is the parity-check matrix for a $(5, 2, 3)$ binary, linear code. The accompanying figure is its Tanner graph, in which the left nodes correspond to symbol bits and the right nodes correspond to parity-checks.

to parity checks in the graph. Thus the number of edges in any Tanner graph for a linear code $\mathbb{C}$ of length n is $O(n^2)$, so that the graph's complexity grows quadratically with length, regardless of the code. Figure 1.5 gives a parity-check matrix and corresponding Tanner graph for a sample binary, linear code.

A generalized version of the Viterbi algorithm, known as the *min-sum* algorithm [66], might allow for decoding with Tanner graphs. This is an iterative algorithm which tries to spread bit-error information throughout the Tanner graph. Thus, check nodes repeatedly *minimize* error among legal site configurations neighboring them, and site nodes repeatedly *sum* errors from their neighbors. This iterative algorithm provenly converges [1, 55, 66] to a maximum-likelihood decision for cycle-free Tanner graphs, although it does not necessarily converge or decode correctly if the graph has a cycle. Thus, if we can represent $\mathbb{C}$ by a Tanner graph *without cycles*, then maximum-likelihood decoding of $\mathbb{C}$ can be achieved in time $O(n^2)$ using the min-sum algorithm.

1.2 Organization

This dissertation is organized as follows. In Chapter 2 we study trellis decoding using the Viterbi algorithm. More precisely, we are interested in *designing* codes that can be efficiently decoded using the Viterbi algorithm on a trellis. We study a generalization of the greedily-constructed "lexicographic codes". This generalization enables the design of codes which meet specific decoding constraints, such as bounds on the time and space complexity alloted for decoding. We also determine a theoretical relationship between cosets of these codes, which allows us to explicitly compute their generator matrices and to improve upon known bounds of their parameters. This research extends and improves earlier work in [56]. In addition, a version of this chapter has been published as a joint work with Alexander Vardy in the *Proceedings of the Conference of Information Sciences and Systems* [60] and is being prepared for submission to the *IEEE Transactions on Information Theory* [58].

Chapter 3 focuses on Tanner graphs in an attempt to get around the exponential growth in decoding complexity which is inherent in trellis decoding. It was mentioned in the background section that decoding can be done very quickly for codes with cycle-free Tanner graphs. However, both intuition and experimentation (cf. [40]) suggest that powerful codes cannot be represented by cycle-free Tanner graphs. The notion that cycle-free Tanner graphs can support only weak codes is by now widely accepted. In Chapter 3 we make this "folk knowledge" precise.

Specifically, we provide a rigorous answer to the question: "Which codes can have cycle-free Tanner graphs?" We consider extensions to this question and show that, as with trellis-decoding, asymptotically good codes will require asymptotically increasing numbers of cycles. The majority of this chapter was presented as a joint work with Tuvi Etzion and Alexander Vardy at the *1998 International Symposium on Information Theory* [19] and has appeared in the *IEEE Transactions on Information Theory* [20] and as a technical report [18].

In Chapter 4 we summarize the results of this dissertation and examine possible future directions for this research. We consider several problems which are still open and provide natural extensions for this work. In addition, we briefly describe unrelated research into full-rank tilings [59] and database reconciliation [44, 57], which was developed concurrently with this work. Finally, the appendices include results of implementing some algorithms from Chapter 2 in addition to some more technical aspects of proofs from Chapter 3.

Chapter 2

Lexicographic Codes

Lexicographic codes, or *lexicodes* for short, are greedily constructed codes which were introduced by Conway and Sloane in [13, 14]. Lexicodes have surprisingly good encoding parameters and include, among other famous optimal codes, the Hamming codes, the Golay code, and certain quadratic residue codes [14, 42]. Several authors [9, 14] have proved that lexicodes are linear, and comparison with optimal linear codes of the same length and dimension shows that lexicodes are usually within one of the optimal minimum distances [14] and often exhibit the smallest known covering radius [27]. Hence, lexicodes may be regarded as a heuristically good "approximation" to the optimal codes. Brualdi and Pless [9] have examined a similar generalization of lexicodes, known as greedy codes, and presented certain bounds on their parameters.

In this chapter, we examine the theoretical underpinnings of lexicodes and investigate various generalizations. Our intent is to graft desired decoding properties onto the heuristically good features of lexicodes. In this way, we develop a means for designing good codes for specific tasks. For example, we can fashion good codes which maintain a desired dimension, error-correction capability, and memory constraint.

Section 2.1 opens the chapter with a definition of the *lexicographic construction*, which iteratively constructs generator matrices for the family of minimum distance d lexicodes, for any d. The lexicographic construction is, in turn, a special case of the generalized lexicographic construction, which we abbreviate as the $\mathfrak{G}$-construction. This generalized construction encompasses a variety of code families which graft different properties onto lexicode features.

Section 2.2 is devoted to various instances of the $\mathfrak{G}$-construction. Specifically, Section 2.2.1 analyzes the role of lexicodes as one of these instances and describes tools for efficiently constructing them. In Section 2.2.2 we demonstrate that a simple modification of the lexicode generator produces "trellis-oriented" lexicodes which locally minimize trellis complexity. These codes exhibit the same "approximately optimal" features of lexicodes but often have a much lower trellis decoding complexity, in some cases reaching the lowest trellis complexity known for their length, rate, and minimum distance. This modification provides a natural heuristic for improving upon the trellis-complexity of a given code. Finally, in Section 2.2.3 we examine methods for designing codes with various trellis characteristics, such as constrained state complexity or constrained Viterbi decoding complexity. These methods are particularly useful for *VLSI* decoding implementations, where layering bounds favor a small trellis state complexity, and mobile communications, where portability severely limits power consumption.

Section 2.3 investigates the coset relationship which supports $\mathfrak{G}$-construction. This relationship establishes bounds on the parameters of the codes produced by the construction and suggests a natural algorithm for computing these codes. The bounds established, though loose, are asymptotically tighter than the lexicode bounds of Brualdi and Pless [9]. We also use the coset relationship to bound the computation time of the $\mathfrak{G}$-construction in Section 2.4.

Finally, in Section 2.5 we discuss various applications of the $\mathfrak{G}$-construction and present our conclusions. Appendix A lists simulation results for the various algorithms described in this chapter.

2.1 The lexicographic construction

A q-ary, length n, minimum distance d lexicode is traditionally defined constructively based on a lexicographic (i.e. dictionary) ordering of vectors in which, for example, 01111 comes before 10000. The construction starts with the set $\mathcal{S} = \{\mathbf{0}\}$ and greedily adds, until exhaustion, the lexicographically earliest vector (in the vector space $\mathbb{F}_q^n$) whose Hamming distance from $\mathcal{S}$ is at least d.

For example, the codewords of the binary lexicode (i.e. $q = 2$) of length $n = 3$ and minimum distance $d = 2$ are marked by a $\bullet$ in Figure 2.1 and would be computed left-to-right across the figure.

This greedy construction always generates a linear code [9, 14]. Thus, we may completely describe a dimension k lexicode by finding its k generators using what we call the *lexicographic*

000	001	010	011	100	101	110	111
•			•		•	•	

Figure 2.1: A simple $n = 3, d = 2$ binary lexicode.

construction. This construction starts with the zero code $\mathcal{L}_0^d = \mathbf{0}$ and greedily adds the lexicographically earliest generator of distance d from the code; k such iterations form the dimension k code $\mathcal{L}_k^d$. Figure 2.2 demonstrates this construction for $d = 3$, resulting in a $(7, 4, 3)$ binary code.

0000**111**
00**11**001
0**1**01010
1001011

Figure 2.2: Generator matrix for the dimension 4, minimum distance 3 binary code $\mathcal{L}_4^3$. The generator padding for each iteration is marked in bold.

We may understand the lexicographic construction more analytically by making use of the covering radius of each intermediate code $\mathcal{L}_i^d$ in the iteration. The covering radius of a length n code is the smallest integer ρ with the property that Hamming balls of radius ρ centered at codewords of the code will cover every vector of $\mathbb{F}_q^n$. In other words, ρ is the maximum distance of a vector in $\mathbb{F}_q^n$ from the code. As an example, one can readily see that the binary lexicode in Figure 2.1 has a covering radius of 1 because every vector in $\mathbb{F}_2^3$ is at most of Hamming distance 1 from a code vector.

An iteration of the lexicographic construction on an intermediate code $\mathbb{C}$ whose covering radius is ρ and minimum distance is d can thus be understood as the addition of a generator vector:

$$\langle 1^{d-\rho} \mid f_{\text{lexi}}(C) \rangle$$

where $1^{d-\rho}$, known as the *generator padding*, has the usual meaning of $(d - \rho)$ successive 1's, and the function $f_{\text{lexi}}(\mathbb{C})$ returns the lexicographically earliest vector of distance ρ from $\mathbb{C}$. As is customary, we use $\langle \cdot | \cdot \rangle$ to denote concatenation.

It is not surprising that the non-trivial[1] linear codes generated by the lexicographic construction

[1]For our purposes, a trivial linear code is one whose generator matrix contains an all-zero column. Trivial codes can be simply reduced to non-trivial codes by deleting the zero columns from their generator matrix.

described above are precisely the lexicodes.

Theorem 2.1 *Over a finite field $\mathbb{F}_q$, a non-trivial (n,k,d) code $\mathbb{C}$ is a lexicode if and only if it is produced by the lexicographic construction, namely $\mathbb{C} = \mathcal{L}_k^d$.*

Proof. We first prove that $\mathcal{L}_k^d$ is always a lexicode, by induction on k for an arbitrary, fixed d. For the base case it is clear that $\mathcal{L}_1^d = \left\{0^d, 1^d\right\}$ is a $(d,1,d)$ lexicode. Now assume as an inductive hypothesis that $\mathcal{L}_k^d$ is the same as the (n,k,d) lexicode $\mathbb{C}_k$. From the definition of the lexicographic construction, $\mathcal{L}_{k+1}^d$ has parameters $(n+\rho, k+1, d)$, where ρ is the covering radius of $\mathcal{L}_k^d$. Consider the $(n+\rho, k', d)$ lexicode $\mathbb{C}_{k'}$ constructed by repeatedly choosing the appropriate lexicographically-earliest vectors in $\mathbb{F}_q^{n+\rho}$. Clearly, we generate $\mathbb{C}_k$ in the process of this construction, so that $\mathbb{C}_k \subseteq \mathbb{C}_{k'}$ and, by the inductive hypothesis, $\mathcal{L}_k^d \subseteq \mathbb{C}_{k'}$. Moreover, the vector

$$v = \left\langle 1^{d-\rho} \mid f_{\text{lexi}}(\mathbb{C}_k)\right\rangle$$

is the lexicographically earliest vector of distance ρ from $\mathcal{L}_k^d$. Thus, it must be the case that $v \in C_{k'}$ and, since $C_{k'}$ is linear, $\mathcal{L}_{k+1}^d \subseteq \mathbb{C}_{k'}$.

$000\ldots0$	$\mathcal{L}_k^d$
$111\ldots1$	w

Figure 2.3: The structure of the (k+1)-th lexicode $\mathcal{L}_{k+1}^d$. There are $d-\rho$ ones on the left-hand side. Since the covering radius of $\mathcal{L}_k^d$ is ρ, any n bits (i.e. the vector w) must be at distance $\leq \rho$ from the n rightmost bits of $\mathcal{L}_{k+1}^d$.

In addition, any vector $v \in \mathbb{F}_q^{n+\rho}$ that is in $\mathbb{C}_{k'}$ but not in $\mathcal{L}_{k+1}^d$ would necessarily have its n right-most bits at distance $\leq \rho$ from $\mathcal{L}_{k+1}^d$, and its other bits at distance $\leq \lceil (d-\rho)/2 \rceil$ from $\mathcal{L}_{k+1}^d$, as we can see in Figure 2.3. Noting that our non-triviality assumption implies that $\rho < d$, we may now use the triangle inequality to see that the distance from v to $\mathcal{L}_{k+1}^d$ (and consequently also $\mathbb{C}_{k'}$) must be at most

$$\rho + \lceil (d-\rho)/2 \rceil \;=\; \lceil (d+\rho)/2 \rceil \;<\; d.$$

This contradicts our constructive definition of $\mathbb{C}_{k'}$, so it must be that $\mathcal{L}_{k+1}^d = \mathbb{C}_{k'}$.

For the converse assertion of the theorem, we need to show that a non-trivial (n, k, d) lexicode $\mathbb{C}$ can be constructed by the lexicographic construction. The code $\mathcal{L}_k^d$ produced by k iterations of the minimum-distance d lexicographic construction is clearly a lexicode, from the first part of the proof. Since all distance d lexicodes are ordered by inclusion, as we saw in the first part of this proof, we may conclude that either $\mathbb{C} \subseteq \mathcal{L}_k^d$ or $\mathcal{L}_k^d \subseteq \mathbb{C}$. However $\mathcal{L}_k^d$ and $\mathbb{C}$ are both non-trivial dimension k codes, so it follows that they must be equal. ∎

Theorem 2.1 allows us to bypass the codeword-by-codeword lexicode construction, and instead directly compute the generator matrix of a desired lexicode.

2.1.1 Generalization

The lexicographic construction may be extended to produce codes with desired characteristics using the *generalized lexicographic construction*, abbreviated $\mathfrak{G}$-construction. The $\mathfrak{G}$-construction replaces the "lexicographically earliest" heuristic used in building lexicodes with an arbitrary function. This allows us to generate arbitrary greedy codes in which various properties are grafted upon the good code parameters of the lexicodes. More formally,

Definition 1 *The generalized lexicographic construction is boot-strapped with a linear (n, k, d) seed code $\mathbb{C}$ and iteratively constructs the family of codes*

$$\mathfrak{G}(f, \mathbb{C}) \;=\; \{\mathfrak{G}_i(f, \mathbb{C}) \,:\, i \in \mathbb{Z}\}$$

using a mapping from codes to vectors:

$$f(\cdot) : \mathbb{C} \subseteq \mathbb{F}_q^* \longmapsto v \in \mathbb{F}_q^n \tag{2.1}$$

where we define

$$\mathbb{F}_q^* = \bigcup_{i=1}^{\infty} \mathbb{F}_q^i.$$

The construction follows the scheme:

- $\mathfrak{G}_0(f, \mathbb{C})$ *is trivially the code* $\mathbb{C}$
- $\mathfrak{G}_i(f, \mathbb{C})$ *is computed by adding to* $\mathfrak{G}_{i-1}(f, \mathbb{C})$ *the generator*

$$\left(1^{\Delta_i} \;\middle|\; \lambda_i\right) \tag{2.2}$$

where $\lambda_i = f(\mathfrak{G}_{i-1})$ *and* $\Delta_i = d-$*minimum Hamming distance from* λ_i *to the code* $\mathfrak{G}_{i-1}(f, \mathbb{C})$.

We will call $f(\cdot)$ the *generating mapping* of the construction. The familiar lexicode family of minimum distance d is thus the simple special case $\mathfrak{G}(f_{\text{lexi}}, \mathbb{S}_d)$, where we use the trivial seed code $\mathbb{S}_d \stackrel{\text{def}}{=} \{0^d, 1^d\}$. The code in Figure 2.2 is $\mathfrak{G}_3(f_{\text{lexi}}, \mathbb{S}_3)$, as can be verified by hand.

The $\mathfrak{G}$-construction serves to homogenize its seed code, since each iteration builds a subcode with the same minimum distance. Thus, there are many linear codes which do not fit into the mold of this construction. For example, the linear code given by the following generator matrix cannot be constructed with the $\mathfrak{G}$-construction:

$$\begin{matrix} 1110000 \\ 0001111 \end{matrix}$$

For sake of simplicity we shall concern ourselves only with binary codes in the remainder of this chapter, though the extension to q-ary codes is fairly straighforward.

2.2 Instances of the $\mathfrak{G}$-construction

The algorithm for computing the $\mathfrak{G}$-construction is technically complicated and will be relegated to Section 2.3. A generating mapping uniquely determines a corresponding family of codes, known as a $\mathfrak{G}$-family, produced by the $\mathfrak{G}$-construction. In this section we shall analyze various generating mappings that can be used in the design these families for specific decoding needs.

2.2.1 Lexicodes

As mentioned in the previous section, the mapping f_{lexi} generates the lexicodes. The computation of f_{lexi} is handled by a simple greedy algorithm, presented as Method 2.2, which operates in linear time and space, subject to appropriate knowledge of the input code. The speed with which we can compute f_{lexi} permits efficient generation of lexicodes, as we shall see later in Section 2.4.

In order to compute f_{lexi} we will first transform the generator matrix of the code into a minimal span generator matrix (`MSGM`) form, in which the sum of the spans of the generators is minimized. Adapting the notation in [43], the span of a binary n-vector $x = (x_1, x_2, x_3, \ldots, x_n)$ is $\mathrm{R}(x) - \mathrm{L}(x)$, where $\mathrm{R}(\cdot)$ and $\mathrm{L}(\cdot)$ are the rightmost (i.e. largest) and leftmost (i.e. smallest) index i, respectively, such that $x_i \neq 0$. Thus, the span of the vector $x = (0001001100)$ is $\mathrm{R}(x) - \mathrm{L}(x) = 8 - 4 = 4$. We can efficiently transform any matrix into `MSGM`-form using the greedy algorithm in [43]. We note that two vectors of an `MSGM` cannot have their leftmost or their rightmost index in common, or else they may be added to produce a generator matrix with shorter span.

Given an `MSGM` for a code and a set of coset representatives $\mathcal{V}$, Method 2.2 computes the lexicographically earliest vector among the cosets represented in $\mathcal{V}$.

Method 2.2 *Consider a set of vectors $\mathcal{V}$ representing cosets of a code $\mathbb{C}$ with length n. Let G be an* `MSGM` *for $\mathbb{C}$ whose generators are in lexicographically increasing order. The following greedy method computes the lexicographically earliest vector among the represented cosets in time $O(n|V|)$ and space $O(1)$.*

1. **for** $v = (v_1, v_2, v_3, \ldots, v_n) \in \mathcal{V}$ **do**
2.   **for** i from n downto 1
3.   **if** $v_{\mathrm{L}(G_i)}$ is a 1 then
4. $v \leftarrow v + G_i$
5.   **store** the modified v;
6. **among** all stored v, return the lexicographically earliest

This method looks for the generators whose left-most 1-bit corresponds to 1-bits in vectors $v \in V$. Figure 2.4 demonstrates this method with the set $\mathcal{V} = \{1010000\}$ on the $(7, 4, 3)$ code described in Figure 2.2. For this case, the vector 0001011 is the lexicographically earliest vector in the same coset as 1110000. Note that the ordering of the generators in the `MSGM` is significant and that different orderings might not yield the lexicographically earliest vector. Thus, if generators G_2

and G_3 are switched, then Method 2.2 yields 0010010, which is not the lexicographically earliest vector desired.

$$\begin{array}{rl} G_1 = & \boxed{0000111} \\ G_2 = & \boxed{0011110} \\ G_3 = & \boxed{0110100} \\ G_4 = & \boxed{1001000} \end{array} \qquad \begin{array}{rl} v: & \boxed{1110000} \\ v \leftarrow v + G_4: & \boxed{0111000} \\ v \leftarrow v + G_3: & \boxed{0001100} \\ v \leftarrow v + G_1: & \boxed{0001011} \end{array}$$

Figure 2.4: Method 2.2 applied to the code in Figure 2.2 with initial condition $\mathcal{V} = \{1110000\}$. The various values taken on by v during the computation are shown on the right-hand region.

We will now prove the correctness and complexity bounds of Method 2.2. In our applications, $\mathcal{V}$ will typically be the set of coset leaders with maximum distance from $\mathbb{C}$, in which case Method 2.2 computes precisely $f_{\text{lexi}}(\mathbb{C})$.

Proof of Method 2.2. We first show correctness of the method by proving that, for each $v \in \mathcal{V}$, lines 2-5 compute the lexicographically earliest vector in the same coset as v. This way, line 6 in the method will return the lexicographically earliest vector among all the cosets represented.

We know from [43, Thm 6.11 and Lemma 6.7] that the rows of G have the *predictable support property*:

$$\operatorname{span}\left(\sum_{j \in J} G_j\right) = \bigcup_{j \in J} \operatorname{span}(G_j) \tag{2.3}$$

for every subset $J \subseteq \{1, 2, 3, ..., n\}$.

Now suppose that v_{best} is the lexicographically earliest vector in the same coset as v, but that instead v_{stored} is the vector stored on line 5 of the method. Our goal will be to show that the difference between these two vectors, denoted v_{diff}, is in fact 0.

Clearly v, v_{best}, and v_{stored} are necessarily in the same coset of $\mathbb{C}$. Moreover, since v and v_{stored} are in the same coset, v_{diff} must be a code word of $\mathbb{C}$ so that we can write (for an appropriate set J_{diff}):

$$v_{\text{diff}} = \sum_{j \in J_{\text{diff}}} G_j \tag{2.4}$$

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

$$\text{span}(v_{\text{diff}}) = \bigcup_{j \in J_{\text{diff}}} \text{span}(G_j) \tag{2.5}$$

Assume (for sake of contradiction) that $L(v_{\text{diff}}) = k$, meaning that the left-most 1 bit of v_{diff} occurs at the k-th index. This implies that, starting from the left-most bit, v_{best} and v_{stored} are identical until the k-th index, on which they differ. Since, by definition, v_{best} comes lexicographically before v_{stored}, it must be that v_{best} has a 0 and v_{stored} a 1 at the k-th index. However, (2.5) then implies that there must be some generator vector G_j whose left-index is k, contradicting our constructive generation of v_{stored}. Thus, the left-index of v_{diff} could not possibly be at location k, for any k, so that v_{diff} must be 0 and correctness of the method is proved.

The space bound for this method follows straightforwardly, as at any point in the algorithm we need to actively store only the lexicographically earliest vector among those determined on line 5 so far. The time bound follows from the fact that lines 3-5 are each computable in constant time, giving a running time of $O(n|V|)$ for lines 1-5; Line 6 can be computed with constant overhead on-line, as each vector is stored. ∎

Method 2.2 provides a fast way of computing the lexicographic generating mapping. Appendix A.1 gives computed parameters for many lexicodes that were thus computed. This list is more exhaustive than what is currently available in the literature [9, 14].

2.2.2 Trellis-oriented lexicodes

We now consider a different generating mapping for the $\mathfrak{G}$-construction which will generate families of codes oriented towards a reduced decoding trellis.

There is a unique, minimal [45], trellis for an arbitrary linear block code. Known as the `BCJR` [3] trellis, it has no more edges or vertices at each time index than any other trellis for the code and can be constructed fairly easily [35, 43]. It is shown in [35, 43] that the minimal span generator matrix (`MSGM`) for a linear code, in which the sum of the spans of the binary generators is minimized, reflects the properties of the corresponding `BCJR` trellis. Namely, if G is an `MSGM` of a code $\mathbb{C}$, comprised of the generators $g_1, \ldots, g_k$, then the number of vertices V_i and edges E_i in the corresponding `BCJR` trellis is given by:

$$\begin{aligned} |V_i| &= 2^{k-p_i-f_i} \\ |E_{i,i+1}| &= 2^{k-p_i-f_{i+1}} \end{aligned} \tag{2.6}$$

where p_i and f_i are the dimensions of the past and future subcodes for the i-th index of $\mathbb{C}$. These dimensions may be computed for $i = 0, 1, \ldots, n$ as follows [43]:

$$\begin{aligned} p_i &= |\{j : \mathrm{R}(g_j) \leq i\}| \\ f_i &= |\{j : \mathrm{L}(g_j) \geq i+1\}| \end{aligned}$$

As before, $\mathrm{R}(\cdot)$ and $\mathrm{L}(\cdot)$ refer to the rightmost and leftmost non-zero indices of a vector, respectively. Since $2|E| - |V| + 1$ steps are required for Viterbi decoding using a trellis [43], the optimal Viterbi decoding complexity for a code can be easily gleaned from its `MSGM`.

With a minor modification of the lexicographic construction we may exploit the above relations to locally minimize two measures of trellis complexity: state complexity, which is the maximum number of states at any time interval of the trellis, and Viterbi decoding complexity. Specifically, $\mathfrak{G}(f_{\text{trelli}}, \mathbb{C})$ is such a family of codes, and it locally minimizes trellis complexity if $f_{\text{trelli}}(\mathbb{C})$ returns the vector with maximum distance from $\mathbb{C}$ whose bitwise reverse is lexicographically earliest. For example, if the vectors $\{1011, 1101, 1110\}$ are at maximum distance from a code $\mathbb{K}$, then $f_{\text{trelli}}(\mathbb{K})$ returns the vector 1110. Because of their locally minimal trellis complexity, these codes may be justly called "trellis-oriented."

Theorem 2.3 *Let $\mathbb{C}$ be a linear code. Among those single-iteration extensions $\mathfrak{G}_1(f, \mathbb{C})$ with shortest length, the generator mapping $f(\cdot) = f_{trelli}(\cdot)$ minimizes the Viterbi decoding complexity and trellis state complexity.*

Theorem 2.3 establishes that $f_{\text{trelli}}(\cdot)$ locally minimizes trellis complexity among local extensions with optimal code parameters. It is possible to improve Viterbi decoding complexity by using a generating mapping which produces a longer extension, but the information rate of the resulting code will be inferior.

Proof. Suppose that G is an MSGM for the (n,k,d) code $\mathbb{C}$ whose past and future subcodes have dimensions p_i and f_i respectively, and whose trellis has vertices V_i and edges $E_{i,i+1}$ at the i-th time interval. Now, consider appending to G the generator $v = \langle 1^{\Delta}|\lambda\rangle$ as in Equation (2.2) of the definition of the $\mathfrak{G}$-construction. The generated code $\mathbb{C}' = \mathfrak{G}_1(f,\mathbb{C})$ will have parameters $(n' = n+d-\mathrm{wt}(\lambda), k+1, d)$. Without loss of generality we may assume that the resulting generator matrix of $\mathbb{C}'$ is an MSGM. If we denote the difference in lengths between $\mathbb{C}'$ and $\mathbb{C}$ by $\nabla n = n' - n$, then a simple analysis shows that $\mathbb{C}'$ will have past and future subcode dimensions:

$$
\begin{aligned}
{p'}_i &= \begin{cases} 0 & \text{if } 0 \le i \le \nabla n \\ p_{i-\nabla n} & \text{if } \nabla n < i < R(v) \\ p_{i-\nabla n}+1 & \text{if } R(v) \le i \le n' \end{cases} \\
{f'}_i &= \begin{cases} k+1 & \text{if } i = 0 \\ k & \text{if } 0 < i \le \nabla n \\ f_{i-\nabla n} & \text{if } \nabla n < i \le n' \end{cases}
\end{aligned}
\tag{2.7}
$$

We may then use (2.6) and (2.7) at each time unit i to compute the number of vertices and edges $(|V_i'|, |E_i'|)$ in the BCJR trellis for $\mathbb{C}'$ based on the vertices and edges $(|V_i|, |E_i|)$ in the BCJR trellis for $\mathbb{C}$:

$$
(|V_i'|, |E_i'|) = \begin{cases} (1,2) & \text{if } i = 0 \\ (2,2) & \text{if } 0 < i < \nabla n \\ (2|V_{i-\nabla n}|, 2|E_{i-\nabla n}|) & \text{if } \nabla n \le i < R(v) \\ (|V_{i-\nabla n}|, |E_{i-\nabla n}|) & \text{if } R(v) \le i \le n' \end{cases}
\tag{2.8}
$$

It is now a simple matter of algebra to compute the difference in Viterbi decoding complexities between the trellises of $\mathbb{C}'$ and $\mathbb{C}$:

$$\begin{aligned}\Delta(Viterbidecodingcomplexity) &= (2|E'| - |V'| + 1) - (2|E| - |V| + 1) \\ &= 3 + 2(\nabla n - 1) + 2\left[\sum_{i=0}^{\mathrm{R}(v)-\nabla n-1} |E_i|\right] - \left[\sum_{i=0}^{\mathrm{R}(v)-\nabla n-1} |V_i|\right]\end{aligned} \tag{2.9}$$

Since $|E_i| \geq |V_i|$ for all i, minimizing the change in Viterbi decoding complexity depends on minimizing $R(v) = R\left(1^{\Delta}|\lambda\right)$, which in turn depends on having the 1 bits of λ as far to the left as possible. On the other hand, λ cannot have weight less than the covering radius ρ of $\mathbb{C}$ if it is to produce an extension of shortest length. Thus, the mapping $f_{\text{trelli}}(\cdot)$ is one of several mappings which meet both criteria for local optimality: minimizing $R(v)$ and having $\text{wt}(\lambda) = \rho$. On the other hand, other intuitive generating mappings, such as picking lexicographically *latest* vectors at maximum distance from the code, are not always locally optimal.

Equation 2.8 also proves that minimizing $\mathrm{R}(v)$ is the appropriate criterion for minimizing state complexity. This is because larger values of $\mathrm{R}(v)$ correspond to doubling more vertices $|V_{i-\nabla n}|$ when generating V_i'. Thus, $f_{\text{trelli}}(\cdot)$ locally minimizes state complexity as well. ∎

Note that (2.8) and (2.9) provides us with a simple alternate method for computing the Viterbi decoding complexity or state complexity of any trivially-seeded $\mathfrak{G}$-code given its `MSGM`.

0000**111**
00**11**100
0**1**10010
1111000

Figure 2.5: Generator matrix for the dimension 4 minimum distance 3 trellis-oriented $\mathfrak{G}$-code. The generator padding for each iteration is marked in bold.

Figure 2.5 depicts the $(7, 4, 3)$ *trellis-oriented* $\mathfrak{G}$-code that was generated similarly to the code in Figure 2.2. In fact, Method 2.2 can be trivially reversed to compute $f_{\text{trelli}}(\cdot)$, without affecting the running time or space:

Method 2.4 *Consider a set of vectors $\mathcal{V}$, and a code $\mathbb{C}$ with `MSGM` G whose generators are in lexicographically increasing order. The following greedy method computes the lexicographically earliest bitwise reversed vector among the represented cosets in time $O(n|V|)$ and space $O(1)$.*

1. **for** $v = (v_1, v_2, v_3, \ldots, v_n) \in \mathcal{V}$ **do**
2. **for** i from 1 to n
3. **if** $v_{\mathrm{R}(G_i)}$ is a 1 then
4. $v \leftarrow v + G_i$
5. **store** the modified v;
6. **among** all stored v, return the lexicographically earliest

The proof of correctness and complexity for Method 2.4 follows trivially from the proof of Method 2.2.

2.2.3 Trellis-bounded lexicodes

It is often useful to bound various parameters of a trellis so that decoding can be efficiently handled by a system with complexity constraints, such as a *VLSI* chip with certain layering constraints or for mobile communications with certain power limits. We consider generating mappings which bound Viterbi decoding complexity and trellis-state complexity. The trellis-state complexity is a strong indicator of the decoding complexity of a code because it asymptotically restricts memory-usage and the number of bifurcations in the trellis.

We may produce $\mathfrak{G}$-families which constrain state complexity by simply restricting the generating mapping to returning only vectors which maintain state complexity bounds. In order to duplicate the nice locally optimal properties of f_{trelli} we similarly design the state-constrained generating mapping f_{state} to return, among all candidate vectors which maintain the state complexity bound, the vector of maximum distance from the given code whose reverse is lexicographically earliest.

We may similarly produce $\mathfrak{G}$-families which constrain Viterbi decoding complexity by using the generating mapping f_{decoding}, which bounds the Viterbi decoding complexity of an extended code by some function $g(k)$ of the dimension k of the underlying code.

The computations of f_{state} and f_{decoding} are straightforward by-products of the mechanism behind Theorem 2.5 (in the next section), which drives the computation of codes in an arbitrary $\mathfrak{G}$-family. Specifically, for any such $\mathfrak{G}$-code, we may quickly derive an `MSGM` from which all the necessary trellis properties may be easily gleaned and suitably constrained. Unfortunately, the worst case (extreme bounding) computation of either of these generating mappings involves exhaustively searching through every coset leader in an attempt to meet the constraint. Appendix A.2 and A.3 demonstrate the effects of bounding various trellis properties.

2.3 Relation between cosets

So far we have addressed the computation of various generating mappings. In this and subsequent sections we will complete our analysis by addressing the theoretical mechanism which supports the $\mathfrak{G}$-construction. Specifically, the $\mathfrak{G}$-construction establishes an important relationship between the coset leaders of the codes produced at successive iterations. This relationship allows us to efficiently compute the coset leaders of many $\mathfrak{G}$-codes (and hence their generator matrices) as well as to determine bounds on their code parameters. We describe this relationship by first associating a *companion* with each coset leader.

Definition 2 *The companion of a coset leader l, with respect to a vector $v \in \mathbb{F}_2^n$, is the leader of the coset containing $l + v$. It is denoted $\kappa_v(l)$, or $\kappa(l)$ in context.*

We show that one iteration of the lexicographic construction produces a code whose coset leaders are the lower-weight vectors between $\langle a|u\rangle$ and $\langle \bar{a}|\kappa(u)\rangle$, for each vector a of appropriate size and coset leader u in the original code. Here $\bar{a}$ refers to the complement of a.

Theorem 2.5 *Consider an (n, k, d) code $\mathbb{C}$ with a fixed set $\mathcal{S}$ of coset leaders, and the linear code $\mathbb{C}'$ spanned by $\mathbb{C} \cup \left\{(1^\Delta|\lambda)\right\}$ for some $\lambda \in \mathbb{F}_2^n$ and $\Delta \in \mathbb{Z}$. Then the set $\mathcal{S}'$ of coset leaders of $\mathbb{C}'$ can be obtained from $\mathcal{S}$ according to the following bijective correspondence:*

$$\phi : a,\, l \in \mathcal{S} \longmapsto l' \in \mathcal{S}'$$

where $a \in \{\langle 0|\mathbb{F}_2^{\Delta-1}\rangle\}$ and l' is defined by the property

$$l' = \begin{cases} \langle a|l\rangle & \text{if } \mathrm{wt}(\langle a|l\rangle) \leq \mathrm{wt}\left(\langle \bar{a}|\kappa_\lambda(l)\rangle\right), \\ \langle \bar{a}|\kappa_\lambda(l)\rangle & \text{if } \mathrm{wt}(\langle a|l\rangle) > \mathrm{wt}\,\langle \bar{a}|\kappa_\lambda(l)\rangle\,. \end{cases}$$

Figure 2.6 demonstrates the use of the ϕ correspondence to generate the coset leaders of the $(7,3,4)$ binary lexicode $\mathcal{L}_4^3$ from the coset leaders of the $(6,2,4)$ binary lexicode $\mathcal{L}_4^2$ over the base field $\mathbb{F}_2$. Specifically, in this example, $\lambda = f_{\text{lexi}}(\mathcal{L}_4^2) = 001011$ and $\Delta = d - \text{wt}(\lambda) = 4 - 3 = 1$ The coset leaders of $\mathcal{L}_4^2$ are

$$l \in S = \{0, 1, 10, 100, 1000, 10000, 100000, 10010\}$$

and the resulting coset leaders of $\mathcal{L}_4^3$ are

$$l' \in S' = \{0, 1, 10, 100, 1000, 10000, 100000, 1000000\}$$

$a \mid l$	$\bar{a} \mid \kappa_\lambda(l)$	corresponding $l' \in \mathcal{S}'$
0 000000	1 010010	0 000000
0 000001	1 100000	0 000001
0 000010	1 010000	0 000010
0 000100	1 001000	0 000100
0 001000	1 000100	0 001000
0 010000	1 000010	0 010000
0 100000	1 000001	0 100000
0 001001	1 000000	1 000000

Figure 2.6: A list of coset leaders of the $(7,3,4)$ lexicode. It is derived from the coset leaders of the $(6,2,4)$ lexicode using Theorem 2.5 with $\lambda = 010010$ and $\Delta = 1$. Spaces between bits are used to highlight concatenations.

Proof of Theorem 2.5. We introduce the following notation to simply the proof:

$$\text{g}(a,l) \stackrel{\text{def}}{=} \langle a \mid l \rangle \text{ and } \text{h}(a,l) \stackrel{\text{def}}{=} \langle \bar{a} \mid \kappa_\lambda(l) \rangle\,. \tag{2.10}$$

This proof then rests upon two observations. The first observation is that $\mathrm{g}(a,l)$ and $\mathrm{h}(a,l)$ are always in the same coset of the new code $\mathbb{C}'$. This is clear because:

$$\begin{aligned} \mathrm{g}(a,l) \ + \ \mathrm{h}(a,l) &= \langle [a+\bar{a}] \ \mid \ [l+\kappa(l)] \rangle \\ &= \langle 1^{\Delta} \ \mid \ [l+(l+\lambda+c)] \rangle, \text{ for } c \in \mathbb{C} \\ &= \left\langle 1^{\Delta} \mid \lambda \right\rangle + \left\langle 0^{\Delta} \mid c \right\rangle \\ &\in \mathbb{C}' \end{aligned}$$

Here Δ is the padding value, as presented in the statement of the theorem.

The second observation of this proof is that $\mathrm{g}(a,l)$ and $\mathrm{g}(a',l')$ are generally not in the same coset of $\mathbb{C}'$. More specifically, these two vectors are in the same coset precisely when l and l' are in distinct cosets of $\mathbb{C}$ and $\bar{a} \neq a'$. To see this we analyze the two possibilities for the sum $\mathrm{g}(a,l) + \mathrm{g}(a',l')$. In the first case, its leading bits are neither 0^{Δ} nor 1^{Δ}. Alternatively, its leading bits are 0^{Δ} but its trailing bits are the sum of distinct coset leaders $l + l' \notin \mathbb{C}$. Both possibilities preclude $\mathrm{g}(a,l) + \mathrm{g}(a',l')$ from being a codeword of $\mathbb{C}'$, proving the observation.

Based on the above two observations, $\mathrm{g}(\cdot,\cdot)$ and $\mathrm{h}(\cdot,\cdot)$ represent the same two-to-one correspondence between (vector, coset-leader) pairs (a,l) and the cosets of $\mathbb{C}'$. In fact, for each $a \in \mathbb{F}_2^n$ and coset leader l of $\mathbb{C}$, the *coset* of $\mathrm{g}(a,l)$ contains only the vectors:

$$\begin{cases} \langle a \mid (l+c) \rangle \\ \langle \bar{a} \mid (l+\lambda+c) \rangle \end{cases}$$

for any $c \in \mathbb{C}$.

In addition, for any $c \in \mathbb{C}$, $\mathrm{g}(a,l)$ cannot have weight greater than the weight of $\langle a | (l+c) \rangle$, because l is a coset leader, and $\mathrm{h}(a,l)$ cannot have weight greater than $\langle \bar{a} | (l+\lambda+c) \rangle$. Thus, all the vectors in the coset containing $\mathrm{g}(a,l)$ will have weight $\geq \min\{\mathrm{wt}(\mathrm{g}(a,l)), \mathrm{wt}(\mathrm{h}(a,l))\}$, so that either $\mathrm{g}(a,l)$ or $\mathrm{h}(a,l)$ must be a coset leader in $\mathbb{C}'$. Since $\mathrm{g}(a,l)$ and $\mathrm{h}(a,l)$ are correspondences, $\mathcal{S}'$ is the complete set of coset leaders of $\mathcal{C}'$. ∎

2.3.1 Bounds on code parameters

Theorem 2.5 yields a description of the coset leaders of a code $\mathfrak{G}_k$ in terms of the coset leaders of its predecessor $\mathfrak{G}_{k-1}$ in the $\mathfrak{G}$-construction. The maximum weight coset leader of $\mathfrak{G}_k$, in turn, determines the covering radius of this $\mathfrak{G}$-code and, hence, the code parameters of the subsequent codes in the construction. This insight allows us to improve the known bounds on the parameters of lexicodes.

Bounding the Covering Radius

Our first bound is a recursive *upper* bound on the covering radius ρ_m of the m-th code of an arbitrary $\mathfrak{G}$-family. We shall use the term Δ_m to refer to the corresponding term in Definition 1 for this m-th code.

Lemma 2.6 *For any $\mathfrak{G}$-family of codes and $m > 1$,*

$$\rho_m \;\leq\; \left\lfloor \frac{d}{2} \right\rfloor + \rho_{m-1}$$

Proof. Consider constructing $\mathfrak{G}_m$ from $\mathfrak{G}_{m-1}$ using Theorem 2.5. Any two coset leaders l and $\kappa(l)$ of $\mathfrak{G}_{m-1}$ must each have weight at most ρ_{m-1}, by the definition of the covering radius. Consider the weight of a corresponding coset leader of $\mathfrak{G}_m$, based on Theorem 2.5:

$$\begin{aligned} \min\{\mathrm{wt}(a \mid l), \mathrm{wt}\,(\bar{a} \mid \kappa_v(l))\} \;&\leq\; \min\{\mathrm{wt}(a) + \rho_{m-1}, \mathrm{wt}(\bar{a}) + \rho_{m-1}\} \\ &\leq\; \lfloor \Delta_m/2 \rfloor + \rho_{m-1} \end{aligned}$$

for all $a \in \mathbb{F}_2^{\Delta_m}$. Since the $\mathfrak{G}$-construction implicitly demands that $\Delta_m \leq d$, we may conclude that any coset leader of $\mathfrak{G}_m$ must have weight no greater than

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

which bounds the covering radius of $\mathfrak{G}_m$ and proves the lemma. ∎

With some combinatorial analysis, we can also establish a recursive *lower* bound on the covering radius of $\mathfrak{G}$-codes.

Theorem 2.7 *For any $\mathfrak{G}$-family of codes whose seed code has length*

$$n_0 > 3\left\lfloor \frac{d-1}{2} \right\rfloor,$$

it necessarily follows that

$$\rho_m \;\geq\; \left\lfloor \frac{d-1}{2} \right\rfloor + \left\lfloor \frac{d-\rho_{m-1}}{2} \right\rfloor.$$

Part of the proof of this theorem rests on a simple combinatorial lemma, which we provide without proof.

Lemma 2.8 *If $a > 3b > 0$ then*

$$\binom{a}{b} > \sum_{i=0}^{b-1} \binom{a}{i} \tag{2.11}$$

Proof It is well known (s.f.r. [15, p. 122]) for binomial distributions that, for $0 < b < ap$:

$$\sum_{i=0}^{b-1} \binom{a}{i} p^i (1-p)^{a-i} < \frac{b(1-p)}{ap-b} \binom{a}{b} p^b (1-p)^{a-b} \tag{2.12}$$

Setting $p = 1/2$ in equation (2.12) gives us the following relation:

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

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

We proceed to the proof of Theorem 2.7.

Proof of Theorem 2.7. Following convention, we let $t = \lfloor (d-1)/2 \rfloor$ denote the number of errors that the codes of a $\mathfrak{G}$-family can correct. It is well-known that each vector in $\mathbb{F}_2^{n_{m-1}}$ of weight $\leq t$ must be a unique coset leader for $\mathfrak{G}_{m-1}$. Moreover, using our assumption about n_0

(which also holds for n_m, since $n_m \geq n_0$), Lemma 2.8 implies that

$$\binom{n_{m-1}}{t} > \sum_{i=0}^{t-1} \binom{n_{m-1}}{i} \tag{2.13}$$

The left-hand side of (2.13) is the number of vectors of weight t, while the right-hand side is the number of vectors of weight $< t$. Thus, by the pigeon-hole principle, there must be at least one coset leader $l \in \mathfrak{G}_{m-1}$ of weight t whose companion $\kappa(l)$ has weight at least t. Since $\Delta_m \geq d - \rho_{m-1}$ for any $\mathfrak{G}$-code, we may choose $a = 0^{\lfloor \Delta_m/2 \rfloor} 1^{\lceil \Delta_m/2 \rceil}$ to get:

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

∎

The result in Theorem 2.7 also applies to all $\mathfrak{G}$-families trivially seeded with the code $\mathbb{S}_d$.

Lemma 2.9 *For any trivially seeded $\mathfrak{G}$-family of codes,*

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

If a $\mathfrak{G}$-family is trivially seeded, then, necessarily, $n_0 = d$ and $\rho_0 = \lfloor \frac{d}{2} \rfloor$.Thus,

$$n_1 = n_0 + (d - \rho_0) = d + \left\lceil \frac{d}{2} \right\rceil \geq 3 \left\lfloor \frac{d-1}{2} \right\rfloor$$

so that Theorem 2.7 applies to the remaining codes in the family.

We now turn our attention to generating mappings which produce codes whose information rate is locally maximized. More specifically, for any code $\mathbb{C}$ with covering radius $\rho_{\mathbb{C}}$, we will consider only generating mappings f with the property that the Hamming distance from $\mathbb{C}$ to $f(\mathbb{C})$ is exactly $\rho_{\mathbb{C}}$. We will call such mappings *minimal generating mappings*, and the corresponding family of codes *minimal $\mathfrak{G}$-codes*, because they locally minimize length (and hence locally maximize information rate) for a given dimension and minimum-distance.

As an example, the generating mappings for the traditional lexicodes and the trellis-oriented lexicodes are both minimal. We may now easily strengthen Lemma 2.6 by observing in its proof that $\Delta_m = d - \rho_{m-1}$ for *minimal* $\mathfrak{G}$*-codes.*

Corollary 2.10 *For any minimal* $\mathfrak{G}$*-family of codes,*

$$\rho_m \;\leq\; \left\lfloor \frac{d + \rho_{m-1}}{2} \right\rfloor$$

The covering radius bounds we have developed for $\mathfrak{G}$-codes translates naturally to length bounds as well.

Bounding Length

By summing Theorem 2.7 over many iterations, we may obtain a lower bound on the length n_m of the m-th code of any minimal $\mathfrak{G}$-family.

Corollary 2.11 *For any trivially seeded, minimal* $\mathfrak{G}$*-code,*

$$n_m \;\leq \left(m + \frac{1}{2}\right)\frac{d+4}{3} - \frac{2}{3}$$

Proof. Consider summing a weaker inequality obtained from Theorem 2.7 by eliminating the floor function. Summing over iterations $1 \ldots m$ of the $\mathfrak{G}$-construction we get:

$$\begin{aligned}
\sum_{i=1}^{m} \rho_i \;&\geq\; \sum_{i=1}^{m}\left(d - \frac{\rho_{i-1}}{2} - 2\right) \\
&\geq\; m(d-2) - \frac{1}{2}\sum_{i=0}^{m-1}\rho_i
\end{aligned}$$

Noting that for trivially seeded codes, $\rho_0 = \left\lfloor \frac{d}{2} \right\rfloor$ and $\rho_m \geq 0$, we may reduce this to:

$$\frac{3}{2}\sum_{i=1}^{m}\rho_i \;\geq\; m(d-2) - \frac{1}{2}(\rho_0 - \rho_m) \;\geq\; m(d-2) - \frac{d}{4}$$

Furthermore, since we are dealing with minimal $\mathfrak{G}$-codes,

$$n_m \;=\; \sum_{i=0}^{m}(d-\rho_i) \;=\; md - \sum_{i=1}^{m}\rho_i$$

so that we may now conclude the proof:

$$n_m \;=\; md - \sum_{i=1}^{m}\rho_i \;\leq\; md - \frac{2}{3}\left(m(d-2) - \frac{d}{4}\right) \;\leq\; \frac{m}{3}(d+4) + \frac{d}{6}$$

∎

In the case of a $\mathfrak{G}$-family seeded by a non-trivial code of length n_0 and covering radius ρ_0, Corollary 2.11 may be easily generalized to:

$$n_m \;\leq\; n_0 + \frac{m(d+4)+\rho_0}{3} + \frac{d}{6}$$

Clearly, Corollary 2.11 applies to all lexicodes and trellis-oriented lexicodes. It is a asymptotically tighter than the similar bound given by Brualdi and Pless [9, Theorem 3.5]:

$$k \geq \begin{cases} n-2-\lfloor \log_2(n-1) \rfloor & \text{if } d=4, \\ \left\lfloor \frac{4n-d-12}{2d-4} \right\rfloor & \text{if } d \equiv 0 \pmod 4, d \neq 4, 8, \\ \left\lfloor \frac{n}{3} \right\rfloor, & \text{if } d=8,\ n>18, \\ \left\lfloor \frac{4n-d-14}{2d-4} \right\rfloor & \text{if } d \equiv 2 \pmod 4. \end{cases} \tag{2.14}$$

Specifically, Corollary 2.11 asymptotically binds $k \geq 3\frac{n}{d}$ whereas Equation (2.14) binds $k \geq 2\frac{n}{d}$. Nevertheless, both of these bounds are weak as illustrated by Appendix A.1.

2.4 Computations

Theorem 2.5 can be directly translated into an algorithm that computes $\mathfrak{G}$-families. The algorithm simply computes the coset leaders of $\mathfrak{G}$-codes in the family, from which the remaining parameters may be easily deduced. This computation scheme has the advantage of being dependent on the

Algorithm 2.12 *Given a dimension k, a generating mapping $f(\cdot)$, and a seed code $\mathbb{C}$, the following algorithm will compute the code $\mathfrak{G}_k(f, \mathbb{C})$:*

Record the cosets of the seed code $\mathfrak{G}_0$
for i from 1 to k **do**
    compute $v = f(\mathfrak{G}_{i-1})$
    for each coset leader λ of $\mathfrak{G}_{i-1}$ **do**
        compute $\kappa_v(\lambda)$, which is the companion of λ
        for each $a \in \langle 0|\mathbb{F}_2^{\Delta_i - 1}\rangle$ **do**
            if $\mathrm{wt}(\langle a|\lambda\rangle) < \mathrm{wt}(\langle 1^{\Delta_i} + a|\kappa(\lambda)\rangle)$ **then**
                $[a|\lambda]$ is a coset leader of $\mathfrak{G}_i$
            else
                $\left\langle (1^{\Delta_i} + a)|\kappa_v(\lambda)\right\rangle$ is a coset leader of $\mathfrak{G}_i$
    Record the covering radius ρ_i of $\mathfrak{G}_i$ from the newly computed coset leaders

co-dimension (i.e. the dimension of the dual code) rather than the dimension or the length alone. Thus, this algorithm is efficient for high-rate codes, and it is presented in Algorithm 2.12.

In Algorithm 2.12 we reuse our earlier notation that $\Delta_i = d - \mathrm{wt}(f(\mathfrak{G}_{i-1}))$. Note that we assume that the coset leaders of the seed code are known or trivially derivable, as is often the case. The correctness of the algorithm follows immediately from Theorem 2.5. Moreover, we can also compute the running time and space of this algorithm in terms of $\ulcorner m \urcorner \stackrel{\text{def}}{=} n_m - m$, the co-dimension of the m-th $\mathfrak{G}$-code,

Lemma 2.13 *Using an oracle for computing the generating mapping, Algorithm 2.12 runs in space $O(2^{\ulcorner m \urcorner})$ and time $O(n_m \ulcorner m \urcorner 2^{\ulcorner m \urcorner})$.*

An oracle is simply a black box that computes a given function in constant time. In this case, we assume the existence of an oracle for computing the generating mapping because the complexity of such a computation can vary greatly among mappings. The proof of Lemma 2.13 follows from a straightforward analysis of the pseudo-code for Algorithm 2.12.

Proof. At each iteration, Algorithm 2.12 stores the coset leaders of the code $\mathfrak{G}_i$ which it is computing and the companion of each leader. Thus, the algorithm requires space

$$O(\#\mathtt{COSET}_m) = O(2^{n_m - m}) = O(2^{\ulcorner m \urcorner})$$

where $\ulcorner m \urcorner = n_m - m$ denotes the co-dimension and $\#\mathtt{COSET}_m$ denotes the number of cosets of $\mathfrak{G}_m$.

The time bound for this algorithm, on the other hand, is somewhat more complicated. Consider the innermost loop of the algorithm during the computation of the i-th extension. Clearly, based on Theorem 2.5, lines 6-10 are executed at most once for each coset of $\mathfrak{G}_i$. Moreover, the companion calculation in line 5 is executed once for each coset of $\mathfrak{G}_{i-1}$. The companion may be determined with a binary search over coset syndromes; the binary search requires time:

$$O\left(log(\#\texttt{COSET}_{i-1})\right) = \ulcorner i-1 \urcorner$$

Finally, each syndrome calculation used to determine the companion of a coset of $\mathfrak{G}_{i-1}$ itself requires time $O(n_i \ulcorner i \urcorner)$, corresponding to matrix-vector multiplications. Recall that our oracle supplies the computation in line 2 of this analysis in constant time.

Putting everything together, we get a running time of

$$O(2^{\ulcorner i \urcorner} \times [\ulcorner i-1 \urcorner + n_i(\ulcorner i \urcorner)]) \quad \subset \quad O\left(2^{\ulcorner i \urcorner} \ulcorner i \urcorner n_i\right)$$

Except for trivial cases, $\ulcorner i \urcorner$ and n_i do not decrease with i so that:

$$\begin{aligned} \sum_{i=1}^{m} O(2^{\ulcorner i \urcorner} \ulcorner i \urcorner n_i) &\subseteq O(n_m \ulcorner m \urcorner \sum_{i=1}^{m} 2^{\ulcorner i \urcorner}) \\ &\subseteq O(n_m \ulcorner m \urcorner 2^{\ulcorner m \urcorner}) \end{aligned}$$

This proves the lemma. ∎

For the specific case of distance 4 lexicodes, this algorithm is particularly fast, since Brualdi and Pless [9, Thm.3.5] show that, under these circumstances:

$$m = n_m - 2 - \lfloor \log_2(n_m - 1) \rfloor \tag{2.15}$$

so that Lemma 2.14 reduces to time $O(n_m^2 \log(n_m))$ and space $O(n_m)$. This is not surprising given that the distance 4 lexicodes are simply shortenings of the extended Hamming codes [14].

The analysis of Algorithm 2.12 assumes an oracle computation of the generating mapping. For the case of lexicodes and trellis-oriented lexicodes, however, Method 2.2 provides an efficient means of computing this mapping in time $O(n_m \times \#\texttt{COSET}_m)$ and constant space. As in the case of lexicodes and trellis-oriented codes, the overhead of computing $f(\mathfrak{G}_{i-1})$ is usually eclipsed by the running time of the remainder of the algorithm.

Corollary 2.14 *Algorithm 2.12 requires time $O(n_m \ulcorner m \urcorner 2^{\ulcorner m \urcorner})$ and space $O(2^{\ulcorner m \urcorner})$ to compute lexicodes and trellis-oriented lexicodes.*

The complexity of Algorithm 2.12 is thus bounded by the co-dimension of the code and by the difficulty of computing the generating mapping. Under practical conditions, we were able to compute lexicodes well beyond length 44 initially reported in [14].

In addition, we were able to construct trellis-oriented codes with code parameters rivaling those of lexicodes, but with much better trellis state complexities. As an example, we generated trellis-oriented $\mathfrak{G}$-code with code parameters similar to lexicodes, but with better state complexity than the corresponding BCH codes heuristically minimized in [34]. These result were predicted by [34] and are demonstrated n Figures 2.7 and 2.8.

`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 2.7: A comparison of the state complexities of the $(32, 16, 8)$ trellis-oriented lexicode (trelli) and the $(32, 16, 8)$ extended BCH code (ext BCH) heuristically minimized in [34]

`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 2.8: A comparison of the state complexities of the $(31, 16, 7)$ trellis-oriented lexicode (`Trelli`) and a $(31, 16, 7)$ extended BCH code (`BCH`) heuristically minimized in [34]

Finally, the trellis state bounded codes that we have computed show improvements, for various code parameters, over the similar codes computed in [67] using different techniques.Figure 2.9 depicts the generator matrix of the $(70, 49, 8)$ trellis-oriented code produced using Algorithm 2.12. Other examples and computations of the various techniques described in this paper are available in Appendix A.

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

Figure 2.9: The $(70, 49, 8)$ trellis-oriented $\mathfrak{G}$-code.

2.5 Applications

The $\mathfrak{G}$-construction lends itself to several applications in code design, some of which we describe in this section.

2.5.1 Code engineering

The $\mathfrak{G}$-families produced under various generating mappings lend themselves to engineering a code for a specific purpose under specific constraints. For example, if given a dimension k and a desired capability of correcting t errors, the mapping f_{lexi} can be used to generate an appropriate, approximately optimal code.

For a heuristic decoding optimization, the mapping f_{trelli} may be used to generate an approximately optimal "trellis-oriented" code. These codes may be constrained further, however, if there is a particular memory limit m on the decoding space. In this case, the mapping f_{state} may be used to get a reasonably short code satisfying dimension, error-correcting capability, and decoding memory constraints. Alternatively, one might wish to find a family of good codes with a bounded decoding time, in which case the mapping f_{decoding} would be appropriate. In each of these cases, the $\mathfrak{G}$-construction, when possible, provides a code that meets the desired constraints and which, empirically, tends to have good parameters.

Appendix A lists various $\mathfrak{G}$-codes, demonstrating their use in this application. For example we see in Appendix A.1 that though the trellis-oriented codes have almost identical code parameters to their lexicode brothers, they tend to have a much better trellis complexity. As another example, we can see in Appendix A.2 that by sacrificing about 12% of the information rate, we can reduce decoding complexity from the 1024 states in the $(38, 21, 8)$ lexicode to a mere 64 states in the $(43, 21, 8)$ state-bounded code. Such a low memory requirement would be ideal for a *VLSI* chip or a smart card.

2.5.2 Code improvement

The $\mathfrak{G}$-construction also enables us to refine the trellis complexity of a given code. Theorem 2.3 assures us that $f_{\text{trelli}}(\cdot)$ is a locally optimal method of extending an arbitrary subcode if we seek to maximize information rate and minimize decoding complexity. Thus, we have a simple, greedy method in which we may attempt to improve a given code $\mathbb{C}$.

Specifically, we may arbitrarily delete a generator of $\mathbb{C}$ and replace it with a trellis-oriented generator. Because of Theorem 2.3 we know that the resulting code will have trellis complexity and code parameters (e.g. length, minimum distance) no worse than the original code.[2]

In fact, this method may be applied to several generators: shorten $\mathbb{C}$ by deleting k generators; replace the generators with k trellis-oriented generators. This allows us to create an amalgam of the original code and a trellis-oriented code. Though the resulting code is not guaranteed to match the decoding performance of our original code, it often performs much better as we see in the following example.

Example 2.15 *Consider the length 31, dimension 16, triple error-correcting BCH code generated by the polynomial*

$$g(x) = 1 + x + x^2 + x^3 + x^5 + x^7 + x^8 + x^9 + x^{10} + x^{11} + x^{15}.$$

It requires $2^{15} = 32,768$ *trellis states and* $262,139$ *steps for Viterbi decoding. Figure 2.10 shows the dramatic improvement of this code as more generator vectors are replaced with trellis-oriented generators.*

We note that both the state and decoding complexities generally decrease as k *increases. Moreover, the computation time also generally decreases because of the decreasing size of the seed code.*

2.5.3 Practical considerations

In practice, computing the $\mathfrak{G}$-construction using Algorithm 2.12 is only feasible for high-rate codes, because the algorithm needs to maintain a list of coset leaders at each iteration. In theory, it might be possible to use approximation techniques to store only "important" coset leaders, but we leave this as an open problem.

Another problem with the $\mathfrak{G}$-construction is that it is theoretically possible for bounding to be so severe as to halt the construction when, for example, $f(\mathbb{C})$ does not return a vector for a particular code $\mathbb{C}$. Though we have not seen this with trellis state bounding (i.e. using the mapping

[2] We do have to take some care that when we delete the generator the resulting code is non-trivial. If the generator of the resulting code has an all-zero column, making the code trivial, we may simply delete the offending column.

k	State complexity	Decoding complexity	Computation time
1	2^{15}	262,139	26:36
2	2^{15}	262,139	22:11
3	2^{13}	94,203	18:30
4	2^{13}	54,659	24:45
5	2^{11}	36,355	20:25
6	2^{12}	43,779	17:17
7	2^{12}	37,251	14:59
8	2^{11}	32,333	14:49
9	2^{12}	39,373	12:50
10	2^{12}	40,159	11:41
11	2^{12}	37,647	10:34
12	2^{11}	26,303	11:22
13	2^{10}	16,611	10:12
14	2^{10}	16,147	9:39
15	2^{10}	13,603	9:41
16	2^{9}	4,907	1:07

Figure 2.10: Improving a $(31, 16, 7)$ BCH code by replacing k of its generators with trellis-oriented generators. The trellis state complexity and Viterbi decoding complexity of the intermediate $(31, 16, 7)$ codes is noted, as is the time (in minutes) needed to compute them on a Sun Ultra-Sparc 1 workstation.

f_{state}), we have seen indications that this can occur with f_{decoding}. Of course, certain constraints will simply not correspond to any existing linear code, so this problem is inherent to code design.

2.6 Future directions

Many questions remain unanswered in this work. First, it is still not clear why lexicodes, or even trellis-oriented $\mathfrak{G}$-codes, have such good code parameters. Second, we suspect that the bounds on parameters of lexicodes can be improved by a more sophisticated count of the worst-case companion pairings. Finally, one should note that the exponential time and space bound (with respect to co-dimension) of Algorithm 2.12 may be improved by various approximation techniques. The actual continuation of the algorithm depends only on the properties of a few coset leaders, and a heuristic approach to choosing these leaders might work well. Speeding up Algorithm 2.12 would allow for the design of lower rate codes with good trellis decoding properties.

Chapter 3

Tanner-Graph Decoding

Though trellises have been historically used for maximum-likelihood decoding, their complexity increases exponentially in the length of a code for asymptotically good codes [36]. For this reason, different graphs have been presented in the recent literature for use in decoding.

3.1 History

Specifically, iterative decoding algorithms on factor graphs [24] have become a subject of much active research in recent years [6, 11, 33, 40, 41, 50, 65, 66].

As an example, the well-known turbo codes and turbo decoding methods [6, 11] constitute a special case of this general approach to the decoding problem. Factor-graph representations for turbo codes were introduced in [65, 66], where it is also shown that turbo decoding is an instance of a general decoding procedure, known as the sum-product algorithm. Another extensively studied [22, 62] special case is trellis decoding of block and convolutional codes. It is shown in [33, 66] that the Viterbi algorithm on a trellis is an instance of the min-sum iterative decoding procedure, when applied to a simple factor graph. The forward-backward algorithm on a trellis, due to Bahl, Cocke, Jelinek, and Raviv [3], is again a special case of the sum-product decoding algorithm. More general iterative algorithms on factor graphs, collectively termed the "generalized distributive law" or GDL, were studied by Aji and McEliece [1, 2]. These algorithms encompass maximum-likelihood decoding, belief propagation in Bayesian networks [23, 47], and fast Fourier transforms as special cases.

It is proved in [1, 55, 66] that the min-sum, the sum-product, the GDL, and other versions of iterative decoding on factor graphs all converge to the optimal solution if the underlying factor graph is cycle-free. If the underlying factor graph has cycles, very little is known regarding the convergence of iterative decoding methods.

This work is concerned with an important special type of factor graphs, known as Tanner[1] graphs. The subject dates back to the work of Gallager [25] on low-density parity-check codes in 1962. Some 20 years later, Tanner [55] extended the approach of Gallager [25, 26] to codes defined by general bipartite graphs, with the two types of vertices representing code symbols and checks (or constraints), respectively. He also introduced the min-sum and the sum-product algorithms, and proved that they converge on cycle-free graphs. More recently, codes defined on sparse (regular) Tanner graphs were studied by Spielman [50, 53], who showed that such codes become asymptotically good if the underlying Tanner graph is a sufficiently strong expander. These codes were studied in a different context by MacKay and Neal [40, 41], who demonstrated by extensive experimentation that iterative decoding on Tanner graphs can approach channel capacity to within about 1 dB. Latest variants [39] of these codes come within about 0.3 db from capacity, and outperform turbo codes.

In general, a Tanner graph for a code $\mathbb{C}$ of length n over an alphabet A is a pair $(\mathcal{G}, \mathcal{L})$, where $\mathcal{G} = (V, E)$ is a bipartite graph and $\mathcal{L} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_r\}$ is a set of codes over A, called *behaviors* or *constraints*. We denote the two vertex classes of $\mathcal{G}$ by $\mathcal{X}$ and $\mathcal{Y}$, so that $V = \mathcal{X} \cup \mathcal{Y}$. The vertices of $\mathcal{X}$ are called *symbol vertices* and $|\mathcal{X}| = n$, while the vertices of $\mathcal{Y}$ are called *check vertices* and $|\mathcal{Y}| = r$. There is a one-to-one correspondence between the constraints $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_r$ in $\mathcal{L}$ and the check vertices $y_1, y_2, \ldots, y_r$ in $\mathcal{Y}$, so that the length of the code $\mathcal{C}_i \in \mathcal{L}$ is equal to the degree of the vertex $y_i \in \mathcal{Y}$, for all $i = 1, 2, \ldots, r$. A *configuration* is an assignment of a value from A to each symbol vertex $x_1, x_2, \ldots, x_n$ in $\mathcal{X}$. Thus a configuration may be thought of as a vector of length n over A. Given a configuration $\chi = (\chi_1, \chi_2, \ldots, \chi_n)$ and a vertex $y \in \mathcal{Y}$ of degree δ, we define the *projection* χ_y of χ on y as a vector of length δ over A obtained from χ by retaining only those

[1]**Note on terminology.** The term *Tanner graph* was first used by Wiberg, Loeliger, and Kötter [66] to refer to the more general graphs introduced in [66]. These were later termed TWL graphs by Forney [33], although *TWLK graphs* would have been more appropriate. By now, the term *factor graphs* is almost universally used in this context, which leaves *Tanner graphs* available to refer to the kind of factor graphs actually studied by Tanner [55]. The emphasis in this chapter (as in all of the literature [17, 40, 41, 50, 55] on the subject) is on a special type of Tanner graphs that come with simple parity-check constraints. These Tanner graphs include the graphs underlying Gallager's low-density parity-check codes [25, 26].

values that correspond to the symbol vertices adjacent to y. Specifically, if $\{x_{i_1}, x_{i_2}, \ldots, x_{i_\delta}\} \subseteq \mathcal{X}$ is the neighborhood of y in $\mathcal{G}$, then $\chi_y = (\chi_{i_1}, \chi_{i_2}, \ldots, \chi_{i_\delta})$. A configuration χ is said to be *valid* if all the constraints are satisfied, namely if $\chi_{y_i} \in \mathcal{C}_i$ for all $i = 1, 2, \ldots, r$. The code $\mathbb{C}$ represented by the Tanner graph $(\mathcal{G}, \mathcal{L})$ is then the set of all valid configurations.

While the foregoing definition of Tanner graphs is quite general, the theory and practice of the subject [17, 40, 39, 41, 50, 55] is focused almost exclusively on the simple special case where all the constraints are single-parity-check codes over $\mathbb{F}_2$. This work is no exception, although we will provide for the representation of linear codes over arbitrary fields by considering the zero-sum codes over $\mathbb{F}_q$ rather than the binary single-parity-check codes. It seems appropriate to call the corresponding Tanner graphs *simple*. Notice that in the case of simple Tanner graphs, the set of constraints $\mathcal{L}$ is implied by definition, so that one can identify a simple Tanner graph with the underlying bipartite graph $\mathcal{G}$. All of the Tanner graphs considered in this correspondence, except in Section 5.3, are simple. Thus, for the sake of brevity, we will henceforth omit the quantifier "simple." Instead, when we consider the general case in Section 5.3, we will use the term *general* Tanner graphs.

We can think of a (simple) Tanner graph for a binary linear code $\mathbb{C}$ of length n as follows. Let H be an $r \times n$ parity-check matrix for $\mathbb{C}$. Then the corresponding Tanner graph for $\mathbb{C}$ is simply the bipartite graph having H as its $\mathcal{X}, \mathcal{Y}$ adjacency matrix. It follows that the number of edges in any Tanner graph for a linear code $\mathbb{C}$ of length n is $O(n^2)$. Thus, if we can represent $\mathbb{C}$ by a Tanner graph *without cycles*, then maximum-likelihood decoding of $\mathbb{C}$ can be achieved in time $O(n^2)$, using the min-sum algorithm for instance.

However, both intuition and experimentation (cf. [40]) suggest that powerful codes cannot be represented by cycle-free Tanner graphs. The notion that cycle-free Tanner graphs can support only weak codes is, by now, widely accepted. Our goal in this correspondence is to make this "folk knowledge" more precise. We provide rigorous answers to the question: Which codes can have cycle-free Tanner graphs?

Our results in this regard are two-fold: we derive a characterization of the structure of such codes and an upper bound on their minimum distance. The upper bound (Theorem 3.6) shows that codes with cycle-free Tanner graphs provide extremely poor trade-off between rate and distance for each fixed length. This indicates that at very high signal-to-noise ratios these codes will perform badly. In general, however, the minimum distance of a code does not necessarily determine its performance at signal-to-noise ratios of practical interest. Indeed, there exist codes — for example, the turbo codes of [6, 11] — that have low minimum distance, and yet perform very well at low signal-to-noise ratios. The development of analytic bounds on the *performance* of cycle-free Tanner graphs under iterative decoding is a challenging problem, which is beyond the scope of this work. Nevertheless, our results on the *structure* of the corresponding codes indicate that they are very likely to be weak: their parity-check matrix is much too sparse to allow for a reasonable performance even at low signal-to-noise ratios.

The rest of this chapter is organized as follows. We start with some definitions and auxiliary observations in the next section. In Section 3, we show that if an (n, k, d) linear code $\mathbb{C}$ can be represented by a cycle-free Tanner graph and has rate $R = k/n \geq 0.5$, then $d \leq 2$. We furthermore prove that if $R < 0.5$, then $\mathbb{C}$ is necessarily obtained from a code of rate ≥ 0.5 and minimum distance ≤ 2 by simply repeating certain symbols in each codeword. Theorem 3.6 of Section 3.4 constitutes our main result: this theorem gives an upper bound on the minimum distance of a general linear code that can be represented by a cycle-free Tanner graph. Furthermore, the bound of Theorem 3.6 is exact. This is proved in Section 3.5 by means of an explicit construction of a family of (n, k, d) linear codes that attain the bound of Theorem 3.6 for all values of n and k. Asymptotically, for $n \to \infty$, the upper bound takes the form:

$$d \lesssim 2\lfloor 1/R \rfloor \tag{3.1}$$

and an immediate consequence of (3.1) is that asymptotically good codes with cycle-free Tanner graphs do not exist. We show in Section 3.5 that the same is true for Tanner graphs *with* cycles, unless the number of cycles increases exponentially with the length of the code. Finally, we also show in Section 3.5 that for every binary code $\mathbb{C}$ that can be represented by a cycle-free Tanner graph, there exists a graph $\mathcal{G}$ such that $\mathbb{C}$ is the dual of the cycle code of $\mathcal{G}$. This establishes an interesting connection between codes with cycle-free Tanner graphs and the well-known [10, 29, 28, 48, 52] class of graph-theoretic *cut-set* codes. Finally, we conclude this chapter with a partial analysis of general Tanner graphs.

3.2 Preliminaries

Let $H = [h_{ij}]$ be an $r \times n$ matrix, with entries drawn from the finite field $\mathbb{F}_q$ of order q. We let $*$ denote a nonzero entry in H. Given H, we define a bipartite graph T as follows: the vertex set of T consists of the set $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ of *symbol* vertices and the set $\mathcal{Y} = \{y_1, y_2, \ldots, y_r\}$ of *check* vertices; there is an edge (y_i, x_j) in T if and only if $h_{ij} = *$. Thus the neighborhood of the vertex $y_i \in \mathcal{Y}$ corresponds to the i-th row of H, and the neighborhood of the vertex $x_j \in \mathcal{X}$ corresponds to the j-th column of H. We say that T is the *Tanner graph* of H, and denote $T = T(H)$. It is obvious that every matrix defines a unique Tanner graph. Over $\mathbb{F}_2$, the converse is also true: every bipartite graph T defines a unique binary matrix H such that $T = T(H)$, which is the $\mathcal{X}, \mathcal{Y}$ adjacency matrix of T.

We say that a bipartite graph T *represents* the linear code $\mathbb{C}$, or simply that T is a Tanner graph for $\mathbb{C}$, if there exists a parity-check matrix H for $\mathbb{C}$ such that T is the Tanner graph of H. In general, a given linear code can be represented by many distinct Tanner graphs. On the other hand, over $\mathbb{F}_2$, a given Tanner graph represents a unique binary code.

We say that a matrix H is *cycle-free* if the corresponding Tanner graph $T(H)$ is cycle-free. Notice that every submatrix of a cycle-free matrix is also cycle-free. We say that a linear code $\mathbb{C}$ over $\mathbb{F}_q$ is *cycle-free* if there *exists* a cycle-free parity-check matrix for $\mathbb{C}$. Observe that if the matrices H and H' differ by a permutation of rows and columns then the Tanner graphs $T(H)$ and $T(H')$ are isomorphic. On the other hand, if H and H' differ by a sequence of elementary row operations then $T(H)$ and $T(H')$ are generally not isomorphic. Thus it is possible to have two parity-check matrices for the same code, one of which is cycle-free while the other is not. It is also possible to have two cycle-free Tanner graphs for the same code that are not isomorphic.

Example 3.1 *Suppose that a parity-check matrix H for the $(5,2,3)$ binary linear code $\mathbb{C}$ is given by*

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

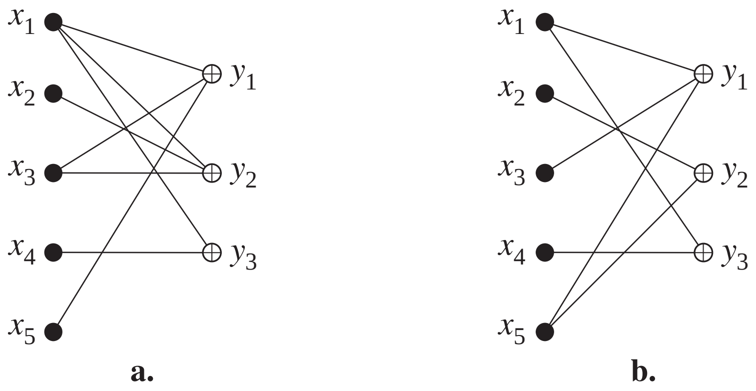


Figure 3.1: Tanner graph with cycles and cycle-free Tanner graph for the same code.

The corresponding Tanner graph $T(H)$ is shown in Figure 3.1a. Notice that H is not cycle-free, since the sequence of edges $(x_1, y_1), (y_1, x_3), (x_3, y_2), (y_2, x_1)$ constitutes a cycle.

However, the code $\mathbb{C}$ is, in fact, cycle-free since adding the first row of H to the second row produces a cycle-free parity-check matrix H' for $\mathbb{C}$. The graph $T(H')$ shown in Figure 3.1b is a cycle-free Tanner graph for $\mathbb{C}$. ∎

A cycle-free graph consisting of a single connected component is called a tree, and a multiple-component cycle-free graph will is a forest. The following simple lemma will serve as our starting point. This lemma is well-known in graph theory – see, for instance, West [64, p. 52] – and is based on the fact that $|E| = |V| - 1$ for trees.

Lemma 3.2 *A graph $\mathcal{G} = (V, E)$ is cycle-free if and only if $|E| = |V| - \omega(\mathcal{G})$, where $\omega(\mathcal{G})$ denotes the number of connected components in $\mathcal{G}$.*

Since every forest contains at least one tree, we have

$$|E| \;\leq\; |V| - 1 \tag{3.2}$$

for any cycle-free graph. If M is an $m \times n$ matrix, then the number of vertices in $T(M)$ is $m + n$ and the number of edges in $T(M)$ is equal to $\mathrm{wt}(M)$ — the total number of nonzero entries in M.

Thus if M is cycle-free, then

$$\mathrm{wt}(M) \le m + n - 1 \tag{3.3}$$

in view of (3.2).

3.3 The structure of cycle-free codes

We start with a simple theorem, which gives a tight upper bound on the minimum distance of high-rate cycle-free linear codes.

Theorem 3.3 *Let $\mathbb{C}$ be an (n, k, d) cycle-free linear code of rate $k/n \ge 0.5$. Then $d \le 2$.*

Proof. Let H be the $r \times n$ cycle-free parity-check matrix for $\mathbb{C}$. We assume without loss of generality that H has full row-rank and that $r = n - k$, since otherwise we can remove the linearly dependent rows of H while preserving the cycle-free property. Let η_i denote the number of columns of weight i in H. If $\eta_0 \ne 0$ then $d = 1$, and we are done. Otherwise, we have at least

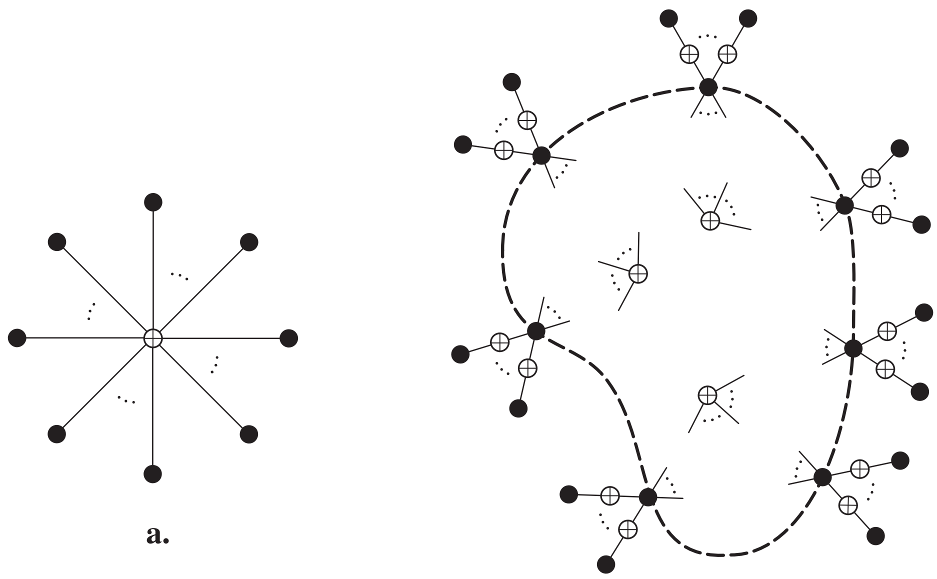
$$\eta_1 + 2(n - \eta_1) \;=\; \eta_1 + 2(\eta_2 + \eta_3 + \cdots + \eta_r) \;\le\; \mathrm{wt}(H) \tag{3.4}$$

ones in H. Moreover,

$$\le\; \mathrm{wt}(H) \;\le\; n + r - 1 \tag{3.5}$$

in view of (3.3). Substituting $r = n - k$ into (3.5), this inequality readily reduces to $\eta_1 \ge k + 1$. Since $k/n \ge 0.5$, it follows that $k \ge r$ and $\eta_1 \ge r + 1$. This means that the number of weight-one columns in H is greater than the number of rows in H. Hence H contains at least two columns of weight one that are scalar multiples of each other, and $d = 2$. ∎

Theorem 3.3 implies that the $(n, n-1, 2)$ code $\mathcal{E}_n$, whose parity-check matrix consists of a single all-1 vector, is, in a sense, the optimal cycle-free code of rate ≥ 0.5. Since all such codes have distance $d \le 2$ and $\mathcal{E}_n$ has the highest rate. The cycle-free Tanner graph for $\mathcal{E}_n$ is depicted in Figure 3.2a.



a. **b.**

Figure 3.2: Tanner graphs for $\mathcal{E}_n$ and for a general low-rate cycle-free code.

To show that the bound of Theorem 3.3 is tight for all n and k, with $n/2 \le k \le n-1$, we may start with the single-parity-check code $\mathcal{E}_{k+1}$ and repeat any symbol (or symbols) in $\mathcal{E}_{k+1}$ until a code of length n is obtained. The following lemma shows that this always produces an $(n, k, 2)$ cycle-free code for $k \ge n/2$.

Lemma 3.4 *Let $\mathbb{C}$ be a cycle-free code of length n and dimension k. Fix a positive integer i, with $i \le n$, and let $\mathbb{C}'$ be the code obtained from $\mathbb{C}$ by repeating the i-th symbol in each codeword. Then $\mathbb{C}'$ is a cycle-free code of length $n+1$ and dimension k.*

Proof. The length and dimension of $\mathbb{C}'$ are obvious. To see that $\mathbb{C}'$ is cycle-free, observe that a Tanner graph T' for $\mathbb{C}'$ can be obtained from the cycle-free Tanner graph T for $\mathbb{C}$ by introducing two new vertices x' and y' and two new edges: (x', y') and (x_i, y'). Clearly this procedure does not create new cycles. ∎

Let $\mathbb{C}$ be a cycle-free code of length n, and let $\mathbb{C}^*$ be the code of length $n+\delta$ obtained from $\mathbb{C}$ by iteratively applying δ times the procedure of Lemma 3.4, while possibly choosing a different value of i at different iterations. We then say that $\mathbb{C}^*$ is a code obtained by *repeating symbols* in $\mathbb{C}$. To make our terminology precise, we further extend the notion of codes obtained by "repeating symbols in $\mathbb{C}$" to also include the codes obtained from $\mathbb{C}^*$ by appending all-zero coordinates. The following proposition shows that *every* low-rate cycle-free linear code has this structure.

Proposition 3.5 *Let $\mathbb{C}$ be an (n, k, d) cycle-free linear code over $\mathbb{F}_q$ of rate $k/n \leq 0.5$. Then, up to scaling by constants in $\mathbb{F}_q$ at certain positions, $\mathbb{C}$ is obtained by repeating symbols in a cycle-free code of rate > 0.5.*

Proof. Let H be the $r \times n$ cycle-free parity-check matrix for $\mathbb{C}$, with $r = n - k$. Then by Lemma 3.2 and (3.2), we have $\mathrm{wt}(H) \leq n + r - 1 \leq 3r - 1$, where the second inequality follows from the fact that $k/n \leq 0.5$. This implies that H contains at least one row of weight ≤ 2. If this row is of weight one, then the corresponding coordinate of $\mathbb{C}$, say the n-th coordinate, is all-zero. Otherwise, assume without loss of generality that this row is of the form $h = (0, 0, \ldots, 0, *, *)$. Then, up to scaling the last two columns of H by constants in $\mathbb{F}_q$, we may further assume that $h = (0, 0, \ldots, 0, 1, -1)$. This would mean that the n-th symbol in $\mathbb{C}$ is a repetition of the preceding symbol. In both cases, we can puncture-out the n-th coordinate of $\mathbb{C}$, and recursively repeat the argument until a cycle-free code of rate > 0.5 is obtained. ∎

Loosely speaking, Proposition 4 implies that every cycle-free code $\mathbb{C}$ of rate ≤ 0.5 can be represented by a Tanner graph whose structure is shown in Figure 3.2b. The dashed line in Figure 3.2b encloses a cycle-free Tanner graph for a code $\mathbb{C}'$ of rate > 0.5 and distance ≤ 2. It follows that to establish a bound on the minimum distance of low-rate cycle-free codes, we need to determine an optimal choice for $\mathbb{C}'$ in Figure 3.2b and an optimal sequence of symbol repetitions. This problem is considered in detail in the next two sections.

Specifically, we will show in Section 3.4 and Section 3.5 that the single-parity-check code constitutes an optimal choice for $\mathbb{C}'$, and every symbol should be repeated equally often.

3.4 The minimum distance of cycle-free codes

The following theorem gives an upper bound on the minimum distance of cycle-free linear codes. Later in this section, we will show that this bound is tight for all values of n and k.

Theorem 3.6 *Let $\mathbb{C}$ be an (n,k,d) cycle-free linear code over $\mathbb{F}_q$. Then*

$$d \;\le\; \left\lfloor \frac{n}{k+1} \right\rfloor + \left\lfloor \frac{n+1}{k+1} \right\rfloor \tag{3.6}$$

Observe that for $k/n \ge 0.5$, the bound in (3.6) reduces to $d \le 2$. This simple special case was dealt with in Theorem 3.3. The proof of Theorem 3.6 for general n and k is considerably more involved and will be presented in Section 3.4.2, after we establish a series of auxiliary lemmas in the next subsection.

3.4.1 Groundwork: auxiliary lemmas

For the sake of brevity, we will consider only binary codes, although our proof readily extends to codes over an arbitrary finite field. Furthermore, with a slight abuse of notation, we will not distinguish between equivalent codes: namely, given a parity-check matrix H for a code $\mathbb{C}$, we will often freely permute the columns of H while still referring to the resulting matrix as a parity-check matrix for $\mathbb{C}$.

The following lemma will be the basis for the induction in our proof of Theorem 3.6.

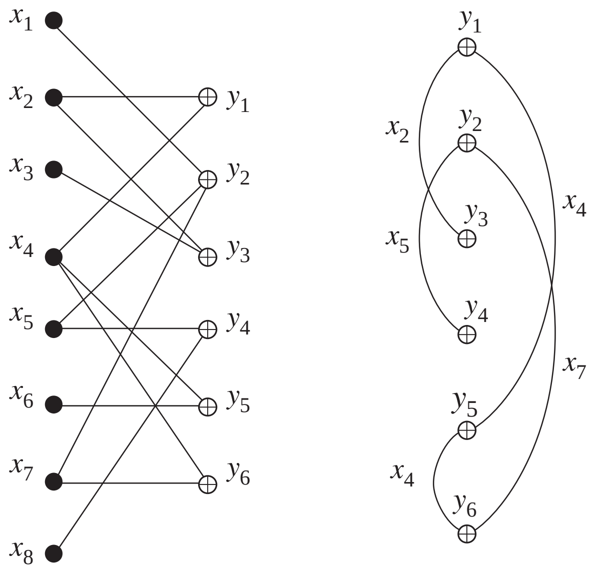
Lemma 3.7 *Let H be a cycle-free binary matrix. Then at least one of the following statements is true:*

- $\bullet$. *The matrix H contains a row of weight two or less;*
- $\diamond$. *The matrix H contains three identical columns of weight one;*
- $\star$. *The matrix H contains two identical columns of weight one, and furthermore the row of H which contains the nonzero entries of these two columns has weight three.*

Proof. Let $T(H)$ be the Tanner graph of H. We now construct another graph $\mathcal{G}$, called the *row-graph* of H, whose vertex set $y_1, y_2, \ldots, y_{r-s}$ corresponds to the rows of H. The edge set of $\mathcal{G}$ is derived from the columns of H of weight ≥ 2, so that a column of weight w in H contributes $w-1$ edges to $\mathcal{G}$. An example illustrating the construction of the row-graph $\mathcal{G}$ for a 6×8 cycle-free matrix is depicted in Figure 3.3.

$$
\begin{array}{c}
\begin{array}{cccccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 & x_8 \end{array} \\
\begin{bmatrix}
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0
\end{bmatrix}
\end{array}
\begin{array}{c} \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{array}
$$

a.



b. **c.**

Figure 3.3: A cycle-free matrix, its Tanner graph, and its row-graph $\mathcal{G}$.

Specifically, there is an edge between y_i and y_j in $\mathcal{G}$ iff $i < j$ and there exists a column $(a_1, a_2, \ldots, a_{r-s})^t$ in H, such that $a_i = a_j = 1$ while $a_{i+1} = a_{i+2} = \cdots = a_{j-1} = 0$. Notice that each such edge (y_i, y_j) in $\mathcal{G}$ corresponds to a path of length two in $T(H)$: namely $\langle (y_i, x_p), (x_p, y_j) \rangle$, where p denotes the position at which the column $(a_1, a_2, \ldots, a_{r-s})^t$ is to be found in H. It follows that if there is a cycle in $\mathcal{G}$, then there is a cycle in $T(H)$. Since $T(H)$ is cycle-free, then so is $\mathcal{G}$. As such, $\mathcal{G}$ necessarily contains at least two vertices of degree ≤ 1, in view of (3.2). Let y^* be one such vertex in $\mathcal{G}$, and let $a^* = (a_1, a_2, \ldots, a_{n-s})$ be the corresponding row of H. If $\mathrm{wt}(a^*) \leq 2$, then ($\bullet$) is true. If $\mathrm{wt}(a^*) = 3$ and $\deg y^* = 0$, then ($\diamond$) is true. If $\mathrm{wt}(a^*) = 3$ and $\deg y^* = 1$, then ($\star$) is true. Finally, if $\mathrm{wt}(a^*) \geq 4$, then ($\diamond$) is true, regardless of whether $\deg y^* = 0$ or $\deg y^* = 1$. ▮

Let $\mathbb{C}$ be an (n, k, d) cycle-free binary linear code, and let H be an $r \times n$ cycle-free parity-check matrix for $\mathbb{C}$, where $r = n - k$. We say that H is in *s-canonical form* if this matrix has the following structure:

$$
H = \left[\begin{array}{c|c} A & \mathbf{0} \\ \hline B & I_s \end{array} \right] \tag{3.7}
$$

where all the rows of B have weight ≤ 1, and I_s is the $s \times s$ identity matrix, for some s in the range $0 \leq s \leq r$. Notice that if $s = 0$ then (3.7) reduces to $H = A$ (which means that every matrix is in 0-canonical form), while if $s = r$ then the corresponding canonical form is $H = [B \,|\, I_r]$. We will use the shorthand $H = A \|_s B$ to denote the s-canonical form in (3.7).

Lemma 3.7 trivially generalizes to matrices in s-canonical form.

Lemma 3.8 *Let $H = A\|_s B$ be a cycle-free binary matrix in s-canonical form, and suppose that $s < r$. Then at least one of the following statements is true:*

$\bullet$. *The matrix A contains a row of weight two or less;*

$\diamond$. *The matrix A contains three identical columns of weight one;*

$\star$. *The matrix A contains two identical columns of weight one, and furthermore the row of H which contains the nonzero entries of these two columns has weight three.*

Proof. Evidently $T(A)$ is a subgraph of $T(H)$, obtained by retaining only the first $n - s$ symbol vertices $x_1, x_2, \ldots, x_{n-s}$, the first $r - s$ check vertices $y_1, y_2, \ldots, y_{r-s}$, and all the edges between these vertices. Since $T(H)$ is cycle-free by assumption, so is $T(A)$. Thus, Lemma 3.7 applies to complete the proof. ∎

We will say that an $r \times n$ matrix H is in *reduced canonical form*, if $H = A\|_s B$ and either $s = r$ or all the rows of A have weight ≥ 3. It is a non-trivial fact that every cycle-free parity-check matrix can be put into such a form.

Lemma 3.9 *Let $\mathbb{C}$ be an (n, k, d) cycle-free binary linear code. Then there exists a cycle-free parity-check matrix for $\mathbb{C}$, which is in reduced canonical form.*

Proof. Let H be an arbitrary cycle-free parity-check matrix for $\mathbb{C}$. We first put H in s-canonical form, for the highest possible s, by means of row and column permutations. This is achieved by considering all the rows of H of weight one, for which the nonzero entry $*$ is contained in a column of weight one, and all the rows of H of weight two such that at least one of the two $*$'s is contained in a column of weight one. Under an appropriate column permutation, these rows of H will form the submatrix $[B \,|\, I_s]$ in (3.7). If there are no such rows, then $s = 0$ and $H = A$. Since row and

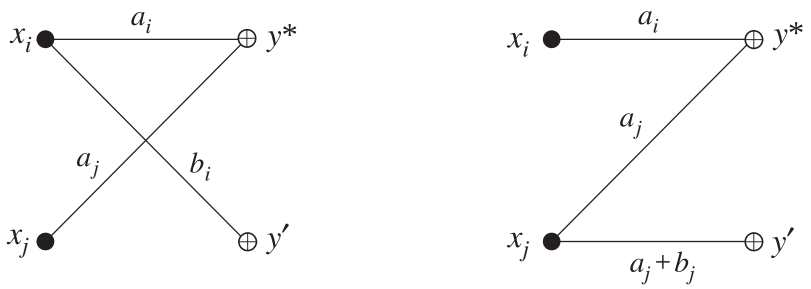


Figure 3.4: Part of the Tanner graph for $\mathbb{C}$ before and after the elementary row operation.

column permutations preserve the cycle-free property of H, this procedure produces a cycle-free parity-check matrix $H' = A\|_s B$ for $\mathbb{C}$, which is in canonical form, although not necessarily reduced.

Since $H' = A\|_s B$ is full-rank by assumption, all the rows of A have weight ≥ 1. The key observation is that certain elementary operations on the rows of H' allow us to eliminate rows of weight one and two in A, while still preserving the cycle-free property.

Indeed, suppose that A has a row $(a_1, a_2, \ldots, a_{r-s})$ of weight one, with a single $*$ in position i. Then we can add this row to all the rows of H' that are nonzero at position i. This procedure is equivalent to deleting all but one of the edges incident at the symbol vertex x_i in $T(H')$, which certainly does not create new cycles. Following this procedure, the single $*$ in $(a_1, a_2, \ldots, a_{r-s})$ is contained in a column of weight one. Hence we can transform the resulting cycle-free parity-check matrix for $\mathbb{C}$ into the form $A'\|_{s+1}B'$, by means of row and column permutations, thereby eliminating the row of weight one in A.

Now suppose that A has a row $(a_1, a_2, \ldots, a_{r-s})$ of weight two, with the two $*$'s in positions i and j, and let y^* be the corresponding check vertex in $T(H')$. We again add this row to all the rows of H' that are nonzero at position i. Let $(b_1, b_2, \ldots, b_n)$ be such a row, and let y' denote the corresponding check vertex of $T(H')$. If $b_j = b_i = *$, then $T(H')$ contains the cycle $\langle (x_i, y^*), (y^*, x_j), (x_j, y'), (y', x_i) \rangle$. Since $T(H')$ is cycle-free, we conclude that $b_i = *$ while $b_j = 0$. Hence, adding row $(a_1, a_2, \ldots, a_n)$ to $(b_1, b_2, \ldots, b_n)$ corresponds to deleting the edge (y', x_i) in $T(H')$ while introducing a new edge (y', x_j), as illustrated in Figure 3.4.

However, this new edge cannot close a cycle, since $T(H')$ is cycle-free and a path from y' to x_j already exists in $T(H')$: indeed $\langle (y', x_i), (x_i, y^*), (y^*, x_j) \rangle$ is such a path. Following all these elementary row operations, the $*$ at position i in $(a_1, a_2, \ldots, a_n)$ is contained in a column of weight one. Thus we can again transform the resulting cycle-free parity-check matrix for $\mathbb{C}$ into the form $A' \|_{s+1} B'$, thereby eliminating the row of weight two in A.

We iteratively repeat the process described in the foregoing two paragraphs, until either $s = r$ or all the rows of A have weight ≥ 3. This procedure produces a cycle-free parity-check matrix for $\mathbb{C}$ that is in reduced canonical form. ∎

If the reduced canonical form in Lemma 3.9 is achieved in the extreme case $s = r$, then it is easy to prove the claim of Theorem 3.6.

Lemma 3.10 *Let $\mathbb{C}$ be an (n, k, d) cycle-free binary linear code. If there is a parity-check matrix for $\mathbb{C}$ of the form $H = [B \,|\, I_r]$, where $r = n - k$ and the rows of B have weight ≤ 1, then the minimum distance of $\mathbb{C}$ satisfies the upper bound of Theorem 3.6.*

Proof. If B contains a column of weight w, then clearly $d \leq w + 1$. Since B is an $r \times k$ matrix, there will always be at least one column with maximum weight $\frac{wt(B)}{k}$, so that

$$d \;\leq\; \frac{\mathrm{wt}(B)}{k} + 1.$$

Moreover, all the rows of B have weight ≤ 1, so that

$$d \;\leq\; \frac{\mathrm{wt}(B)}{k} + 1 \;\leq\; \frac{r}{k} + 1 \;=\; \frac{n}{k} \tag{3.8}$$

As d is an integer, this implies that $d \leq \lfloor n/k \rfloor$. It is easy to see that $\lfloor n/k \rfloor \leq 2\lfloor n/(k+1) \rfloor$, unless $k = 1$ and n is odd. But, in the latter case, both (3.6) and (3.8) reduce to $d \leq n$. ∎

3.4.2 Proof of the main result

We are now in a position to proceed with the proof of Theorem 3.6. Part of this proof involves tedious calculations which are needed to establish the tight bound, and we delegate these to the Appendix B. The proof is by induction on the length n of the code. Thus we first transform (3.6) into the form:

$$
d \le \begin{cases} 2\left\lfloor \dfrac{n}{k+1} \right\rfloor & \text{if} \quad n+1 \not\equiv 0 \mod (k+1) \\ 2\left\lfloor \dfrac{n}{k+1} \right\rfloor + 1 & \text{if} \quad n+1 \equiv 0 \mod (k+1) \end{cases} \tag{3.9}
$$

that is more conducive to induction on n. It can be easily seen with simple algebraic manipulation that equations (3.6) and (3.9) are equivalent.

As the induction basis, we may consider codes of length $n = 2$, for which the bound of Theorem 3.6 holds trivially. As the induction hypothesis, we assume that the minimum distance of every cycle-free linear code of length $n' < n$ satisfies the bound of Theorem 3.6.

The induction step is established as follows. Let $\mathbb{C}$ be an (n, k, d) cycle-free binary linear code. We may assume that $2 \le k \le n-1$, since for $k = 1$ the bound of (3.9) reduces to $d \le n$, while if $k = n$ then $d = 1$ and (3.9) obviously holds with equality.

By Lemma 3.9, there exists an $r \times n$ cycle-free parity-check matrix $H = A\|_s B$ for $\mathbb{C}$, which is in reduced canonical form. If $s = r$ then the induction step follows immediately from Lemma 3.10. Otherwise, Lemma 3.8 implies that either ($\diamond$) or ($\star$) is true. Observe that case ($\bullet$) of Lemma 3.8 does not occur, since by the definition of a reduced canonical form, the matrix A does not have rows of weight ≤ 2. Furthermore, both ($\diamond$) and ($\star$) imply that A contains at least two identical columns of weight one. Let i and j denote the positions at which these two columns are found in A. Further, let w_i and w_j denote the weight of the corresponding columns of B and we denote $w = w_i + w_j + 2$.

It follows from the canonical form structure of $H = A\|_s B$ that the i-th bit, respectively j-th bit, of $\mathbb{C}$ is repeated w_i times, respectively w_j times, in the last $n-s$ positions. Further observe that the sum of the i-th and the j-th columns of H together with the corresponding $w_i + w_j$ columns of the identity matrix produces the all-zero r-tuple. Hence there is a codeword of weight $w = w_i + w_j + 2$ in $\mathbb{C}$ and $d \le w$.

We now shorten $\mathbb{C}$ at positions i and j to obtain an (n', k', d') code $\mathbb{C}'$. That is, we consider the subcode of $\mathbb{C}$ consisting of all the codewords that are zero on positions i and j and define $\mathbb{C}'$ to be the code obtained by puncturing out the $w_i + w_j + 2$ zero positions in this subcode. Notice that

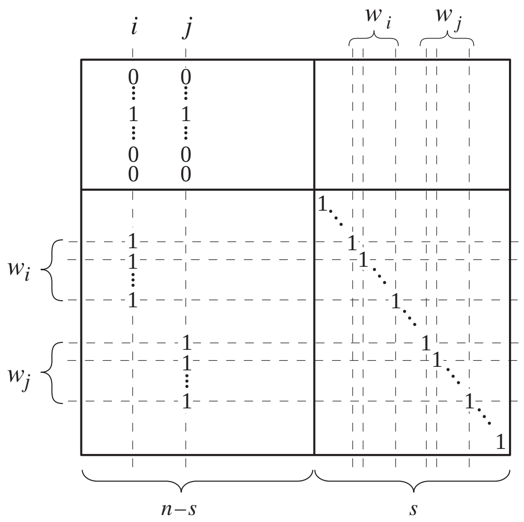


Figure 3.5: Deleting rows and columns of $H = A\|_s B$ to shorten a cycle-free code.

shortening $\mathbb{C}$ at positions i and j is equivalent to deleting $w_i + w_j + 2$ columns of H and $w_i + w_j$ rows of H, as illustrated in Figure 3.5. It is easy to see that the parameters of the resulting code $\mathbb{C}'$ satisfy

$$n' \;=\; n - w \qquad\qquad k' \;\geq\; k-2 \qquad\qquad d' \;\geq\; d \tag{3.10}$$

Furthermore, since H is cycle-free by assumption, the parity-check matrix for $\mathbb{C}'$ which results by deleting rows and columns of H is also cycle-free.

It follows that $\mathbb{C}'$ is a cycle-free code of length $n' < n$, and we can invoke the induction hypothesis. We distinguish between two cases.

Case 1. $n' + 1 \not\equiv 0 \mod (k'+1)$

In this case, the induction hypothesis implies that $d' \leq 2\lfloor n'/(k'+1)\rfloor$. Taking into account the relations (3.10) between the parameters of $\mathbb{C}$ and $\mathbb{C}'$, we obtain

$$d \;\leq\; 2\left\lfloor \frac{n'}{k'+1} \right\rfloor \;\leq\; 2\left\lfloor \frac{n-w}{k-1} \right\rfloor \;\leq\; 2\left\lfloor \frac{n-d}{k-1} \right\rfloor \tag{3.11}$$

where the third inequality follows from the fact that $d \leq w$. It is shown in Appendix B that the relation between n, k, and d in (3.11) implies (3.9).

Case 2. $n' + 1 \equiv 0 \mod (k' + 1)$

We again apply the induction hypothesis. Notice that in this case, the upper bound of Theorem 3.6 may be re-written as

$$d' \ \leq\ 2\left\lfloor\frac{n'}{k'+1}\right\rfloor + 1 \ =\ 2\,\frac{n'+1}{k'+1} - 1 \tag{3.12}$$

where $(n'+1)/(k'+1)$ is a positive integer. Suppose that $d \leq d'$ is an even integer. Then, since the right-hand side of (3.12) is an odd integer, we have

$$d \ \leq\ 2\,\frac{n'+1}{k'+1} - 2 \ \leq\ 2\,\frac{n-d+1}{k-1} - 2 \tag{3.13}$$

where the second inequality in (3.13) follows from (3.10) along with the fact that $d \leq w$. It is shown in Appendix B that (3.13) implies (3.9).

Now suppose that d is odd. In this case, the bound of (3.12) does not suffice to establish (3.9), and we need to use the additional structure present in statements ($\diamond$) and ($\star$) of Lemma 3.8. Suppose that ($\diamond$) is true, and the matrix A contains three identical columns of weight one, at positions α, β, γ. Let $w_\alpha, w_\beta, w_\gamma$ denote the weight of the corresponding columns of B. Notice that at least one of $w_\alpha + w_\beta$, $w_\alpha + w_\gamma$, $w_\beta + w_\gamma$ is an even integer. Hence we can choose the two positions i and j in Figure 3.5 from the three positions α, β, γ, in such a way that $w = w_i + w_j + 2$ is even. Since $d \leq w$ and d is odd, it follows that $d \leq w - 1$. In conjunction with (3.12), we thus obtain

$$d \ \leq\ 2\,\frac{n'+1}{k'+1} - 1 \ \leq\ 2\,\frac{n-w+1}{k-1} - 1 \ \leq\ 2\,\frac{n-d}{k-1} - 1 \tag{3.14}$$

It is shown in Appendix B that if d is odd, then (3.14) implies (3.9). Now suppose that ($\star$) is true, and the matrix H contains a row of weight three, with the three $*$ at positions h, i, j. Then after deleting $w_i + w_j + 2$ columns of H and $w_i + w_j$ rows of H as illustrated in Figure 3.5, we are left with a row of weight one, with the single $*$ at position h. This means that the h-th position in $\mathbb{C}'$ is entirely zero; this position can be punctured-out without decreasing the dimension or the minimum distance. We thus obtain an (n^*, k^*, d^*) code $\mathbb{C}^*$, with $n^* = n' - 1$,

$k^* = k'$, and $d^* = d'$. Applying the induction hypothesis to $\mathbb{C}^*$, we get

$$d \;\le\; d^* \;\le\; 2\left\lfloor \frac{n^*}{k^*+1} \right\rfloor \;\le\; 2\left\lfloor \frac{n-w-1}{k-1} \right\rfloor \;\le\; 2\left\lfloor \frac{n-d}{k-1} \right\rfloor \tag{3.15}$$

where the second inequality follows from the fact that if $n'+1 \equiv 0 \mod (k'+1)$ then $n^*+1 \not\equiv 0 \mod (k^*+1)$. The right-hand side of (3.15) is the same as relation (3.11), which was already considered in Case 1.

It remains to consider the case where $\mathbb{C}' = \{\mathbf{0}\}$, namely $k' = 0$. But in this case $k \le 2$ in view of (3.10), and the upper bound of Theorem 3.6 follows directly from the Griesmer bound [42, p. 547]. Since we have now exhausted all the possibilities, this establishes the induction step and completes the proof of Theorem 3.6.

Optimal cycle-free codes. We now construct a family of cycle-free codes that attain the bound of Theorem 3.6 with equality, for all values of n and k. The construction is quite simple: as in Section 3.3, we start with the single-parity-check code $\mathcal{E}_{k+1}$ of dimension k, and repeat the symbols of $\mathcal{E}_{k+1}$ until a code $\mathbb{C}$ of length n is obtained. It is obvious that the dimension of $\mathbb{C}$ is k, and by Lemma 3.4 this code is cycle-free. The minimum distance of $\mathbb{C}$ will depend on the sequence of symbol repetitions. The idea is to repeat each symbol in $\mathcal{E}_{k+1}$ equally often and as much as possible. For example, for $n = 13$ and $k = 3$, we obtain the following parity-check matrix in reduced canonical form (0 bits have been intentionally replaced with blank spaces to accentuate the structure of the matrix):

$$H = \left[\begin{array}{cccc|ccccccccc}
1 & 1 & 1 & 1 & & & & & & & & & \\
\hline
1 & & & & 1 & & & & & & & & \\
1 & & & & & 1 & & & & & & & \\
1 & & & & & & 1 & & & & & & \\
 & 1 & & & & & & 1 & & & & & \\
 & 1 & & & & & & & 1 & & & & \\
 & & 1 & & & & & & & 1 & & & \\
 & & 1 & & & & & & & & 1 & & \\
 & & & 1 & & & & & & & & 1 & \\
 & & & 1 & & & & & & & & & 1
\end{array}\right] \tag{3.16}$$

which defines a $(13,3,6)$ cycle-free code. In general, the number of symbols to be repeated is $k+1$, while the number of positions available is $n-(k+1)$. Write:

$$n-(k+1) \;=\; a(k+1)+b$$

where a, b are integers, and $0 \le b \le k$. This decomposition of the number of available positions means that in our construction exactly $k-b+1$ symbols of $\mathcal{E}_{k+1}$ will be repeated

$$a \;=\; \left\lfloor \frac{n-(k+1)}{k+1} \right\rfloor \;=\; \left\lfloor \frac{n}{k+1} \right\rfloor - 1$$

times, while the remaining b symbols of $\mathcal{E}_{k+1}$ will be repeated $a+1$ times. If $b \le k-1$, then at least two symbols of $\mathcal{E}_{k+1}$ are repeated exactly a times. Since $\mathcal{E}_{k+1}$ contains a codeword of weight 2 in every two positions, the minimum distance of the resulting code $\mathbb{C}$ is

$$d \;=\; 2+a+a \;=\; 2\left\lfloor \frac{n}{k+1} \right\rfloor \tag{3.17}$$

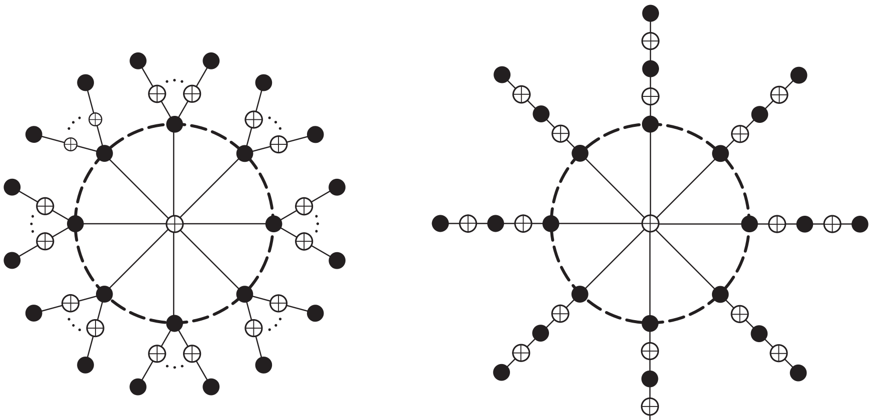
If $b = k$, then only one symbol in $\mathcal{E}_{k+1}$ is repeated a times, while all the other symbols are repeated $a+1$ times. In this case, the minimum distance of $\mathbb{C}$ is

$$d \;=\; 2+a+(a{+}1) \;=\; 2\left\lfloor \frac{n}{k+1} \right\rfloor + 1 \tag{3.18}$$

Notice that $b = k$ if and only if $n+1 \equiv 0 \mod (k+1)$. Hence it follows from (3.17) and (3.18) that the code $\mathbb{C}$ constructed in this manner attains the bound of Theorem 3.6 with equality.

Figure 3.6 schematically shows two alternative cycle-free Tanner graphs for codes resulting from this construction (compare the Tanner graph in Figure 3.6a with Figure 3.2b).

We point out that although cycle-free codes obtained by repeating symbols in $\mathcal{E}_{k+1}$ have the highest possible minimum distance, they are not the only codes with this property. For example,



a. b.

Figure 3.6: Two alternative Tanner graphs for optimal cycle-free codes.

consider the following parity-check matrix in reduced canonical form:

$$H = \left[\begin{array}{ccccc|cccccccc}
1 & 1 & 1 & & & & & & & & & & \\
1 & & & 1 & 1 & & & & & & & & \\
\hline
 & 1 & & & & 1 & & & & & & & \\
 & 1 & & & & & 1 & & & & & & \\
 & & 1 & & & & & 1 & & & & & \\
 & & 1 & & & & & & 1 & & & & \\
 & & & 1 & & & & & & 1 & & & \\
 & & & 1 & & & & & & & 1 & & \\
 & & & & 1 & & & & & & & 1 & \\
 & & & & 1 & & & & & & & & 1
\end{array}\right] \tag{3.19}$$

It is easy to see that this matrix defines a $(13, 3, 6)$ cycle-free code $\mathbb{C}'$, whose distance attains the bound of Theorem 3.6 with equality. This code was obtained by repeating symbols in a $(5, 3, 2)$ code. It can be readily verified that $\mathbb{C}'$ is not equivalent to the $(13, 3, 6)$ cycle-free code $\mathbb{C}$, defined by the parity-check matrix in (3.16) and obtained by repeating symbols in $\mathcal{E}_4$. For instance, $\mathbb{C}$ contains the all-one codeword, while $\mathbb{C}'$ does not.

3.5 Further results

In this section, we discuss three distinct topics: the connection between binary cycle-free codes and cut-set codes of a graph, the asymptotic behavior of Tanner graphs with cycles, and the extension of the results of the previous section to general Tanner graphs.

3.5.1 Cycle-free codes and graph-theoretic codes.

There is an interesting connection between cycle-free codes and cut-set codes of a graph. Let $\mathcal{G} = (V, E)$ be a multi-graph (a graph that may contain multiple edges with both endpoints the same) with $n = |E|$ edges and $m = |V|$ vertices. A *cut-set* in $\mathcal{G}$ is a set of edges which consists of all the edges having one endpoint in some set $X \subset V$ and the other endpoint in $V \setminus X$. Under the operation of symmetric difference, the cut-sets in $\mathcal{G}$ form a subspace of the binary vector space of all subsets of E. Hence replacing subsets of E by their characteristic vectors in $\mathbb{F}_2^n$ produces a binary linear code $\mathbb{C}(\mathcal{G})$, called the *cut-set code* of $\mathcal{G}$. The dual code of $\mathbb{C}(\mathcal{G})$ is the *cycle code* of $\mathcal{G}$, defined as the linear span of the characteristic vectors of cycles in $\mathcal{G}$. Graph theoretic codes, namely cut-set codes and cycle codes of a graph, have been extensively studied — see [10, 29, 28, 51, 52] for instance. The connection between cycle-free codes and cut-set codes of a graph can be summarized as follows.

Theorem 3.11 *Let $\mathbb{C}$ be a cycle-free binary linear code of length n. Then there exists a graph $\mathcal{G}$ with n edges, such that $\mathbb{C}$ is a cut-set code of $\mathcal{G}$.*

Proof. Let H be an $r \times n$ cycle-free parity-check matrix for $\mathbb{C}$, and let $T = T(H)$ be the corresponding cycle-free Tanner graph that represents $\mathbb{C}$. The following procedure converts T into a graph $\mathcal{G}$, such that $\mathbb{C}$ is the cut-set code of $\mathcal{G}$. We will describe this procedure assuming that T is a tree, in which case $\mathcal{G}$ is connected. In case T is a forest consisting of ω trees, the same procedure should be carried out independently for each tree in T, and $\mathcal{G}$ will have ω connected components.

Let $\mathcal{Y} = \{y_1, y_2, \ldots, y_r\}$ be the set of check vertices in T, and let $X_i \subseteq \mathcal{X}$ denote the neighborhood of $y_i \in \mathcal{Y}$ for $i = 1, 2, \ldots, r$. Further define $X_i^* = X_1 \cup X_2 \cup \cdots \cup X_i$. Since T is a tree, it is always possible to enumerate the check vertices in T in such a way that X_i intersects X_{i-1}^* in *one and only one* symbol vertex for all i. Given such enumeration $y_1, y_2, \ldots, y_r$, we construct $\mathcal{G}$ iteratively, check-vertex by check-vertex. First, we represent y_1 and its neighborhood X_1 by a

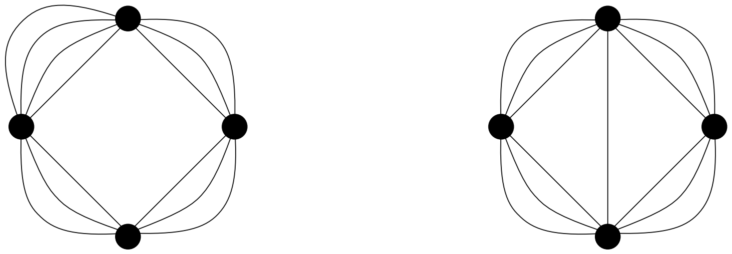


Figure 3.7: Two inequivalent cycle-free cut-set codes.

cycle $\mathcal{G}_1$ consisting of $|X_1|$ edges and $|X_1|$ vertices. Now suppose that $X_2 \cap X_1 = \{x_2\}$. Then we create $\mathcal{G}_2$ from $\mathcal{G}_1$ by appending $|X_2| - 1$ edges — one for each symbol vertex in X_2 except x_2 — and $|X_2| - 2$ vertices, in such a way that the edges corresponding to the symbol vertices in X_2 form a cycle in $\mathcal{G}_2$. And so forth: if $X_i \cap X_{i-1}^* = \{x_i\}$, we create $\mathcal{G}_i$ from $\mathcal{G}_{i-1}$ by appending $|X_i| - 1$ edges and $|X_i| - 2$ vertices, in such a way that the edges corresponding to the symbols in X_i form a new cycle. It is easy to see that $\mathcal{G} = \mathcal{G}_r$ will contain exactly n edges and $n - (r-1)$ vertices. Furthermore, the code $\mathbb{C}^\perp$ generated by H is precisely the cycle code of $\mathcal{G}$. Since the cut-set code of $\mathcal{G}$ is the dual of its cycle code, our proof is complete. ∎

For example, the cycle-free codes defined by the parity-check matrices in (3.16) and (3.19) are cut-set codes of the graphs depicted in Figure 3.7a and Figure 3.7b, respectively.

For cut-set codes, it is well-known [29, 48] that $2n \geq md$, where m is the number of vertices in the underlying graph $\mathcal{G}$. Indeed, this follows immediately from the fact that if the minimum distance of $\mathbb{C}(\mathcal{G})$ is d, then every vertex of $\mathcal{G}$ must have degree at least d, otherwise the cut-set that isolates this vertex will have less than d edges. It is also well-known that $\dim \mathbb{C}(\mathcal{G}) = m - \omega(\mathcal{G})$, where $\omega(\mathcal{G})$ is the number of connected components in $\mathcal{G}$. Thus we obtain the following cut-set bound on the minimum distance of cycle-free codes

$$d \;\leq\; \frac{2n}{k + \omega(\mathcal{G})} \;\leq\; \frac{2n}{k+1} \tag{3.20}$$

If it is known that d is even, then the cut-set bound of (3.20) obviously implies Theorem 3.6. In general, however, Theorem 3.6 is stronger than the cut-set bound based on Theorem 3.11. Indeed, there exist cut-set codes that are not cycle-free. As a simple example, consider the $(6, 3, 3)$ cut-set

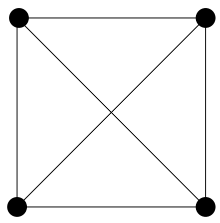


Figure 3.8: A cut-set code which is not cycle-free.

code of the graph depicted in Figure 3.8, and notice that the minimum distance of this code violates the upper bound of Theorem 3.6.

We have proved that every cycle-free binary linear code is a cut-set code. We now attempt to make this correspondence more precise by examining exactly which cut-set codes correspond to cycle-free binary linear codes. An answer to this question follows from a closer look at the construction in Theorem 3.11 and relies on *star graphs*, which are graphs where the removal of a vertex, which is connected to all the other vertices, results in a forest.

Theorem 3.12 *There is a one-to-one correspondence between cycle-free binary linear codes and cut-set codes of graphs whose dual is a star.*

Proof. Consider a cycle-free Tanner graph T for a given cut-set code, the corresponding graph $\mathcal{G}$ from the construction in Theorem 3.11, and its dual graph $\mathcal{G}^D$. Absent the vertex corresponding to the outer region of $\mathcal{G}$, each path in $\mathcal{G}^D$ corresponds to a path through the Tanner graph T: vertices in $\mathcal{G}^D$ correspond to check nodes of T, and edges in $\mathcal{G}^D$ correspond to symbol nodes of T. Since a path in T cannot contain a cycle, the same is true for $\mathcal{G}^D$ absent the outer vertex. Thus, $\mathcal{G}^D$ is a star corresponding to T.

The other direction of the correspondence is similarly straightforward. Given a cut-set code of a graph whose dual is a star, we may simply reverse the algorithm in Theorem 3.11 to arrive at a Tanner graph. This Tanner graph must be cycle free, or else $\mathcal{G}^D$ has a cycle among vertices other than the outer vertex. ∎

3.5.2 Asymptotics for Tanner graphs with cycles

It is obvious from Theorem 3.6 that Tanner graphs without cycles cannot support asymptotically good codes. Starting with Theorem 3.6, it is not difficult to show that the same is true for Tanner graphs with cycles, unless the number of cycles increases exponentially with the length of the code as $n \to \infty$. To see this, suppose the the cycle rank of a Tanner graph $T = (V, E)$ representing an (n, k, d) code $\mathbb{C}$ is $c = |E| - |V| + \omega(T)$. This means that T contains 2^c cycles and unions of disjoint cycles (cf. [48, p.137]). Now let x_i be a symbol vertex that lies on a cycle in T. Then removing x_i and all the edges incident on x_i from T produces a graph whose cycle rank is strictly less than c. This procedure is equivalent to shortening $\mathbb{C}$ at the i-th position to obtain an (n', k', d') code $\mathbb{C}'$ with $n' = n - 1$, $k' \geq k - 1$, and $d' \geq d$. Since the cycle rank strictly decreases each time we cut a cycle in T in this way, after repeating this procedure $t \leq c$ times we obtain a cycle-free code $\mathbb{C}^*$. Clearly $\mathbb{C}^*$ is an (n^*, k^*, d^*) code with $n^* = n - t$, $k^* \geq k - t$, and $d^* \geq d$. Thus Theorem 3.6 implies

$$d \;\leq\; d^* \;\leq\; 2\left\lfloor \frac{n^*}{k^*+1} \right\rfloor + 1 \;\leq\; 2\,\frac{n-t}{k-t+1} + 1 \tag{3.21}$$

Now let $\gamma = c/n$, and notice that $t/n \leq \gamma$. Hence if $\lim_{n\to\infty} \gamma = 0$, then (3.21) asymptotically reduces to $d \lesssim 2/R$, as in (3.1). Thus to support an asymptotically good sequence of codes, c must grow linearly with n, which means that the number of cycles 2^c grows exponentially with n. It would be useful to find out how the parameter $\gamma = c/n$, which has to do with the number of cycles, trades off versus the traditional asymptotic parameters $\delta = d/n$ and $R = k/n$ as $n \to \infty$. It would be also interesting to investigate, at least asymptotically, codes that have Tanner graphs of prescribed minimum girth.

3.5.3 General Tanner graphs without cycles

We now return to the case of general Tanner graphs, as defined in the introduction to the chapter, and observe that every general Tanner graph $(\mathcal{G}, \mathcal{L})$ can be converted into a simple Tanner graph for the same code through a vertex-splitting procedure. Indeed, let $y \in \mathcal{Y}$ be a check vertex in $\mathcal{G}$, let $\{x_{i_1}, x_{i_2}, \ldots, x_{i_\delta}\} \subseteq \mathcal{X}$ be the neighborhood of y, and let $\mathcal{C}$ be the corresponding constraint code of length δ. If $\dim \mathcal{C} = \kappa$, we split y into $\delta - \kappa$ vertices $y'_1, y'_2, \ldots, y'_{\delta-\kappa}$ and create edges between

$x_{i_1}, x_{i_2}, \ldots, x_{i_\delta}$ and $y'_1, y'_2, \ldots, y'_{\delta-\kappa}$ according to a parity-check matrix H for $\mathcal{C}$. An obvious but important observation is this: if H is cycle-free, then this procedure does not create new cycles. Thus we have proved the following statement.

Proposition 3.13 *If a linear code $\mathbb{C}$ can be represented by a general Tanner graph $(\mathcal{G}, \mathcal{L})$ such that $\mathcal{G}$ is cycle-free and all the constraints in $\mathcal{L}$ are cycle-free, then $\mathbb{C}$ can be represented by a simple Tanner graph without cycles.*

An immediate consequence of Proposition 3.13 is that all the results derived so far for simple Tanner graphs, including the bound of Theorem 3.6, straightforwardly extend to general Tanner graphs with cycle-free constraints.

In the general case, where check constraints are not necessarily cycle-free, it appears to be very difficult to say anything about the structure/properties of the code being represented. As an example, consider a general Tanner graph for $\mathbb{C}$ which contains a single check vertex $\mathcal{Y} = \{y\}$ with the corresponding constraint code being $\mathbb{C}$ itself. The existence of this cycle-free representation for $\mathbb{C}$ obviously does not provide any information whatsoever about $\mathbb{C}$.

Notwithstanding the trivial "counter-example" discussed above, it is plausible that if the underlying Tanner graph is cycle-free, the distance of $\mathbb{C}$ should be limited by the distances of the constraint codes in some manner. Furthermore, if simple decoding is sought, simple constraint codes must be used. It thus appears that the range of code parameters that are possible with cycle-free Tanner graphs will depend on the decoding complexity tolerated. We leave further investigation of this relation as an open problem.

Chapter 4

Conclusions and Future Directions

The problem of maximum likelihood decoding in the context of correction of transmission errors is known to be NP-complete in the general case [5]. In fact, it cannot even be approximated (within a constant factor) in polynomial time [54] and is fixed-parameter hard [16]. Because of this obvious bottleneck, a variety of decoding schemes have been proposed to deal with specific families of codes.

We have considered two popular decoding paradigms in this work: Viterbi decoding on time-indexed trellises and iterative decoding on factor graphs. In the former case we have presented a method for designing families of good error-correcting codes that satisfy specific decoding constraints, and in the latter case we have shown that current theoretical knowledge about the convergence of iterative decoding does not apply directly to asymptotically good codes.

4.1 Trellis decoding

In Chapter 2 we determined a generalized lexicographic construction which is capable of designing codes for efficient decoding. This construction can generate trellis-oriented codes, which locally minimize trellis complexity. More importantly, however, it also enables the design of codes whose decoding trellises are constrained, either by state complexity or by Viterbi decoding complexity.

Thus one may pick a block size in which to parse information (corresponding to the dimension k of the resulting error-correcting code) and a desired number t of errors which should be corrected, and then specify any one the following additional constraints:

1. the trellis should be locally oriented towards decoding efficiency,

2. or the trellis must occupy less than m bits of memory,

3. or the trellis must allow error-correction using at most $f(k)$ operations.

In most cases, Algorithm 2.12 then automatically provides a code which matches these criteria and generally has little redundancy.

For example, if we were to design a triple-error correcting code (i.e. $t = 3$) for encoding 8-bit messages, one version of the above constraints could produce:

1. a trellis-oriented code of length 19, which requires 64 trellis states and 715 operations for Viterbi decoding,

2. or a code of length 24 whose trellis has been constrained to 16 states and requires 301 operations for Viterbi decoding.

Using the technique in Section 2.5.2 we could also improve the state-constrained code above to one with a 16 state trellis requiring only 295 operations for Viterbi decoding.

The computations of Algorithm 2.12 are made possible by the theoretical backbone of Theorem 2.5. This theorem relates the cosets of successive codes in a generalized lexicographic construction. We have analyzed the bijective mapping describing the coset relationship and have presented a minor improvement over [9] on the parameter bounds of lexicodes.

4.1.1 Improving the construction

Many questions about generalized lexicodes remain unanswered, and there are several opportunities for improving their construction.

For one, it is still not clear why lexicodes, or even trellis-oriented $\mathfrak{G}$-codes, have such good code parameters. Perhaps the closest lead is that they can be regarded as the coding-theoretic analogs of laminated lattices over $\mathbb{R}^n$, which are the best known sphere packings in most low dimensions [12, 14]. Nevertheless, we suspect that the bounds on parameters of the generalized lexicodes can be improved by a more sophisticated count of the worst-case companion pairings. A quick glance at the tables in the appendix shows that the bounds in Section 2.3.1, though better than what is known so far, are still quite weak.

From the perspective of construction, one should note that the exponential time and space bound (with respect to the co-dimension) of Algorithm 2.12 may be improved with approximation techniques. The actual continuation of the algorithm depends only on the properties of a few coset leaders, and a heuristic approach to choosing these leaders might work well. Experimental results show that a straightforward strategy of ignoring additional coset leaders after a prescribed memory limit gives reasonable results for several iterations, but thereafter performance considerably degrades as the set of known coset leaders becomes stale. Thus, a more sensible heuristic is needed.

Speeding up Algorithm 2.12 would allow for the design of lower rate codes with good trellis decoding properties. However, it would also be interesting to derive complexity-theoretic performance measures for the decoding time needed for lexicodes and generalized lexicodes. At least for the trellis-oriented codes this result should follow from an analysis of Equation 2.9 which relates the decoding complexities of successive trellis-oriented codes by means of their differences in length (which correspond to their differences in covering radius). Thus, the decoding complexity of such code may be simply gleaned from the covering radii of preceding codes in its construction, though a closed-form for the result of this process is still needed.

Finally, the greedy construction of the lexicode family might lend itself to faster decoding than general codes. However, we suspect that the $\mathfrak{G}$-codes are defined broadly enough so as to be difficult to decode. That is, this class of codes is likely to contain hard instances of the decoding problem. Proving that it does, and discerning precisely which codes are not instances of the $\mathfrak{G}$-construction, is an interesting topic for future research.

4.1.2 Tanner graphs

Tanner graphs and their generalization to factor graphs have only recently gained renewed interest, and seem to show significant promise for future research. Our research in Chapter 3 uncovered some relations and properties of Tanner graphs.

The main result of the chapter shows that a code that can be represented by cycle-free Tanner graphs must necessarily have asymptotically poor error correction. This is particularly significant in light of the fact that the well-established decoding algorithms on Tanner graphs are only guaranteed to converge to the maximum-likelihood decoding for cycle-free graphs. In fact, the structure of cycle-free codes is so weak that exponentially many cycles are needed for asymptotically good error-correction. On the path towards these results, we determined precisely a family of cycle-free codes that exhibit the maximum possible error-correction, and we also establish a relationship between cycle-free codes and graph theoretic cut-set codes.

These various results lead to several natural extensions. For example, the connection between cycle-free codes and cut-set codes expressed in Section 3.5 suggests the possibility of a simpler graph-theoretical analysis of our main result. The details of our main result make its proof quite complicated, and it is possible that in a graph-theoretic setting it would follow more directly.

Another extension of our research involves a more careful analysis of the cycle structure of the Tanner graph of a code. Though it is impossible to entirely eliminate cycles in an asymptotically good code, it is possible to constrain the relative number of cycles in a graph so as to allow for many cycle-free subgraphs in which the *min-sum* decoding algorithm can operate efficiently. Specifically, it would be interesting to study how the parameter $\gamma = c/n$, which has to do with the relative number of cycles in a code, trades off versus the traditional asymptotic parameters $\delta = d/n$ and $R = k/n$ as $n \to \infty$.

Complexity of Min-Sum Decoding

In general, there have been few results about the complexity and performance of min-sum decoding on an arbitrary Tanner graph. As was stated in Chapter 3, the min-sum algorithm runs in $O(n^2)$ time for cycle-free Tanner graphs, but we have shown that cycle-free Tanner graphs represent essentially trivial codes. The intuition in Section 3.5.3 suggests that for the Tanner graph of a given code there is a trade-off between cycle-complexity and the computational complexity of constraint checking. It would be interesting to study convergence and computational properties of the min-sum algorithm on Tanner graphs that have cycles. Of specific interest is the case when this trade-off is optimized.

Furthermore, given the localized structure of the min-sum algorithm, one could imagine that it would be easily parallelizable. In fact, the algorithm in its most general form breaks down problems into localized subproblems by means of variable marginalization. Thus it might be interesting to pursue such marginalization as a method of automatic parallelization.

Boolean Satisfiability

There is an interesting link between maximum likelihood decoding and boolean formula satisfiability. Specifically, given a noise-corrupted input z, the problem of maximum-likelihood decoding on a binary channel is to find the vector c closest in Hamming distance to z which satisfies the following parity-check equation over $\mathbb{F}_2$:

$$Hc^T = 0. \tag{4.1}$$

We can translate Equation (4.1) into a satisfiability problem by associating each row in H with a clause of `XOR`'ed literals. Thus a row $[1, 0, 0, 1, 0, 0, 1]$ might correspond to $c_1 \oplus c_4 \oplus c_7$, where c_i refers to the value of the i-th input symbol and $\oplus$ denotes `XOR`. Finding a maximum-likelihood decoding is thus akin to finding the fewest bit flips of z needed to satisfy the conjunction of the negation of these clauses. This translation into the realm of boolean formulae opens the door to a vast array of literature in computer science.

4.2 Other works

We have also examined other problems during the course of developing this dissertation. For one, we have looked at full-rank tilings in eight and nine dimensions [59], based on an open problem in [21]. Such tilings are related to the existence of perfect codes. We modified computational techniques based on the graph-isomorphism problem, pioneered in [46], and added some manual analysis to prove the non-existence of these tilings.

Secondly, we have looked at the problem of reconciling two physically separated, unordered databases whose contents are related [44, 57]. Specifically, we have considered determining the mutual difference of these databases with a minimum communication complexity. This type of problem arises naturally from gossip protocols used for the distribution of information. We analyzed two instances of the reconciliation problem, a client-server model and a more general peer-to-peer model, and provided interactive solutions for both models. For the former instance, we also provided a simple one-message reconciliation algorithm, based on elementary symmetric polynomials, which has an almost optimal communication complexity. Finally, we demonstrated several applications of our reconciliation algorithms, including an improvement of [30]'s solution of the resource discovery problem.

Appendix A

Generalized Lexicodes

We show data gleaned from the implementation of many of the algorithms in Chapter 2. An older version of some of this data appeared in [56].

A.1 Trellis-oriented $\mathfrak{G}$-codes

The following tables depicts properties of some distance 4, 6, and 8 lexicodes and trellis-oriented $\mathfrak{G}$-codes. Over a binary field, lexicodes with odd minimum distance are simply punctured lexicodes with even minimum distance, so it is redundant to list their properties [14].

The minimum distance 4 lexicodes are all extended Hamming codes or shortenings thereof [14]. The minimum distance 6 and 8 codes listed also have optimal error-correction capability in the sense that each has the best known minimum distance for its length and dimension as compared to [8]. We can see that the trellis-oriented codes almost always have lower decoding complexity than the corresponding lexicodes.

Table A.1: Parameters of $\mathbf{d} = \mathbf{4}$ codes. We display the following parameters for lexicodes and trellis-oriented $\mathfrak{G}$-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>
</table>

continued on the next page...

Table A.1 (continued): Parameters of the $\mathbf{d} = \mathbf{4}$ codes

Dim-ension	*Lexicodes* Length	*Trellis-oriented* Length	*Lexicodes* States	*Trellis-oriented* States	*Lexicodes* $2\|E\| - \|V\| + 1$	*Trellis-oriented* $2\|E\| - \|V\| + 1$
22	28		5		1,339	
23	29		5		1,603	
24	30		5		1,627	
25	31		5		1,699	
26	32		5		1,723	
27	34		5		1,737	
28	35		5		1,775	
29	36		5		1,791	
30	37		5		1,931	
31	38		5		1,955	
32	39		5		2,027	
33	40		5		2,051	
34	41		5		2,643	
35	42		5		2,667	
36	43		5		2,739	
37	44		5		2,763	
38	45		5		3,027	
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	

continued on the next page...

Table A.1 (continued): Parameters of the $\mathbf{d = 4}$ codes

Dim-ension	*Lexicodes* Length	*Trellis-oriented* Length	*Lexicodes* States	*Trellis-oriented* States	*Lexicodes* $2\|E\|-\|V\|+1$	*Trellis-oriented* $2\|E\|-\|V\|+1$
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	

continued on the next page...

Table A.1 (continued): Parameters of the $\mathbf{d} = \mathbf{4}$ codes

Dim-ension	*Lexicodes* Length	*Trellis-oriented* Length	*Lexicodes* States	*Trellis-oriented* States	*Lexicodes* $2\|E\| - \|V\| + 1$	*Trellis-oriented* $2\|E\| - \|V\| + 1$
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	
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	

continued on the next page...

Table A.1 (continued): 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* $2\|E\| - \|V\| + 1$
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	

continued on the next page...

Table A.1 (continued): Parameters of the $\mathbf{d = 4}$ codes

Dim-ension	*Lexicodes* *Length*	*Trellis-oriented* *Length*	*Lexicodes* *States*	*Trellis-oriented* *States*	*Lexicodes* $2\|E\| - \|V\| + 1$	*Trellis-oriented* $2\|E\| - \|V\| + 1$
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 A.2: Parameters of $\mathbf{d} = \mathbf{6}$ codes. We display the following parameters for lexicodes and trellis-oriented $\mathfrak{G}$-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>
</table>

continued on the next page...

Table A.2 (continued): Parameters of the $\mathbf{d = 6}$ lexicodes

<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>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>
<tr><td>39</td><td>52</td><td>53</td><td colspan="2">12</td><td>178,651</td><td>146,493</td></tr>
<tr><td>40</td><td>53</td><td>54</td><td colspan="2">12</td><td>190,939</td><td>146,783</td></tr>
<tr><td>41</td><td colspan="2">55</td><td colspan="2">12</td><td>203,229</td><td>147,555</td></tr>
<tr><td>42</td><td colspan="2">56</td><td colspan="2">12</td><td>215,519</td><td>150,251</td></tr>
<tr><td>43</td><td colspan="2">57</td><td colspan="2">12</td><td>221,667</td><td>151,035</td></tr>
<tr><td>44</td><td colspan="2">58</td><td colspan="2">12</td><td>240,107</td><td>151,835</td></tr>
</table>

continued on the next page...

Table A.2 (continued): Parameters of the $\mathbf{d} = \mathbf{6}$ lexicodes

Dim-ension	*Lexicode* Length	*Trellis-oriented* Length	*Lexicode* States	*Trellis-oriented* States	*Lexicode* $2\|E\| - \|V\| + 1$	*Trellis-oriented* $2\|E\| - \|V\| + 1$
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*

continued on the next page...

Table A.2 (continued): Parameters of the $\mathbf{d} = \mathbf{6}$ lexicodes

<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>68</td><td colspan="2">83</td><td colspan="2">13</td><td>819,675</td><td>671,195</td></tr>
<tr><td>69</td><td colspan="2">84</td><td>13</td><td>14</td><td>834,011</td><td>913,883</td></tr>
<tr><td>70</td><td colspan="2">85</td><td>13</td><td>14</td><td>930,267</td><td>916,955</td></tr>
<tr><td>71</td><td colspan="2">86</td><td>13</td><td>14</td><td>954,843</td><td>938,459</td></tr>
<tr><td>72</td><td colspan="2">87</td><td>13</td><td>14</td><td>979,419</td><td>956,891</td></tr>
<tr><td>73</td><td colspan="2">88</td><td colspan="2">14</td><td>1,487,323</td><td>1,030,619</td></tr>
<tr><td>74</td><td colspan="2">89</td><td colspan="2">14</td><td>1,490,395</td><td>1,055,195</td></tr>
<tr><td>75</td><td>90</td><td>91</td><td colspan="2">14</td><td>1,518,043</td><td>1,055,293</td></tr>
<tr><td>76</td><td>91</td><td>92</td><td colspan="2">14</td><td>1,548,763</td><td>1,055,583</td></tr>
<tr><td>77</td><td>92</td><td>93</td><td colspan="2">14</td><td>1,659,355</td><td>1,056,355</td></tr>
<tr><td>78</td><td>93</td><td>94</td><td colspan="2">14</td><td>1,683,931</td><td>1,056,459</td></tr>
<tr><td>79</td><td colspan="2">95</td><td colspan="2">14</td><td>1,782,237</td><td>1,058,491</td></tr>
<tr><td>80</td><td colspan="2">96</td><td colspan="2">14</td><td>1,806,815</td><td>1,059,483</td></tr>
<tr><td>81</td><td colspan="2">97</td><td colspan="2">14</td><td>1,880,547</td><td>1,066,843</td></tr>
<tr><td>82</td><td colspan="2">98</td><td colspan="2">14</td><td>1,929,707</td><td>1,067,355</td></tr>
<tr><td>83</td><td colspan="2">99</td><td colspan="2">14</td><td>1,978,875</td><td>1,079,515</td></tr>
<tr><td>84</td><td colspan="2">100</td><td colspan="2">14</td><td>2,003,483</td><td>1,109,979</td></tr>
<tr><td>85</td><td colspan="2">101</td><td colspan="2">14</td><td>2,077,275</td><td>1,121,755</td></tr>
<tr><td>86</td><td colspan="2">102</td><td colspan="2">14</td><td>2,101,979</td><td>1,127,899</td></tr>
<tr><td>87</td><td colspan="2">103</td><td colspan="2">14</td><td>2,108,379</td><td>1,212,891</td></tr>
<tr><td>88</td><td colspan="2">104</td><td colspan="2">14</td><td>2,139,611</td><td>1,225,179</td></tr>
<tr><td>89</td><td colspan="2">105</td><td colspan="2">14</td><td>2,177,499</td><td>1,419,739</td></tr>
<tr><td>90</td><td colspan="2">106</td><td colspan="2">14</td><td>2,183,643</td><td>1,976,795</td></tr>
</table>

continued on the next page...

Table A.2 (continued): Parameters of the $\mathbf{d} = \mathbf{6}$ lexicodes

Dim-ension	*Lexicode* Length	*Trellis-oriented* Length	*Lexicode* States	*Trellis-oriented* States	*Lexicode* $2\|E\| - \|V\| + 1$	*Trellis-oriented* $2\|E\| - \|V\| + 1$
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 A.3: Parameters of $\mathbf{d} = \mathbf{8}$ codes. We display the following parameters for lexicodes and trellis-oriented $\mathfrak{G}$-codes: length, dimension, max. number of states and Viterbi decoding complexity with the BCJR trellis.

Dim-ension	*Standard* Length	*Trellis-oriented* Length	*Standard* States	*Trellis-oriented* States	*Standard* $2\|E\| - \|V\| + 1$	*Trellis-oriented* $2\|E\| - \|V\| + 1$
1	8		1		17	
2	12		2		35	
3	14		3		71	
4	15		4		143	
5	16		4		195	
6	18		5		341	
7	19		6		647	
8	20		6		779	
9	21		7		1,547	
10	22		8		2,395	
11	23		9		4,219	
12	24		9		4,475	
13	28		9		4,529	
14	30		9		4,777	
15	31		9		5,463	
16	32		9		5,515	
17	34		9		7,645	*5,693*
18	35		9		12,671	*6,143*
19	36		9		12,803	*6,275*
20	37		10	*9*	24,267	*7,691*
21	38		10	*9*	25,115	*8,539*

continued on the next page...

Table A.3 (continued): Parameters of the $\mathbf{d} = \mathbf{8}$ lexicodes

Dimension	*Standard* *Length*	*Trellis-oriented* *Length*	*Standard* *States*	*Trellis-oriented* *States*	*Standard* $2\lvert E\rvert - \lvert V\rvert + 1$	*Trellis-oriented* $2\lvert E\rvert - \lvert V\rvert + 1$
22	39		10	*9*	26,939	*10,363*
23	40		10	*9*	27,195	*10,619*
24	42		10		31,805	*17,853*
25	43		11		41,791	*33,087*
26	44		11		41,987	*33,283*
27	45		12		65,227	
28	46		12		67,163	
29	47		12		72,571	*78,203*
30	48		12		72,827	*78,459*
31	49	*50*	13	*12*	135,611	*80,317*
32	50	*51*	13	*12*	169,019	*87,103*
33	51	*52*	13	*12*	248,123	*88,643*
34	52	*53*	13		248,507	*137,547*
35	53	*54*	14	*13*	427,579	*138,331*
36	54	*55*	14	*13*	431,419	*142,203*
37	55	*56*	14	*13*	442,171	*142,459*
38	56	*57*	14		442,555	*274,875*
39	58		14		487,997	*308,283*
40	59		14		628,031	*457,019*
41	60		14		628,227	*460,091*
42	62		14		629,197	*460,861*
43	63		14		631,903	*464,703*
44	64		14		632,035	*464,899*

continued on the next page...

Table A.3 (continued): Parameters of the $\mathbf{d} = \mathbf{8}$ lexicodes

Dim-ension	*Standard* *Length*	*Trellis-oriented* *Length*	*Standard* *States*	*Trellis-oriented* *States*	*Standard* $2\|E\| - \|V\| + 1$	*Trellis-oriented* $2\|E\| - \|V\| + 1$
45	65		15	*14*	1,263,819	*581,835*
46	66		15		1,287,195	*1,053,275*
47	67		15	*16*	1,346,235	*2,106,299*
48	68	*69*	15	*16*	1,701,883	*2,106,557*
49	70		15	*16*	1,702,397	*2,111,487*
50	71		15	*16*	1,704,575	*2,115,203*
51	72		15	*16*	2,041,731	*2,232,203*
52	73		15	*16*	2,126,219	*2,539,931*
53	74		16		3,835,291	*3,351,995*
54	75		16	*17*	3,976,635	*6,123,003*
55	76	*77*	16	*17*	4,849,147	*6,123,389*
56	77	*78*	17		7,937,659	*6,125,439*

A.2 State-bounded $\mathfrak{G}$-Codes

The following tables contain the code parameters for trellis state-bounded $\mathfrak{G}$-codes with log-state bounds of 4 (i.e. $2^4 = 16$ states maximum), 5 (i.e. $2^5 = 32$ states maximum), and 6 (i.e. $2^6 = 64$ states maximum) respectively. For each of these bounds, we list the length of the generated code for each of the minimum distances 4 through 8. Blank entries correspond to codes we have not computed due to limited computational resources. The distance 6 codes have also been plotted in Figure A.7.

Table A.4: Codes with a trellis log-state bound of 4. Displayed are lengths of the $\mathfrak{G}$-codes generated by $f_{\text{state}}(\cdot)$ with a *maximum log-state bound of 4* (i.e. 16 states), listed by dimension and minimum distance d.

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			

continued on the next page...

Table A.4 (continued): Codes with a trellis log-state bound of 4

Dimension	d=4	d=5	d=6	d=7	d=8
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				

Table A.5: Codes with a trellis log-state bound of 5. Displayed are lengths of the $\mathfrak{G}$-codes generated by $f_{\text{state}}(\cdot)$ with a *maximum log-state bound of 5* (i.e. 32 states), listed by dimension and minimum distance d.

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		

continued on the next page...

Table A.5 (continued): Codes with a trellis log-state bound of 5

Dimension	*d=4*	*d=5*	*d=6*	*d=7*	*d=8*
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				

Table A.6: Codes with a trellis log-state bound of 6. Displayed are lengths of the $\mathfrak{G}$-codes generated by $f_{\text{state}}(\cdot)$ with a *maximum log-state bound of 6* (i.e. 64 states), listed by dimension and minimum distance d.

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		

continued on the next page...

Table A.6 (continued): Codes with a trellis log-state bound of 4

Dimension	*d=4*	*d=5*	*d=6*	*d=7*	*d=8*
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				

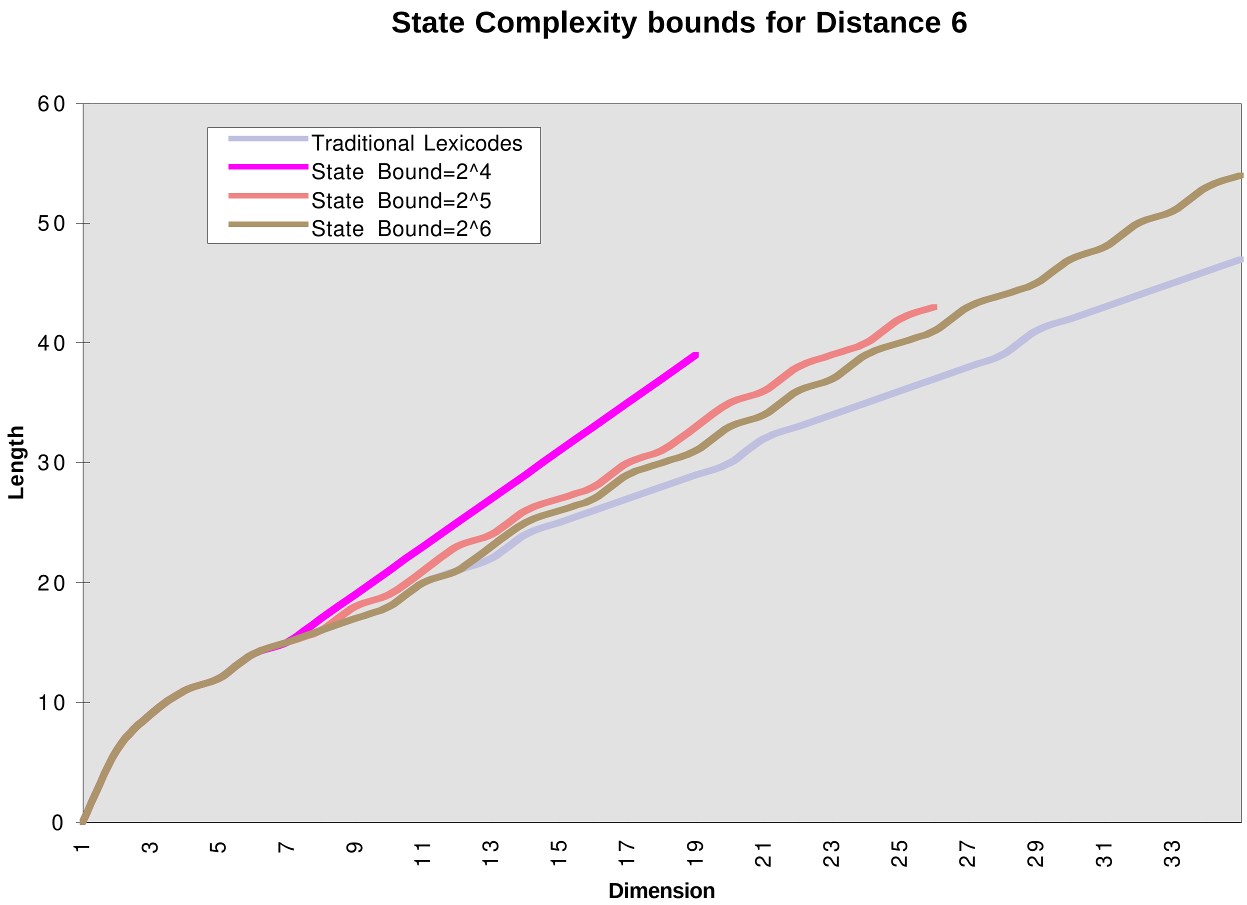


Figure A.7: The effects of bounding the *state complexity* of distance 6 $\mathfrak{G}$-codes with various constants.

A.3 Bounding decoding complexity

Figures A.8 and A.9 show the results of applying various Viterbi decoding constraints to $\mathfrak{G}$-codes. Specifically, Figures A.8 and A.9 show the decoding complexity achieved under linear and quadratic decoding bounds. The degradation in code length, compared to the standard lexicodes, is modest for these examples.

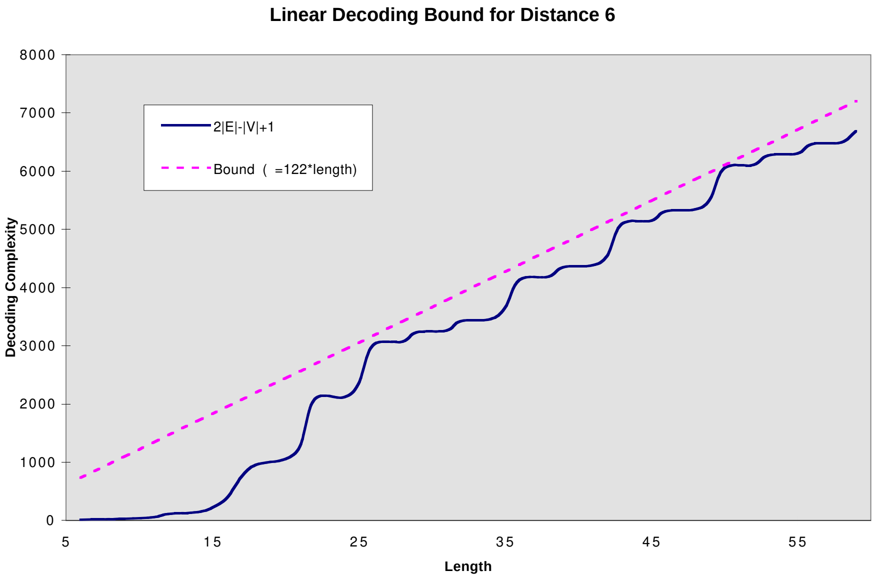


Figure A.8: The effects of bounding the decoding complexity of distance 6 $\mathfrak{G}$-codes with a *linear* function.

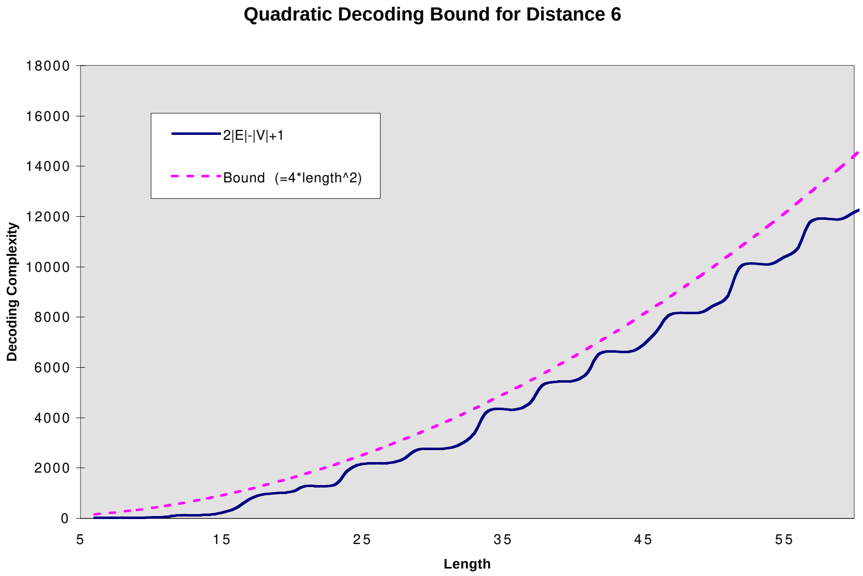


Figure A.9: The effects of bounding the decoding complexity of distance 6 $\mathfrak{G}$-codes with a *quadratic* function.

Appendix B

Tanner Graphs

We will show that each of the three relations (3.11),(3.13),(3.14) between n, k, and d derived in Section 4.2 implies (3.9), providing d is an integer in (3.11),(3.13) and d is an odd integer in (3.14). In order to make the appendix self-contained, we now re-state these inequalities:

$$d \;\leq\; 2\left\lfloor \frac{n-d}{k-1} \right\rfloor \tag{9}$$

$$d \;\leq\; 2\,\frac{n-d+1}{k-1} - 2 \tag{11}$$

$$d \;\leq\; 2\,\frac{n-d}{k-1} - 1 \tag{12}$$

Notice that what we are trying to establish has nothing to do with graphs or codes; this is just manipulation of integer inequalities. In particular, we have following simple lemma.

Lemma 11. *If $a \leq b/c$ and a, b, c are positive integers, then $a \leq (b+a)/(c+1)$.*

The proof of Lemma 11 is straightforward, and is left to the reader. We first deal with (3.14), assuming d is odd. Taking the common denominator and applying (twice) Lemma 11, we see that (3.14) implies

$$d \;\leq\; \frac{2n-(k-1)}{k+1} \;=\; \frac{2(n+1)}{k+1} - 1 \tag{20}$$

Since $(d+1)/2$ is an integer for odd d, it follows from (20) that $(d+1)/2 \leq \lfloor (n+1)/(k+1) \rfloor$. This may be re-written as:

$$d \;\leq\; 2\left\lfloor \frac{n+1}{k+1} \right\rfloor - 1 \tag{21}$$

If $n+1 \not\equiv 0 \mod (k+1)$ then $\lfloor (n+1)/(k+1) \rfloor = \lfloor n/(k+1) \rfloor$, and (21) clearly implies (3.9). If $(n+1)/(k+1)$ is an integer, then (21) is precisely the equivalent form of (3.9) given in (3.12).

It is easy to see that if d is an odd integer, then (3.11) implies (3.14). Since this case was already established above, it remains to prove (3.11) for even d. Using once again Lemma 11, we see that (3.11) implies $d \leq 2n/(k+1)$, or equivalently $d/2 \leq n/(k+1)$. Since $d/2$ is an integer for even d, we can take the integer part of $n/(k+1)$ in the above expression. It follows that for even d, we have

$$d \;\leq\; 2\left\lfloor \frac{n}{k+1} \right\rfloor$$

which clearly implies (3.9). Finally, it can be readily seen that if d is an integer, then (3.13) implies (3.11). Hence, our proof of Theorem 5 is now complete.

References

[1] S.M. Aji and R.J. McEliece. A general algorithm for distributing information in a graph. *Proc. IEEE Int. Symp. Inform. Theory*, page 6, 1997.

[2] S.M. Aji and R.J. McEliece. The generalized distributive law. *IEEE Transactions on Information Theory*, July 1998. Submitted for publication.

[3] L.R. Bahl, J. Cocke, F. Jelinek, and J. Raviv. Optimal decoding of linear codes for minimizing symbol error rate. *IEEE Transactions on Information Theory*, 20:284–287, 1974.

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

[5] E.R. Berlekamp, R.J. McEliece, and H.C.A. van Tilborg. On the inherent intractability of certain coding problems. *IEEE Transactions on Information Theory*, 24:384–386, 1978.

[6] C. Berrou and A. Glavieux. Near optimum error correcting coding and decoding: turbo-codes. *IEEE Trans. Commun.*, 44:1261–1271, 1996.

[7] R.E. Blahut. Theory and practice of data transmission codes, 1994.

[8] A.E. Brouwer and T. Verhoeff. An updated table of minimum-distance bounds for binary linear codes. *IEEE Transactions on Information Theory*, 39:662–677, 1993.

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

[10] J. Bruck and M. Blaum. Neural networks, error-correcting codes, and polynomials over the binary n-cube. *IEEE Transactions on Information Theory*, 35:976–987, 1989.

[11] A. Glavieux C. Berrou and P. Thitimajshima. Near Shannon limit error-correcting coding and decoding: turbo codes. *Proc. IEEE Int. Conf. on Communications*, pages 1064–1070, 1993.

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

[13] J.H. Conway. Integral lexicographic codes. *Discrete Math*, 83:219 – 235, 1990.

[14] J.H. Conway and N.J.A. Sloane. Lexicographic codes: error-correcting codes from game theory. *IEEE Transactions on Information Theory*, 32:337–348, 1986.

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

[16] R.G. Downey, M.R. Fellows, A. Vardy, and G. Whittle. On the parametrized complexity of certain fundamental problems in coding theory. To appear.

[17] M. Esmaeili and A.K. Khandani. Acyclic tanner graphs and maximum-likelihood decoding of linear block codes. preprint, May 1998.

[18] T. Etzion, A. Trachtenberg, and A. Vardy. Which codes have cycle-free tanner graphs? Technical report, Technion - Israel Institute of Technology, November 1997. Technical Report #0925.

[19] T. Etzion, A. Trachtenberg, and A. Vardy. Which codes have cycle-free tanner graphs? In *1998 IEEE International Symposium on Information Theory*, Cambridge, MA, 1998.

[20] T. Etzion, A. Trachtenberg, and A. Vardy. Which codes have cycle-free tanner graphs? *IEEE Transactions on Information Theory*, 45(6):2173 – 2180, September 1999.

[21] T. Etzion and A. Vardy. On perfect codes and tilings: Problems and solutions. *SIAM Journal of Discrete Math*, 11(2):205–233, May 1998.

[22] J. Feigenbaum, G.D. Forney Jr., B.H. Marcus, R.J. McEliece, and A. Vardy. Special issue on "codes and complexity". *IEEE Transactions on Information Theory*, 42:1649–2064, November 1996.

[23] B.J. Frey. *Baysean networks for pattern classification, data compression, and channel coding.* PhD thesis, University of Toronto, Canada, July 1997. Department of Electrical Engineering.

[24] B.J. Frey, F.R. Kschischang, H.A. Loeliger, and N. Wiberg. Factor graphs and the sum-product algorithm. In preparation, 1997.

[25] R.G. Gallager. Low-density parity-check codes. *IRE Trans. Inform. Theory*, 8:21–28, January 1962.

[26] R.G. Gallager. *Low-Density Parity-Check Codes.* MIT Press, Cambridge, MA, 1963.

[27] R.L. Graham and N.J.A. Sloane. On the covering radius of codes. *IEEE Transactions on Information Theory*, 31:385–401, 1985.

[28] S.L. Hakimi and J. Bredeson. Graph theoretic error-correcting codes. *IEEE Transactions on Information Theory*, 14:584–591, 1968.

[29] S.L. Hakimi and H. Frank. Cut-set matrices and linear codes. *IEEE Transactions on Information Theory*, 11:457–458, 1965.

[30] Mor Harchol-Balter, Tom Leighton, and Daniel Lewin. Resource discovery in di stributed networks. In *18th Annual ACM-SIGACT/SIGOPS Symposium on Principles of Distributed Computing*, Atlanta, GA, May 1999.

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

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

[33] G.D. Forney Jr. The forward-backward algorithm. In *Proc. 34-th Annual Allerton Conference on Communications, Control and Computing*, Monticello, IL., October 1996.

[34] F.R. Kschischang and G.B. Horn. A heuristic for ordering a linear block code to minimize trellis state complexity. In *Proc. 32-nd Annual Allerton Conference on Communications, Control and Computing*, pages 75–84, Monticello, IL, September 1994.

[35] F.R. Kschischang and V. Sorokine. On the trellis structure of block codes. *IEEE Transactions on Information Theory*, 41:1924–1937, 1995.

[36] A. Lafourcade and A. Vardy. Lower bounds on trellis complexity of block codes. *IEEE Transactions on Information Theory*, 41(6), November 1995.

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

[38] 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.*, 32:456–471, 1984.

[39] D.J.C. MacKay. Gallager codes that are better than turbo codes. In *Proc. 36-th Allerton Conference on Communications, Control and Computing*, Monticello, IL., September 1998.

[40] D.J.C. MacKay. Good error-correcting codes based on very sparse matrices. *IEEE Transactions on Information Theory*, 1999. To appear.

[41] D.J.C. MacKay and R.M. Neal. Near Shannon limit performance of low-density parity-check codes. *Electronics Letters*, 32:1645–1646, 1996.

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

[43] R.J. McEliece. On the BCJR trellis for linear block codes. *IEEE Transactions on Information Theory*, 42:1072–1092, 1996.

[44] Y. Minsky and A. Trachtenberg. Efficient reconciliation of undoredered databases. *41st Annual ACM Symposion on Theory of Computing*, May 1999. Submitted.

[45] D.J. Muder. Minimal trellises for block codes. *IEEE Transactions on Information Theory*, 34:1049–1053, 1988.

[46] P.R.J. Ostergard, T. Baicheva, and E. Kolev. Optimal binary one-error-correcting codes of length 10 have 72 codewords. *IEEE Transactions on Information Theory*, 45(4):1229–1231, May 1999.

[47] J. Pearl. *Probabilistic Reasoning in Intelligent Systems:Networks of Plausible Inference*. Kaufmann, San Mateo, CA, 1988.

[48] W.W. Peterson and Jr. E.J. Weldon. *Error-Correcting Codes*. MIT Press, Cambridge, MA, 2-nd edition, 1961.

[49] C.E. Shannon. A mathematical theory of communication. *Bell Syst. Tech. J.*, 27:379–423, 623–656, 1948.

[50] M. Sipser and D.A. Spielman. Expander codes. *IEEE Transactions on Information Theory*, 42:1710–1722, 1996.

[51] P. Solé and T. Zaslavsky. The covering radius of the cycle code of a graph. *Discrete Applied Math.*, 45:63–70, 1993.

[52] P. Solé and T. Zaslavsky. A coding approach to signed graphs. *SIAM J. Discrete Math.*, 7:544–553, 1994.

[53] D.A. Spielman. Linear-time encodable and decodable codes. *IEEE Transactions on Information Theory*, 42:1723–1731, 1996.

[54] J. Stern. Approximating the number of error locations within a constant ratio is NP-complete. *Lect. Notes Comp. Science*, 673:325–331, 1993.

[55] R.M. Tanner. A recursive approach to low-complexity codes. *IEEE Transactions on Information Theory*, 27:533–547, 1981.

[56] A. Trachtenberg. Computational methods in coding theory. Master's thesis, University of Illinois at Urbana-Champaign, 1996.

[57] A. Trachtenberg and Y. Minsky. Efficient reconciliation of unordered databases. Technical report, Cornell University, Ithaca, NY, November 1999.

[58] A. Trachtenberg and A. Vardy. Designing lexicographic codes for a given trellis complexity. In preparation.

[59] A. Trachtenberg and A. Vardy. Full rank tilings in eight and nine dimensions. In preparation.

[60] A. Trachtenberg and A. Vardy. Lexicographic codes: constructions, bounds, and trellis complexity. In *Proc. 31st Annual Conference on Information Sciences and Systems*, pages 521–527, Baltimore, MD, March 1997.

[61] A. Vardy. Algorithmic complexity in coding theory and the minimum distance problem. In *Twenty-ninth annual ACM symposium on Theory of computing*, 1997.

[62] A. Vardy. Trellis structure of codes. In V.S. Pless and W. Cary Huffman, editors, *Handbook of Coding Theory*. Elsevier Science Publishers, Amsterdam, 1998.

[63] A.J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13:260–269, 1967.

[64] D.B. West. *Introduction to Graph Theory*. Prentice-Hall, Englewood Cliffs, NJ, 1996.

[65] N. Wiberg. *Codes and Decoding on General Graphs*. PhD thesis, Linkoping University, 1996.

[66] N. Wiberg, H.-A. Loeliger, and R. Kötter. Codes and iterative decoding on general graphs. *Euro. Trans. Telecommun.*, 6:513–526, 1995.

[67] S. Zhang. Design of linear block codes with fixed state complexity. Master's thesis, University of Toronto, 1996.

Vita

Ari Trachtenberg was born in Haifa, Israel in 1973. He received his S.B. degree from the Massachusetts Institute of Technology in 1994 in mathematics with computer science, and his M.S. degree from the University of Illinois at Urbana-Champaign in 1996 in computer science.

At the University of Illinois, he was a University Fellow and later a Computational Science and Engineering Fellow from 1994 to 1997. In the summer of 1997, he was a research intern at Hewlett Packard Laboratories, Palo Alto, CA and, in the summers of 1998 and 1999, a computer science instructor with the Center for Talented Youth at the Johns Hopkins University. Throughout 1997 and 1998, Ari was a teaching assistant in the computer science department at the University of Illinois, and thereafter he was a research assistant at the Coordinated Science Laboratory.

He was awarded the Mavis Memorial Fund Scholarship from the College of Engineering in 1999, based on academic performance, interest in engineering education, and published research. His research interests include coding theory (trellis and iterative decoding), algorithmic complexity theory, and cryptography.