EINDHOVEN UNIVERSITY OF TECHNOLOGY

Modulation codes

Citation for published version (APA):
Hollmann, H. D. L. (1996). *Modulation codes*. [Phd Thesis 2 (Research NOT TU/e / Graduation TU/e), Mathematics and Computer Science]. Technische Universiteit Eindhoven. https://doi.org/10.6100/IR473849

DOI:
10.6100/IR473849

Document status and date:
Published: 01/01/1996

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 04. Oct. 2023

Modulation codes

Modulation codes

Proefschrift

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. M. Rem, voor een commissie aangewezen door het College van Dekanen in het openbaar te verdedigen op

dinsdag 17 december 1996 om 16.00 uur

door

Hendrik Dirk Lodewijk Hollmann

geboren te Utrecht

Dit proefschrift is goedgekeurd door de promotoren:
prof.dr. J.H. van Lint
en
prof.dr. P.H. Siegel

CIP-gegevens Koninklijke Bibliotheek, Den Haag
Hollmann, H.D.L.
Modulation codes
Proefschrift Technische Universiteit Eindhoven,-Met lit. opg.,
-Met samenvatting in het Nederlands.
ISBN 90-74445-32-2
Trefw.: modulatie code, digitaal, communicatie

The work described in this thesis has been carried out
at the PHILIPS RESEARCH LABORATORIES Eindhoven,
the Netherlands, as part of the Philips Research Programme.

©Philips Electronics N.V. 1996
All rights are reserved. Reproduction in whole or in part is
prohibited without the written consent of the copyright owner.

Acknowledgments

This work has been carried out at the Philips Research Laboratories in Eindhoven. I am indebted to the management, and in particular to my group leader Cees Stam, for providing the opportunity to write this thesis. Furthermore, I warmly thank my colleagues Ludo Tolhuizen and Stan Baggen, who in various stages read most of this work and who proposed innumerable improvements.

It is a pleasure for me to express my gratitude to Kees Schouhamer Immink, co-author of Chapters II and III, who introduced me to the field of modulation codes and to many of the people involved. This work has greatly benefited from his guidance and insight, and from the many stimulating discussions that we shared.

I also wish to thank my promotor and former teacher Jack van Lint for valuable suggestions concerning the presentation of my work. His everlasting enthusiasm for and interest in mathematics has always been a great example to me. I was proud to be his pupil, and it is an honour to have him as my promotor now.

The thoughtful criticisms and suggestions of Brian Marcus have added greatly to the second part of this work. His careful reading of the manuscript saved me from some later embarrassment. I am very grateful for his stimulations and his interest in my work.

I would also like to thank my promotor Paul Siegel and the other members on my committee for their time and efforts.

This thesis is typeset with LaTeX, in the teTeX distribution from Thomas Esser and using Times-Roman fonts in combination with the mathtime font package of Michael Spivak. Special thanks are due to Jan van der Steen and Olaf Weber, both from CWI, Amsterdam, and to my colleague Ronald van Rijn, for their indispensable help with the installation of teTeX and for many other efforts.

Finally, a most warm thanks goes to my wife Brigitte and our kids Kirsten, Sanne, Line, and Tommie, for their encouragement and support—offered on the scarce occasions when we did meet :-)

Contents

1 Introduction **1**

1.1 Digital communication 2

1.1.1 A/D conversion 3

1.1.2 Data-compression 4

1.1.3 Protection against errors; error-correcting codes 6

1.2 Modulation codes 11

1.2.1 Type of constraints; constrained systems 11

1.2.2 Coding rate and capacity 14

1.2.3 The capacity of constraints of finite type 16

1.2.4 Sofic systems and their capacity 20

1.2.5 Encoding and encoders 25

1.2.6 Decoding and decoders 27

1.2.7 Synchronization 29

1.2.8 Code-construction methods 30

1.3 Overview of the contents of this thesis 45

References 47

2 Performance of Efficient Balanced Codes **53**

2.1 Introduction 53

2.2 Balancing of Codewords 55

2.3 Spectrum and sum variance of sequences 56

2.4 A counting problem 58

2.5 Performance Appraisal 65

2.6 Conclusions 65

Acknowledgement 66

References 66

3 Prefix-synchronized Runlength-limited Sequences **69**

3.1 Introduction 70

3.2 Preliminaries 71
3.3 Enumeration of Sequences 72
3.3.1 Capacity 77
3.4 Code Design 79
3.4.1 Graph description 80
3.4.2 Results 82
3.4.3 Worked example 84
3.4.4 Incidental constraints 85
3.5 Conclusions 86
Appendix: Approximations to the capacity bounds 87
References 88

4 On the construction of bounded-delay encodable codes for constrained systems 91
4.1 Introduction 91
4.2 Codes for constraints of finite type 95
4.3 Look-ahead encoders and BD codes 99
4.4 Local encoder structures and BD partitions 105
4.5 A general construction method 113
4.6 Finite-state encoders from look-ahead encoders 120
4.7 Further examples 125
4.7.1 A new rate-2/5 (2, 18, 2)-constrained code 125
4.7.2 A new rate-2/3 (1, 6)-constrained code 130
4.7.3 A new rate-2/3 (1, 9)-constrained code 135
4.8 Conclusions 138
Acknowledgments 138
4.9 Appendix A: additions to Section 4.4 139
4.9.1 Proof of Theorem 3 139
4.9.2 Proof of Theorem 4 141
4.9.3 Some comments on Theorem 4 141
4.9.4 The complexity problem for BD partitions 142
4.10 Appendix B: connection with the ACH algorithm 143
4.11 Appendix C: BD codes from ACH 148
References 150

5 Bounded delay encodable, block-decodable codes for constrained systems 153
5.1 Introduction 153
5.2 Preliminaries 155
5.3 BDB codes 159

5.4 Principal state-sets for one-symbol look-ahead BDB codes 163
5.5 State-trees for M-symbol look-ahead BDB codes 166
5.6 The state combination method 173
5.7 Stable state-sets 177
5.8 Discussion 181
Acknowledgments 181
Appendix A 181
Appendix B 183
References 185

6 A block-decodable $(d, k) = (1, 8)$ runlength-limited rate $8/12$ code 187
6.1 Introduction 187
6.2 Principal state-sets for BDB codes 189
6.3 Begin and end types of code words 191
6.4 Description of the code 193
6.5 Discussion 195
Acknowledgement 196
References 197

7 On an approximate eigenvector associated with a modulation code 199
7.1 Introduction 199
7.2 Notation and background 201
7.3 Preliminaries 204
7.4 The number of children in $\mathcal{N}$ of paths in $\mathcal{M}$ 205
7.5 How $\mathcal{M}$ and $\mathcal{C}$ determine $\tilde{\theta}$ 207
7.6 Main results 208
7.7 Some remarks 209
7.8 Applications 210
7.9 Discussion 213
Appendix: Some examples 214
References 215

Summary 217

Samenvatting 223

Curriculum Vitae 229

Index 231

Chapter 1

Introduction

Modulation codes, the topic of this thesis, are one of the elements employed in a digital communication system. We are all familiar with these highly efficient and reliable means to transport information in time or in place, in the shape of, e.g., a CD-player, a computer, or possibly a modem or a fax. In this chapter, we are interested in the principles that makes them work.

In Section 1.1 we position modulation and modulation codes within a digital communication scheme. We first present the usual sender-channel-receiver model for such systems and indicate the function of the other elements composing it: analogue-to-digital conversion, data compression and error-protection.Here, we will see that modulation codes function nearest to the channel and are designed to ensure that the data-stream attains certain properties that are beneficial during subsequent transmission or storage.

In separate subsections, we present a mathematician's view on the other elements of the system while keeping a firm eye on the practical significance of the discussion. In Section 1.2 we provide the necessary background on modulation codes, as detailed as needed to understand and appreciate the other parts of this thesis. This chapter ends with Section 1.3, where we present an overview of the thesis.

We have tried to keep the presentation in this introductory chapter as simple as possible. The only strict prerequisites are a working knowledge of elementary calculus. At a few places, some familiarity with vectors and matrices is also required. The material in Subsection 1.2.4, by far the most difficult part of this introduction, is important but need only be skimmed to understand most of what comes after it.

The subsequent six chapters consist of work that has previously appeared (or will appear, in the case of Chapter 7) in international journals. We have used

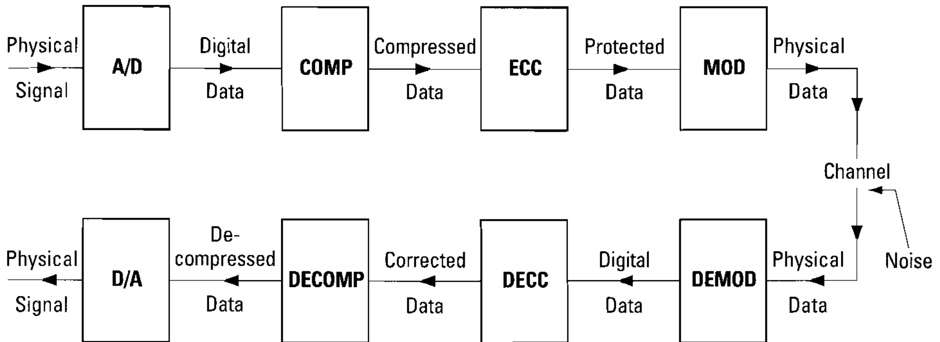

Figure 1: *Schematical form of a digital communication system.*

the occasion to correct some minor mistakes in these papers, and we have also added Appendix C to Chapter 4 and Appendix B to Chapter 5 in order to clarify certain issues. We thank the IEEE organization for its permission to reproduce this material here. For a detailed overview of these chapters we refer to Section 1.3.

At the end of the thesis we provide an index for Chapter 1; this may be useful, e.g., to obtain information about concepts that the reader is assumed to be familiar with in other chapters.

1.1 Digital communication

Present-day digital transmission or storage systems can be represented schematically as in Figure 1. Such a *digital communication system* is usually considered as consisting of three parts, the *sender*, the *receiver*, and the *channel*, the communication pathway along which data can be transported, in some physical form, from the sender to the receiver. Here, the sender may be separated from the receiver either in place (transmission) or in time (storage). Due to the physical nature of the channel, the transmitted signal that carries the information will be corrupted by physical "noise" when it reaches the receiver. (Think of the effects of lightning during transmission, or damaged areas of a disk.) The task of the system is to enable reliable communication even when this takes place over such a *noisy channel*.

At the sender's side, the analogue, physical input data (for example, the electrical current obtained when sound is captured by a microphone) is first passed through an *A/D converter* where it is translated into digital form. Here within each *time-slot*, an interval of time of some fixed duration, the continuously varying physical data is approximated with the aid of a finite number of *bits*, quantums of information each capable of holding either a "zero" or a "one". Then this *dig-*

ital source data, the stream of bits emitted by the A/D converter, is *compressed*, an operation where we try to represent the same (or almost the same) information with fewer bits. The resulting bit-stream is then *protected* against errors by means of an error-correcting code or ECC. Here, some *redundancy* is introduced, that is, (redundant) bits are added to the bit-stream; as a result the original data can still be recovered even if some *errors* occur, i.e., if some bit-values are changed. Finally, the bit-stream is *modulated*, prepared for "transmission over the channel" in some physical form. In the case of transmission, the transmitted signal may consist, e.g., of a sequence of constant-amplitude pulses of either A or $-A$ units in amplitude and T seconds in duration, modulated by some carrier wave; in the case of storage, the sequence of bits may be represented, e.g., by a sequence of pits and lands (CD-system for optical recording) or by successive regions of positive or negative magnetization (magnetic recording).

For various technical and physical reasons, some signals have a much lower probability than others to pass relatively unharmed through the channel. Therefore, it can be beneficial to avoid transmission of bit-streams that result in such signals. This can be achieved by employing a *modulation code* to transform the bit-stream into a *channel-bit-stream* that is more suitable for transmission over the channel. As was the case for error-correcting codes, this adaptation to the channel necessarily leads to an increase of the number of bits in the bit-stream.

At the *receiver*'s side, the whole chain of events is reversed. First, the transmitted channel bits are recovered as well as possible from their physical form, a process called *demodulation*. At this stage, some of the channel bits will have an incorrect value, that is, they may contain *errors*. Some systems also associate *reliability* information with each recovered bit. Such information can be beneficially employed in the subsequent data processing. Next, the resulting channel-bit-stream is transformed back through the modulation code. Then the error-correcting code is used to correct errors as much as possible, the data is *decompressed*, and restored to something (hopefully) resembling its original physical form by the D/A-converter.

We will now take some time to discuss these various parts of the system in some detail. All of them are (at least in part) based on some mathematical ideas; in our presentation we will concentrate on the mathematics that make them work.

1.1.1 A/D conversion

We will model the physical data at the sender's input as a real-valued function $F(t)$ of the time t. For example, music can be described in physical terms as variations in air-pressure. When captured by a microphone, these variations are translated into variations in electrical current. This function F, the *signal*, can be thought

of as a superposition of vibrations, each with a fixed frequency and time-varying amplitude. (The composition of a signal can be investigated with a mathematical technique called *Fourier analysis*.) These signals are, or can be considered to be, *band-limited*, that is, composed of vibrations within a finite range of frequencies. (For example, our ears are insensitive to frequencies outside the range of 20Hz–20.000Hz.)

The digitization of such a band-limited signal starts with an operation called *sampling*: at uniformly spaced time-instances nT_s, $n = 0, 1, \ldots$, the signal value is probed and only the values $F(nT_s)$, $n = 0, 1, \ldots$, are retained. Surprisingly, if T_s is chosen small enough, then no information is lost in this process. Indeed, if $f_{\max}$ is the maximum frequency that occurs in the signal F, then F can be reconstructed *exactly* from the samples $F(nT_s)$ provided that T_s is not larger than $1/(2f_{\max})$. This statement is called the *Nyquist-Shannon sampling theorem* [44], [49].

To complete the digitization, the sample values are *quantized*, rounded off and represented by a *fixed* number of bits. Here we lose some information about the signal; the signal as reconstructed from the quantized samples will differ from the actual signal. The difference, called the *quantization noise*, of course depends on the number of bits available to represent each sample: the more bits, the smaller this noise will be. As an example, in the (*stereo*) CD-system, each of the two signals is sampled at a rate of $1/T_s = 44100$ times per second, and each of the two samples is then quantized to 16 bits.

1.1.2 Data-compression

This part of the system (which is not present in all digital systems) tries to reduce the amount of bits in the bit-stream by a process called *source coding*. If a *source* (in our case the A/D-convertor) emits a data stream that contains *redundancy*, then the data stream can be compressed by removing this redundancy. The concept of redundancy is best illustrated by an example. Evidently, an English sentence can be un ersto d eve if so e let ers are mi si g, which signifies that such sentences contain redundancy. To start with, some letters occur more often than others. The well-known Morse alphabet, which represent one of the first examples of the practical use of source coding, profits from this fact by assigning a short code (e.g., e$\rightarrow$ $\cdot$) to frequently occurring letters and longer codes (e.g., q$\rightarrow$ $- - \cdot -$) to less frequent letters. Next, also some pairs of letters occur more often than others. So, it might even be more profitable to encode *pairs* of letters.

The amount of information generated by a source is called its *entropy*; it is measured in bits (binary digits) per source symbol. These notions were introduced by Shannon in his classical paper [48]. As an example he investigated the entropy

of (a source emitting sentences in) the English language and found an estimate of about 1.3 bits per symbol, where the symbols are the 26 letters in the alphabet together with the "space". That is, an English text can be represented with an average of about 1.3 bits per symbol; in other words, the average symbol in an English sentence contains about 1.3 bits of information.

A particular way of representing sequences of source symbols emitted by a source (such as the Morse alphabet mentioned above) is called a *source code*. The entropy of a source tells us the maximum possible *compression rate*, the maximum number of source symbols that can be represented by an average encoded bit of a source code for that source.

As a simple example, consider a source that emits four possible symbols "a", "b", "c", and "d", where at each instant the probability that a particular one of these symbols is emitted next is 1/2, 1/4, 1/8, and 1/8, respectively. We could of course represent the four symbols by two bits each, as "00", "01", "10", and "11", respectively. Since each emitted symbol is either an "a" or not, with both events equally probable, it seems reasonable to suppose that the symbol "a" represents one bit of information. Similarly, each event out of four possibilities can be represented with two bits. If each of these events occurs with equal probability, it seems that each of them represents two bits of information. So we should think of the symbol "b" as representing two bits of information. A similar reasoning leads to the conclusion that each of the letters "c" and "d" represent three bits of information. By a simple calculation we now find that the entropy, the average amount of information per symbol, equals 1.75 bits per symbol. According to this calculation, it should be possible to represent a stream of letters emitted by this source by an average of only 1.75 bits per symbol, instead of with two bits as proposed earlier. This is indeed possible: encode the letters as

$$a \longrightarrow 0, \qquad b \longrightarrow 10, \qquad c \longrightarrow 110, \qquad d \longrightarrow 111.$$

Now each symbol is represented with the "correct" number of bits. Moreover, it is not difficult to see that a stream of bits resulting from the encoding of a sequence of letters can always be broken up into a sequence of encodings of symbols in one and *only one* way (essentially, this is because the four "codewords" 0, 10, 110, and 111 have the property that no one is the beginning of another one, that is, they form a *prefix code*). Therefore, the original sequence can be exactly recovered from the encoded sequence. As a consequence, distinct sequences have distinct encodings, which is of course a requirement for any alternative representation of source sequences.

So far, we have considered *lossless source coding*, that is, we required that the original source output can be recovered *exactly* from its encoding. For some applications, this requirement is too strict. For example, consider the case where the

source emits video data (digitized pictures). Here, a slight degradation in picture-quality may well be acceptable. In such a case, it is sufficient that a sufficiently close approximation of the original can be recovered. If the encoding has this property, then we speak of *lossy source coding*. The mathematical framework for this case is called *rate-distortion theory*. This theory tells us, under certain assumptions about the source and the *distortion measure*, what is the highest possible rate at which such a source may be encoded, given any maximum on the acceptable distortion. Unfortunately, practical sources tend not to satisfy these assumptions, but nevertheless a result as this can still give strong indications about what rates should be possible for such sources.

1.1.3 Protection against errors; error-correcting codes

When information in the form of a sequence of bits of a given length is sent over a noisy channel, then upon reception certain of the bits will have attained the wrong value; the sequence will contain *errors*. This raises the problem of how to enable reliable transmission over such a channel.

The problem can be understood as follows. If two transmitted sequences are almost equal, then the channel can easily transform one into the other; in other words, the receiver cannot very well distinguish between corrupted versions of such sequences. Therefore, we need to ensure that different transmitted sequences are *very* different, or, as we say, we need to to create a sufficient amount of *distance* between transmitted sequences. A common method to do so is by the insertion into the bit-stream of *parity-check-bits*, bits that hold the parity of the number of "ones" among bits in certain given positions. An *error-correcting code* consists of a collection of rules how to generate these parity-check-bits and where to insert them, the *encoding rules*, together with the procedure to be used at the receiver's side to handle the received sequence, i.e., how to detect and/or correct the errors, the *decoding rules*.

The fraction R of original bits or *information bits* in the resulting sequence is called the *rate* of the code; the quantity $1 - R$ is termed its *redundancy*. So the fewer bits are added, the higher the rate. The redundancy of a code reflects the price that has to be paid for its error-protecting capabilities.

We can think of the *decoding* of a given received sequence as looking among all possibly transmitted sequences for the one that has the highest probability of having been sent (maximum-likelihood decoding). These probabilities of course depend on how the channel acts on the transmitted sequences.

In digital communication, an often-used channel-model is that of a *binary symmetric memoryless channel (BSMC)*: here we assume that each separate bit has a fixed probability $p < 1/2$ of being in error. In this model, the probability

that a certain sequence was sent, given the received sequence, only depends on the *Hamming distance* of this sequence to the received sequence, where the distance is measured by the number of bit-positions in which these sequences differ. (Having a few errors is more probable than having a great many of them.)

The number $C = C(p) = -p \log_2 p - (1-p) \log_2(1-p)$ is called the *capacity* of the channel. (Unless specified otherwise, all logarithms will be to the base 2.) One of the highlights of information theory, the mathematical theory of digital communication, is a theorem obtained by Shannon in his landmark paper [48] which states that reliable communication over a BSMC is possible at any coding rate $R < C$. (A more precise statement is that for any given rate $R < C$ and any arbitrarily small number $\epsilon > 0$, there exist codes of rate R with the property that the word-error-probability after decoding is at most equal to ϵ.) Remark that the way to achieve this is *not* simply to repeat each information bit a number of times!

The above theoretical result only holds "on the long run", that is when the number of transmitted bits gets larger and larger. (Obviously not much can be achieved if we only wish to transmit, e.g., a single bit.) Fortunately the result still approximately holds when the number of bits is very large but fixed. A more serious problem is that no one knows how to construct such codes! (Facts such as this, together with the almost religious admiration for Shannon amongst many theoreticians, may explain why some of the more practice-oriented people do not have a high regard of information theory as a practical science.) Indeed, although in fact Shannon showed that "almost every" code at the right rate has the desired property, surprisingly all known *constructed* codes are much worse than what according to theory should be achievable. (Recently, some progress in this direction has occurred with the invention of a class of codes called *turbo codes* [6]; for an recent overview, see [5].) Although no codes fulfilling the promises of Shannon's theorem are known, many families of codes have been constructed that enable reliable communication at a reasonable rate. Among the codes that have found practical use, we may distinguish two types: *convolutional codes* and *block-codes*.

Block-codes operate on sequences of a fixed length k, and transform each such sequence into a *codeword* of a fixed length n by adding $n-k$ parity-check-bits. (So the rate of the code equals k/n.) When such codes are used to transmit long sequences, the sequence is first grouped into words of length k and is then transformed by the code into a sequence of n-bit codewords.

An important parameter of a block-code is its *minimum distance*, the minimum Hamming distance between any two different code sequences. A code with minimum distance d is capable of correcting $e = \lfloor (d-1)/2 \rfloor$ errors per codeword. That is, assuming that a received word contains *at most e* errors, it is possible to recover the original codeword. To see this, we first note that any received word that contains fewer than $d/2$ errors has Hamming distance less than $d/2$ to the

originally transmitted codeword, but all other codewords, being at Hamming distance at least d to this codeword, have Hamming distance at least $d/2$ to this word. (Think of such received words as being contained in a "sphere" with radius $e < d/2$ centered around the originally transmitted codeword. As the centers of all such spheres are at Hamming distance at least d, these spheres are all disjoint and any word can be contained in at most one of them.) Hence the decoding rule "decode a received word into the (unique) codeword at Hamming distance at most e if such a codeword exists, and give up otherwise" does what is required. This decoding procedure is called *bounded-distance decoding*. A code that can correct e errors when using bounded-distance decoding is called an *e-error-correcting code*.

Bounded-distance decoding can only be practical if a received word can be decoded without needing to inspect all codewords to see which of them is closest to the received word. For example, a code with rate $3/4$ and codeword length $n = 256$ (so with $k = 192$) contains $2^{192} \approx 10^{58}$ codewords. Obviously, in practical applications inspection of even a fraction of these codewords is impossible. Fortunately, many good codes have by design a rich algebraic structure that makes it possible to implement bounded-distance decoding in a much simpler way; instead of by inspection of all codewords a received word can be decoded by *computation* of the closest codeword by means of algebraic operations.

The correction power of an e-error-correcting code with word-length n can be measured by the fraction e/n of correctable errors per codeword. Unfortunately, for a given rate and given value of e/n, the decoding complexity of these codes becomes prohibitively large for large codeword lengths n. A method, often used in practical applications, to obtain good, long codes that can still be efficiently decoded from shorter such codes is a construct called *product codes*. We may think of a codeword in a product code as an array in which each row is a codeword from some given code called the *row code* and each column is a codeword from another given code called the *column code*. (Mostly we assume that both the row- and column code can be efficiently decoded by bounded-distance decoding.) Product-codes are examples of a class of codes called *cooperating codes* [52].

A product code can be efficiently decoded by a procedure where first each of the columns of a codeword is decoded using the decoding procedure for the column code, and subsequently each row is decoded using the decoding procedure for the row code. (This is a typical example of an engineering principle called "divide and conquer".) Here the row-code may be considered as providing an additional check on the bits of consecutive column codewords.

Although product codes have a (much) smaller minimum distance than codes of the same length obtained by other means, such codes when used in combination with the above efficient decoding procedure perform very well with respect

to the only criterion that really matters in practical applications, namely the bit-error-probability after decoding. Indeed, with respect to practical applications, the importance of the minimum distance of a code is often over-estimated. We illustrate this point with the following, admittedly somewhat extreme example. Consider a code obtained from a code with minimum distance d by replacing one codeword of this code by a word at distance one to some other codeword. The resulting code has minimum distance one, but if we would use this code in practice then we would note no appreciable difference in performance in comparison with the original code, simply since the probability of transmitting one of the two words that are close together is neglectably small. (Also, bounded-distance decoding does not achieve channel-capacity, see [56] or [12].)

Convolutional codes are codes operating on potentially infinite bit-sequences. The main difference with block-codes of concern here is in the way that such codes are decoded. An e-error-correcting block-code typically is decoded by bounded-distance decoding. As a consequence, if more than e errors are present the decoder gives up or, much worse, finds a closest codeword different from the codeword originally sent; note that such a *miscorrection* will always lead to the introduction of (many) *additional* errors. So in that case decoding does not help, but instead makes things even worse. In contrast, a convolutional code is suitable for decoding by a procedure that approximates maximum-likelihood decoding and (therefore) does not suffer from such bad behaviour. The *Viterbi decoder* which implements this decoding procedure will produce an encoded sequence in which the bit-error-probability will certainly be lower than that of the channel (assuming a "reasonable" choice for the convolutional code has been made). Therefore, in practical applications, convolutional codes can be profitably used even for a bad channel, i.e., when the bit-error-probability p of the channel is relatively large, while block-codes are more suitable for use when the channel is good, typically for values of p up to 10^{-3}.

The reason for this particular upper bound on p can be understood as follows. Fix the encoding rate R. When an e-error-correcting block-code with codeword length n and rate R is used on a BSMC with error-probability p, then a received codeword will contain on average pn errors. Therefore such a code can be profitably used only when pn is (much) smaller than e, that is, when p is (much) smaller than e/n. Even when the number e/n can be made larger than p, this may require the use of codes for which both e and the codeword length n are very large, which in turn requires the use of decoders of a large complexity. When we now consider the available codes for practical values of the code rate and the codeword length we are lead to roughly the above bound on p. (The above reasoning at least partly explains the interest among engineers in long good block-codes such as algebraic-geometry codes for which decoding procedures of reasonable

complexity are now being developed.)

Often, on a bad channel, a *combination* of a convolutional- and block-code is used. Here a convolutional code is used closest to the channel to somewhat improve the bit-error-probability. The combination of the convolutional code and the channel may then be considered as a new, good channel (sometimes called the em super-channel) for which a block-code can profitably be be used. Here we see another example of cooperating codes.

In practice, most channels do not behave like a BSMC. Due to various physical causes such as lightning during transmission or damaged areas on disk, practical channels suffer from temporary degradations and therefore errors tend to be clustered together in *bursts*, where a burst is a sequence of consecutive unreliable bits, i.e., bits that are in error with high probability. The number of consecutive unreliable bits is called the *length* of the burst. We refer to a channel where a mixture of "incidental" errors (such as on the BSMC) and bursts occur as a *bursty channel.* To combat burst, basically three type of measures are taken, all of which aim to make the practical, bursty channel look more like a BSMC.

The first type of measure is to use *symbol-error-correcting codes.* Codewords of such a code are best considered as being composed not of bits but of *symbols*, where each symbol consists of a fixed number of consecutive bits. The number m of bits in a symbol is called the symbol-size of the code. These codes are designed for the correction of a certain number e of erroneous symbols or *symbol-errors* per codeword; here it makes no difference whether a *single* bit or *every* bit within a symbol is in error.

A symbol-error-correcting code has the disadvantage that a single bit-error now destroys an entire symbol of the code. However, on a bursty channel this disadvantage is outweighed by the advantage that now a burst of length b only affects approximately b/m symbols. A well-known example of symbol-error-correcting codes are the *Reed-Solomon (RS) codes.* These codes have found widespread use in digital communication systems. For example, RS codes based on 8-bit symbols are employed in every CD-system.

The second type of measure to combat burst is to apply *interleaving.* Here we aim at "smearing out" the bits in a burst over several codewords. Basically, instead of placing codewords one after the other, we now interweave groups of codewords before transmission.

The third type of measure is to use product codes (or product-like codes) in combination with an arrangement where the columns of a product-codeword appear as consecutive words in the bit-stream. In such an arrangement a burst will affect relatively few column-codewords, and will therefore cause only a few errors in row-codewords checking these columns. In fact, many interleaving methods may be considered as product codes without row-checks (i.e., with a row code of

rate 1).

Altogether, the use of interleaving in combination with product codes based on RS codes have lead to practical, highly performant error-correcting codes which, for example, enable a CD-player to play high-quality sound from a disk even when a small sector of the disk is literally cut out. For further literature on error-correcting codes, we refer to the textbooks [40] and [39].

1.2 Modulation codes

1.2.1 Type of constraints; constrained systems

Modulation codes are employed to transform or *encode* arbitrary (binary) sequences into sequences that possess certain "desirable" properties. Note the difference with error-correcting codes introduced earlier: an error-correcting code is used to ensure that *pairs* of encoded sequences have certain properties (namely being "very different"), while a modulation code serves to ensure certain properties of *each individual* encoded sequence.

Which properties are desirable strongly depends on the particular storage- or communication system for which the code is designed. For example, in most digital magnetic or optical recording systems we want only to store sequences that contain neither very short nor very long "runs" of successive zeroes or ones. The technical reasons behind this requirement have to do with the way in which a stored sequence is read back from the storage medium (disk or tape) and can be explained briefly as follows. (A more detailed explanation can be found, e.g., in [42] or in [30].)

Digital recording systems mostly use *saturation recording*, where the storage medium is magnetized in one of two opposite directions, one corresponding to a "zero" and the other to a "one". Each bit written on tape has a fixed length (typically in the order of a few tens of μm in magnetic recording), which, due to the constant speed of the reading head over the tape, translates to a fixed duration in time.

The task of the reading head is to locate the *transitions*, the changes of the direction of magnetization that correspond to a transition ($0 \rightarrow 1$ or $1 \rightarrow 0$) in the bit sequence. Commonly these transitions show up as a (positive or negative) *peak* in the output signal of the reading head ("peak-detection"). This output signal, or read signal as it is mostly called, is sampled at discrete time instants to determine the presence or absense of a transition. In order to generate the proper sample moments, the device has to possess an internal "clock" that is matched to the length of the bits. This clock is usually generated by a device called a phase-locked loop (PLL). The PLL is driven by the read signal, and is adjusted, "corrected", at each

occurrence of a transition. By a long absence of a new transition the clock may become too inaccurate ("clock-drift"), and may thus cause erroneous detection of the transitions and/or a wrong count of the number of bits between successive transitions. Therefore, we must avoid sequences containing long runs.

In reality, the output signal is not restricted to a single peak at the corresponding sampling moment, but also shows up in the form of minor peaks at neighbouring time instances. Therefore, if successive transitions are not sufficiently spaced apart, the corresponding read signals may interfere with each other and may cause a transition being missed or wrongly located, an effect called *intersymbol interference* (ISI). To avoid this problem, we need to ensure that transitions in the bit sequence are sufficiently far apart, that is, we must also avoid sequences containing short runs.

Constraints on the sequence to be written of the type discussed above are referred to as *run-length constraints*. Traditionally, the precise constraints are described in terms of two parameters d and k. A *(d, k)-RLL sequence* is a binary sequence where the number of successive zeroes or ones is constrained to be between $d + 1$ and $k + 1$. We often prefer to think directly in terms of the sequence of transitions and non-transitions in a (d, k)-RLL sequence. This sequence can easily be seen to be a *(d, k)-constrained sequence*, a sequence in which each two successive symbols "1" are separated by at least d and at most k symbols "0". (Conversely, the two possible sequences for which a given (d, k)-constrained sequence is the sequence of transitions and non-transitions are both (d, k)-RLL sequences.)

Some digital magnetic recording systems utilize (d, k)-constrained sequences in combination with *equalization* to further counteract intersymbol interference. Usually, equalization is performed at the read side in order to improve the signal shape at the detector. The reading head typically does not respond well to low- and high-frequency signals, and many equalization methods boost up the resulting high-frequency noise on the read signal. It can therefore be benificial to transfer some of the offending part of the equalization to the write side, which is the idea behind a method referred to as *write equalization* [50, 51]. In a typical embodyment of this method, the two-valued, constrained signal at the write side is first passed through a digital recursive filter before being written on tape or disk. Of course, this is only feasible if the output signal of the filter is again two-valued. The question which filters have this property in the case that the input signal is known to obey a (d, k) constraint is answered in [22] (see also [55]).

A run-length constraint forms an example of a constraint that is specified by the absence in admissible sequences of a finite number of "forbidden patterns". (The patterns that are forbidden to occur in admissible sequences are usually referred to as "forbidden subwords".) For example, the (d, k)-constraint can be

specified in terms of the forbidden subwords

$$11, 101, \ldots, 10^{d-1}1, 0^{k+1}.$$

(Here 0^r is the usual notation for a string of r successive symbols "0".) Such a constraint is commonly referred to as a *constraint of finite type*. Many constraints that occur in practice are of this kind.

The constraints mentioned up to now have a natural formulation directly in terms of the bits that make up the sequence. These are sometimes referred to as *time-domain constraints*. Another type of constraints are the *frequency domain constraints*, where restrictions are enforced on the energy content per time unit of the sequence at certain frequencies, that is, on the *power spectral density function* of the sequence (see, e.g., [30]). Encoding for such constraints can be thought of as *spectral shaping*. Most of these constraints belong to the family of *spectral null constraints*, where the power density function of the sequence must have a zero of a certain order at certain specific frequencies. Among these, the constraint that specifies a zero at DC, the zero frequency, has significant practical applications. For example, this constraint is commonly employed in magnetic recording since common magnetic recorders do not respond to low-frequency signals [30]. Sequences that satisfy this constraint are often referred to as *DC-free* or *DC-balanced* sequences.

In our discussion of such spectral constraints, we will follow the literature and represent the encoded bits by 1 and -1. A sequence $x_1, x_2, \ldots$, where each x_i takes a value ± 1, is called *DC-free* if its *running digital sum (RDS)*

$$\mathrm{RDS}_t = x_1 + \cdots x_t$$

takes on only finitely many different values. In that case, the power spectral density function vanishes at the zero frequency (DC).

Of practical relevance is the *notch width*, the width of the region around the zero frequency where the spectral density is low. A good measure of this width [30] is the *sum-variance* of the sequence, the average value of the squared running digital sums RDS_t^2: the lower the sum-variance, the wider the notch.

One usual way to ensure a low sum-variance is to constrain the code sequences to be *N-balanced*: we only allow sequences whose RDS take on values between $-N$ and N, for some fixed number N. Note that such a constraint cannot be specified in terms of a finite collection of forbidden subwords, that is, it is not of finite type.

As we have seen above, for technical reasons we may wish to put various constraints on the sequences that are to be stored or sent over the channel. Sequences that satisfy our requirements are called *constrained sequences* and the

collection of all constrained sequences is called the *constrained system.* A *modulation code* for a given constrained system consists of an *encoder* to translate arbitrary sequences into constrained sequences, and a *decoder* to retrieve the original sequence from the encoded sequence. The bits making up the original sequence and the encoded sequence are usually referred to as *source bits* and *channel bits*, respectively. The collection of all possible outputs of the encoder, i.e., the collection of all *code sequences*, is called the *code system* of the modulation code.

1.2.2 Coding rate and capacity

To achieve encoding, there is a price that has to be paid. Indeed, since there are, obviously, more arbitrary sequences of a given length than there are constrained sequences of the same length, the encoding process will necessarily lead to an *increase* of the number of bits in the channel-bit-stream. This increase is measured by a number called the *rate* of the code. If, on the average, p source bits are translated into q channel bits, then the rate R of the code is $R = p/q$. The quantity $1 - R$ is called the *redundancy* of the code.

All other things equal, we would of course like our code to have the highest possible rate (or, equivalently, the lowest possible redundancy). However, it turns out that for all practical constraints there is a natural barrier for the code rate, called the *capacity* of the constraint (or of the corresponding constrained system), beyond which no encoding is possible. This discovery by Shannon [48] has been of great theoretical and practical importance. Indeed, once we know the capacity C of a given constrained system, then, on the one hand, we know that the best encoding rate that could possibly be achieved is bounded from above by C, and, on the other hand, once we have actually constructed a code with an encoding rate R, the number R/C, called the *efficiency* of the code, serves as a benchmark for our engineering achievement.

We will now try to explain the existence of this natural limit to the achievable rate. At first, we will do so by means of a relatively simple example. To this end, we consider the constrained system consisting of all sequences that do not contain two consecutive symbols "0", that is, we consider *(0, 1)-constrained sequences.* For reasons that will become clear later on, we will refer to this constrained system as the *Fibonacci system.*

It turns out that the number N_n of those sequences of length n, and in particular their *growth rate* for large n, is of crucial importance. Let us now explain why this is so. To that end, suppose that we can encode at a rate R. By the definition of the rate, this means that, in the long run (i.e., for large n), approximately Rn source bits are translated into n channel bits. There are 2^{Rn} distinct source sequences of length Rn, all of which need to be translated into distinct constrained sequences

of length n, of which there are only N_n. Therefore, we necessarily have that $2^{Rn} \leq N_n$, or, equivalently, that $R \leq n^{-1} \log N_n$.

For small lengths n, the number N_n is easily found by listing all admissible sequences of these lengths. For example, we have that $N_1 = 2$ (both "0" and "1" are admissible), $N_2 = 3$ ("01", "10", and "11"), and $N_3 = 5$ ("010", "011", "101", "110", and "111"). Furthermore, we have that

$$N_n = N_{n-1} + N_{n-2}, \tag{1}$$

for all $n \geq 3$. This can be understood as follows. Let $N_n(0)$ and $N_n(1)$ denote the number of admissible sequences of length n beginning with a "0" or a "1", respectively. Obviously, $N_n = N_n(0) + N_n(1)$. If an admissible sequence begins with a "1", then it consists of a "1" followed by some admissible sequence of length $n-1$. Since, conversely, any sequence of length n obtained from an admissible sequence of length $n - 1$ by putting a "1" in front of it is admissible, we conclude that in fact $N_n(1) = N_{n-1}$. Similarly, if an admissible sequence begins with a "0", then the first two symbols must be "01" (since "00" is forbidden), hence it consists of the word "01" followed by an admissible sequence of length $n - 2$. Again, any sequence thus obtained is admissible, and we conclude that $N_n(0) = N_{n-2}$. Summarizing, we find that $N_n = N_n(1) + N_n(0) = N_{n-1} + N_{n-2}$.

The recurrence relation (1) for the numbers N_n, which identify them as the *Fibonacci numbers*, allows us to compute as many of them as we like, and, moreover, enables us to determine their growth rate. Indeed, the mathematical theory of such recurrence relations states that the solution exhibits *exponential* growth. In our particular case, this means that N_n can be approximated as λ^n, for some real number λ. If true, then (1) suggests that $\lambda^n = \lambda^{n-1} + \lambda^{n-2}$ holds for all n, which is the case if and only if

$$\lambda^2 = \lambda + 1. \tag{2}$$

This quadratic equation for λ has the two solutions ϕ and θ, where

$$\phi = (1 + \sqrt{5})/2, \qquad \theta = (1 - \sqrt{5})/2. \tag{3}$$

Observe that since ϕ and θ both satisfy the equation (2), any sequence of numbers N_n with

$$N_n = a\phi^n + b\theta^n,$$

for some fixed numbers a and b, indeed satisfies the recurrence relation (1). Then a proper choice for the numbers a and b can ensure that both N_1 and N_2 have the required value. In the case at hand, it can be shown that

$$N_n = \phi^{n-1} + \theta^{n-1}.$$

Since $\phi > |\theta|$, we have that $N_n \approx \phi^{n-1}$, and hence that

$$\lim_{n\to\infty} \frac{\log N_n}{n} = \log \phi. \tag{4}$$

(Here the function "log" refers to the logarithm with base 2.)

This last expression enables us to establish an upper bound for the coding rate of our constraint. Indeed, from (4), it follows that

$$R \leq C,$$

where

$$C = \log \phi = 0.6942\cdots$$

is the capacity of our constraint.

1.2.3 The capacity of constraints of finite type

Consider a constrained system $\mathcal{L}$. As in the example above, we let N_n denote the number of constrained sequences of length n. We now *define* the capacity $C(\mathcal{L})$ of the constrained system $\mathcal{L}$ as

$$C(\mathcal{L}) = \lim_{n\to\infty} \frac{\log N_n}{n}, \tag{5}$$

provided that this limit exists. Intuitively, this quantity represents the *average amount of information (in bits) carried by a bit of a constrained sequence*. Now a similar reasoning as in the above example shows that encoding at a rate R can only be possible if $R \leq C(\mathcal{L})$ and, moreover, it suggests the possibility of encoding at rates arbitrarily close to $C(\mathcal{L})$.

What is less clear is whether this limit exists and how it could be computed. Obviously, if the constrained system consists of some arbitrary collection of sequences, nothing much can be said, so let us restrict our attention to the more structured type of constrained systems as encountered in practical applications. To keep things simple and to help our intuition as much as possible, we will at first only consider constrained systems of finite type. Recall that such a system is described in terms of a finite collection of "forbidden subwords". (For example, the Fibonacci system discussed earlier is specified by the forbidden subword "00".) So let us assume that our constrained system $\mathcal{L}$ is specified by the finite collection $\mathcal{F}$ of forbidden subwords, and suppose that all words (patterns) in $\mathcal{F}$ consist of at most $m + 1$ bits, that is, have *length* at most $m + 1$, for some integer m. (For the Fibonacci system, we have $m = 1$.) By the way, note that

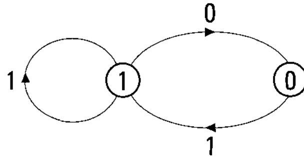

Figure 2: *A presentation of the Fibonacci system.*

we might as well assume that *all* words in $\mathcal{F}$ have length $m + 1$. Indeed, forbidding, e.g., the word 000 is (almost) equivalent to forbidding the four words 00000, 00001, 00010, and 00011. (This makes a difference only at the end of a sequence, but does not change the behaviour of $n^{-1} \log N_n$ for $n \to \infty$.)

We are interested in how a given constrained sequence can be extended to a longer constrained sequence. Since all forbidden subwords have a length at most $m + 1$, to answer this question we need only to know the *last m bits* of our sequence. These last m bits carry all the necessary information and are referred to as the (terminal) *state* of the sequence. If we know the state of our sequence, we know both how it can be extended and the state of the extended sequence that is obtained. Indeed, we could imagine the generation of a long constrained sequence as a process where we move from state to state and generate a new bit by each transition from a state to the next state. This is a crucial image which the reader should keep in mind for what follows.

In fact we have now rediscovered an alternative description of a constrained system of finite type in terms of a (labeled directed) *graph*. Here, a graph $G = (V, A)$ consists of a collection V of vertices or *states*, and a collection A of labeled arcs or *transitions* between states. We will write beg(α) and end(α) to denote the *initial state* and the *terminal state* of the transition α. Some authors refer to a "labeled directed graph" as defined above as a *finite-state transition diagram* or *FSTD*. The constrained system $\mathcal{L}(G)$ *presented* by G consists of all sequences that can be obtained by reading off the labels of a *path* in the graph, or, as we will say, the sequences that are *generated* by paths in G. (Instead of the word "path", some authors use the word *walk*.)

For example, the graph presenting the Fibonacci system is depicted in Figure 2. It has two states, 0 and 1, and three transitions, a transition from state 1 to state 1 with label 1, a transition from state 1 to state 0 with label 0, and a transition from state 0 to state 1 with label 1. (There is no transition from state 0 to state 0, since that would allow the generation of a sequence containing the forbidden word 00.) More general, a (d, k)-constrained system can be presented by the graph in Figure 3. Here, we follow common practice in that the state label indicates the number of trailing zeroes of the corresponding terminal state.

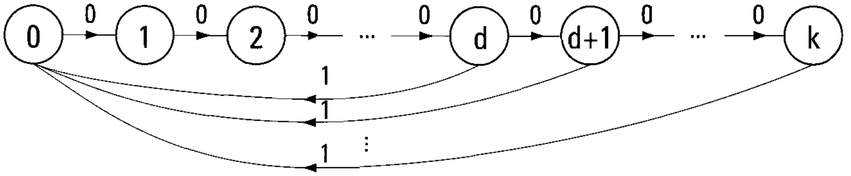

Figure 3: *Presentation of a (d, k)-constrained system.*

Let us now see how such a presentation allows us to determine the number of constrained sequences of a given length. We can consider such a sequence to consist of the first m bits, referred to as the *initial state* of the sequence, followed by the bits generated by the path that has as its states the consecutive m-bit subwords of the sequence. Note also that different paths in the graph from the same initial state generate different sequences. Therefore, if for $n \geq m$ we let $N_n(s)$ denote the number of constrained sequences of length n with initial state s, then for $n \geq m + 1$ these numbers satisfy the recurrence relation

$$N_n(s) = \sum N_{n-1}(\mathrm{end}(\alpha)), \tag{6}$$

where the sum is over all transitions α with $\mathrm{beg}(\alpha) = s$. For example, for the Fibonacci system, we obtain that

$$N_n(0) = N_{n-1}(1)$$

and

$$N_n(1) = N_{n-1}(0) + N_{n-1}(1)$$

for $n \geq 2$, from which the recurrence relation (1) was obtained.

As in the case of the Fibonacci system, the mathematical theory of the type of recurrence relations (6) shows that the number $N_n = \sum_s N_n(s)$ of constrained sequences of length n exhibits exponential growth, but in this case the theory is much more involved. Since the underlying ideas are crucial not only for capacity computations, but also for a good understanding of the field of code construction as a whole, we will nevertheless go somewhat more into details.

It turns out that, at least for the graphs derived from finite-type constraints as explained above, all information concerning the numbers $N_n(s)$ can be obtained from what is called the *adjacency matrix* of the graph. Here, the adjacency matrix D of a graph G is a square array of non-negative numbers, with both rows and columns indexed by the states of G, where the number $D(s, t)$ in the sth row and tth column counts the number of transitions in G from state s to state t. Readers familiar with matrix theory will have no difficulty verifying that then the (s, t)-entry of the nth power D^n of D in fact equals the number of paths of length n

in G from state s to state t. So the sum of the entries in D^n actually counts the number of paths of length n in the graph.

We can use the adjacency matrix to investigate the recurrence relations (6) as follows. Let $\pi_s^{(n)}$ denote the number of paths in G of length n starting in s, or, equivalently, the number of sequences generated by such paths, and let $\pi^{(n)}$ denote the vector with as its entries the numbers $\pi_s^{(n)}$. As we observed earlier, we have that

$$N_n(s) = \pi_s^{(n-m)}, \tag{7}$$

hence from (6) we obtain that

$$\pi^{(n)} = D\pi^{(n-1)}, \tag{8}$$

for all $n \geq 1$. (By the way, it is also easy to see *directly* that this recurrence relation holds.) Note that, as a consequence of (7) and an earlier observation, the number N_n actually grows like the sum of the entries of powers D^n of the adjacency matrix D! Using the relation (8) repeatedly, we can express the vectors $\pi^{(n)}$, $n > m$, in terms of the vector $\pi^{(m)}$ as

$$\pi^{(n)} = D^{n-m}\pi^{(m)}. \tag{9}$$

At this point, we need a result from the classical Perron-Frobenius theory for non-negative matrices (note that our adjacency matrix D is of this type). The said result states that the largest real ("Perron-Frobenius") eigenvalue $\lambda = \lambda_D$ of D has the property that the matrices $(\lambda^{-1}D)^n$ tend to a limit if n tends to infinity, see e.g. [47], [43]. Now divide both sides of the equation (9) by λ^n, let n tend to infinity, and then use this result. If we do so, we find that for $n \to \infty$ the vectors $\lambda^{-n}\pi^{(n)}$ tend to some limit vector π. Hence there are (non-negative) constants π_s, $s \in V$, such that

$$\lim_{n\to\infty} N_n(s)/\lambda^n = \pi_s.$$

(In fact, using (8) it can be shown that the vector π is a *right-eigenvector* of the matrix D, that is, $\lambda\pi = D\pi$.) This result is just what is needed, and implies that the capacity $C(\mathcal{L})$ of the constrained system $\mathcal{L}(G)$ presented by G is given by

$$C(\mathcal{L}) = \lim_{n\to\infty} \frac{\log N_n}{n} = \log \lambda_D.$$

This expression enables us to actually calculate the capacity of any given constrained system of finite type. Indeed, the Perron-Frobenius eigenvalue can be computed, e.g., as the largest real zero of the *characteristic polynomial* $\chi_D(\lambda) = \det(\lambda I - D)$ of the matrix.

For example, in the case of the Fibonacci system discussed earlier, the adjacency matrix D of the graph in Figure 2 that presents this system is

$$D = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}.$$

So the characteristic polynomial of D is

$$\chi_D(\lambda) = \lambda^2 - \lambda - 1,$$

and again we find that the capacity equals $\log \phi$ with ϕ as in (3).

1.2.4 Sofic systems and their capacity

Up to now we have limited the discussion of capacity to constrained systems of finite type. Fortunately, most of what has been said can be extended to a much wider class of constrained systems. The key to the capacity result in the previous section was the presentation of a constraint of finite type by means of a labeled directed graph constructed from the forbidden words, and the correspondence between words in the constrained system and paths in this graph.

This observation motivates the introduction of *sofic systems*, constrained systems that can be presented by some finite labeled directed graph. Indeed, it is possible to compute the capacity of a general sofic system by a method similar to the one discussed for systems of finite type, but there are some complications. Before we explain this method (and the reason for the complications), we discuss sofic systems in more detail.

Sofic systems are of great theoretical and practical importance. It turns out that about every constrained system encountered in practical applications is in fact a sofic system. (This is less remarkable than it seems once we realize that about any digital device that we build is actually a *finite-state device*, whose possible output sequences necessarily constitute some sofic system.)[1]

In the previous section, we have shown that each constrained system of finite type is in fact a sofic system, and we have seen how to obtain such a presentation given the collection of its forbidden subwords. However, as we will see later on, many (in fact "almost all") sofic systems are not of this special type. the *even system*, which consists of all sequences where two successive symbols "1" are separated by an even number of symbols "0", is sofic but not of finite type. (The

[1] The theoretical importance of sofic systems, and the original reason that motivated Weiss [54] to investigate them, is the fact that the class of sofic systems is the smallest class that contains the systems of finite type and is closed under *factor maps*, maps that can be realized as a sliding-block decoder. Sliding-block decoders are discussed in Subsection 1.2.6.

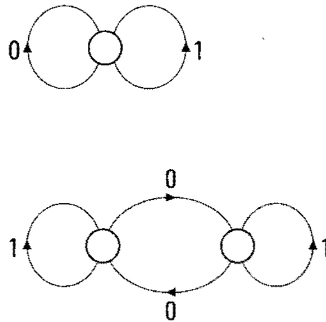

Figure 4: *Two presentations of the full system.*

forbidden subwords are precisely the words of the form $10^{2n+1}1$ with $n \geq 0$.) We leave it to the reader to find a presentation of this sofic system. (Hint: there are two "ending-conditions" of a constrained sequence, namely ending in an even (possibly null) or in an odd number of zeroes.)

It is important to realize that a given sofic system has many different presentations. (Indeed, several code construction methods actually amount to finding a "suitable" presentation.) For example, the *full system* consisting of all binary sequences can be presented by both graphs in Figure 4 (and by infinitely other ones). Fortunately, it can be shown constructively that each sofic system has a unique *minimal deterministic presentation*, called the *Fisher cover* or the *Shannon cover* of the system. That is, among all *deterministic* presentations, presentations where in each state the outgoing transitions from that state carry distinct labels, there is a unique one with the minimal number of states.

The Fisher cover of a sofic system is important: it can be constructed from any presentation of the sofic system; moreover, many properties of a sofic system can be read off directly from its Fisher cover. (We will mention one such property later in this section.)

To explain how the Fisher cover can be obtained, we first have to discuss an important means to reduce the number of states in a presentation, an operation called *state merging*. This works as follows. Suppose that the outgoing transitions in two given states can be paired off in such a way that the two transitions in each of the pairs carry the same label and end in the same state. Then these states have the same *follower set*, where the follower set of a state is the collection of sequences that can be generated by paths leaving this state. So from the point-of-view of sequence generation, these two states accomplish the same, hence they can as well be combined, or *merged* as it is usually called, into a single state. (Note that if the original presentation is deterministic, then the resulting presentation will again be deterministic.)

To construct the Fisher cover from a given presentation, we proceed as follows. First, we transform the presentation into one that is deterministic. This can be achieved, e.g., by using a variant of the well-known *subset construction* for finite automata (see for example [28], [29]), or by *state-splitting* (see Subsection 1.2.8). Then we reduce the number of states in the presentation by state merging. This process of merging states is repeated until no further merging is possible. It can be shown that the resulting presentation, the Fisher cover, does not depend on the order in which the merging was carried out, and that, moreover, no deterministic presentation of the system can have fewer states; moreover, such a presentation must be equal to the Fisher cover if it has the same number of states. (For further details on this construction, we refer to [38].)

Readers familiar with automata theory will recognize the parallel with *regular languages*, the class of languages generated by finite automata. Here also, a given regular language can be generated by many different finite automata among which there is a unique minimal deterministic one. In fact, the resemblance is more than superficial; a sofic system is in fact a regular language, and the construction of the Fisher cover for a sofic system parallels the construction of its minimal finite deterministic automaton. [2]

Which sofic systems are constrained systems of finite type? (We simply refer to such systems as "of finite type" or "finite-type".) To answer this natural question, we first need to introduce a new notion. Let G be a presentation of a sofic system $\mathcal{L}$. This *presentation* is said to be *of finite type* if there are numbers m, the *memory*, and a, the *anticipation*, such that any two paths in G that generate a given sequence in $\mathcal{L}$ are equal, with the possible exception of at most m initial and a terminal transitions. That is, given a (long) sequence in $\mathcal{L}$, we can actually reconstruct the path in G used to generate the sequence, up to a few transitions at the beginning and the end of the path.[3]

For example, the presentation of the Fibonacci system in Figure 2 is of finite type, with memory $m = 1$ and anticipation $a = 0$. Indeed, a "0" is generated by a unique transition, a "1" preceded by a "1" is necessarily generated by the loop, and a "1" preceded by a "0" is necessarily generated by the transition from state 0 to state 1. For another example, in Figure 4, the upper presentation of the full system is of finite type, with memory 0 and anticipation 0. The lower presentation

[2]For more information on the connections between automata theory and symbolic dynamics, the field to which sofic systems belong, we refer to [3].

[3]The labeling map from bi-infinite paths to bi-infinite sequences induced by the labeling of a given presentation is said to be of finite type if the presentation itself is of finite type. In that case, the induced map is easily seen to be one-to-one, and its inverse is a *factor map*, i.e., the induced map can be inverted by means of a sliding-block decoder. This observation plays an important role in various code construction methods that involve transformations of a given finite-type presentation.

is not of finite type: the two loops (both labeled "1") give rise to arbitrarily long, distinct paths that generate all-one sequences.

As shown by the last example, a given sofic system may have presentations of finite type and other presentations that are not of finite type. Nevertheless, the notion of a presentation of finite type is important. Indeed, we will now show that a sofic system is of finite type if and only if it possesses *some* presentation that is of finite type. This can be accomplished as follows.

1) We have already seen that a *sofic system* of finite type has a *presentation* of finite type; indeed, in the presentation that we constructed, the states in the path that generates a sequence are determined by the subwords of length m of the sequence. So this presentation has memory m and anticipation $a = 0$ (i.e., it is deterministic).

2) Conversely, each *presentation* of finite type actually presents a *system* of finite type. In fact, if the presentation has memory m and anticipation a, then the corresponding sofic system can be described in terms of forbidden subwords of length at most $m + a + 2$; the forbidden words are precisely the words of that length that cannot be generated by a path in the graph. (Indeed, for any word $x = x_{-m} \cdots x_a$ in $\mathcal{L}$, let $\alpha = \alpha[x_{-m} \cdots x_a]$ denote the common transition α_0 in all paths $\alpha_{-m} \cdots \alpha_a$ that generate x. The crucial observation is that for any word $x_{-m} \cdots x_{a+1}$ in $\mathcal{L}$, generated by $\alpha_{-m} \cdots \alpha_{a+1}$, say, the terminal state of $\alpha_0 = \alpha[x_{-m} \cdots x_a]$ and the initial state of $\alpha_1 = \alpha[x_{-m+1} \cdots x_{a+1}]$ are equal. Hence, if all subwords of size $m+a+2$ of some bi-infinite sequence $\{x_n\}_{n\in Z}$ are contained in $\mathcal{L}$, then this sequence is generated by the bi-infinite path $\{\alpha[x_{n-m} \cdots x_{n+a}]\}_{n\in Z}$. So whether or not a sequence is contained in $\mathcal{L}$ is determined by its subwords of length at most $m + a + 2$, that is, $\mathcal{L}$ is of finite type.)

So we see that indeed a sofic system is of finite type precisely when some presentation of the constraint is of finite type. At first, this characterization seems not of much use for showing that a given sofic system is *not* of finite type. Moreover, even if the constraint has a presentation of finite type, it could well be next to impossible to find one. Fortunately, this is not the case: it can be shown that a sofic system is of finite type precisely when its Fisher cover is a presentation of finite type (that is, has finite memory; since the Fisher cover is deterministic by definition it has anticipation 0). Indeed, a given presentation of a constraint can easily be transformed into a deterministic presentation for the same constraint by state-splitting (this operation is explained in Subsection 1.2.8); this latter presentation is of finite type if and only if the original presentation is of finite type. Moreover, it is not difficult to see that state-merging, and hence the construction of the Fisher cover from a given deterministic presentation, preserves the property of being of finite type. Then our claim immediately follows from the above result.

Now let us return to the problem of computing the capacity of a general sofic

system, i.e., the evaluation of the limit in (5).[4] Let us recall how this was done for systems of finite type. We first derived a (deterministic) presentation for the system, and this gave us a recurrence relation for the numbers $N_n(s)$ of sequences in the system with initial state s. Then we used the adjacency matrix D of the graph to establish that the sum N_n of these numbers grows like the number of paths of length n n the graph, that is, as the sum of the entries of D^n. Since according to Perron-Frobenius theory this sum grows exponentially in λ_D, the largest real eigenvalue of D, the same holds for N_n, which, according to the definition of the capacity, establishes that the capacity equals $\log \lambda_D$.

We can do the same thing for a general sofic system, using the adjacency matrix of some presentation of the system, but there is a problem. As observed earlier, the entry $D^n(s,t)$ of the nth power D^n of the matrix D actually counts the number of paths of length n from state s to state t in the graph that presents the system, and like before the sum of these entries, that is, the total number of paths of length n in the graph, grows exponentially in λ_D.

However, it may well happen that many of these paths generate the *same* sequence, and if so it may well happen that the number of sequences of length n is significantly smaller than the number of paths of that length, thus turning our computations into "nonsense". (For a most dramatic example, imagine that all labels in the graph are equal to "0". Then the sofic system presented by the graph consists of the sequences 0^n, $n \geq 0$, only, so the capacity is zero, but the Perron-Frobenius eigenvalue of the corresponding adjacency matrix could be anything!)

By the way, note that this problem cannot occur if the presentation is of finite type, but we have seen before that the only sofic systems possessing such a presentation are those of finite type! Fortunately, to avoid this problem it is sufficient that the presentation is *deterministic*. (Note that the presentation for a constraint of finite type as derived earlier is deterministic!) Indeed, in that case all paths of length n that begin in a given state generate different sequences, and this is sufficient to conclude that the exponential growth rate of the number of constrained sequences of length n and the number of paths in the graph of length n are equal.

Summarizing the above, we conclude that the capacity of a sofic system presented by a given graph can be computed as follows. First, we construct a deterministic presentation for the system. This can be done by splitting states until

[4] The fact that this limit exists immediately follows from Feteke's Lemma ([11], see also [46]): if the non-negative numbers a_n are such that $a_{m+n} \leq a_m + a_n$, for all $n, m \geq 0$, then $a^* = \lim_{n\to\infty} a_n/n$ exists and $a^* \leq a_n/n$ for all n. Indeed, if N_n is the number of sequences of length n contained in a *subword-closed* constrained system $\mathcal{L}$, then obviously $N_{m+n} \leq N_m N_n$, hence an application of this Lemma with $a_n = \log N_n$ shows that such constrained systems have a well-defined capacity. Note that sofic systems are subword-closed; in fact, the sofic systems are precisely the subword-closed regular languages.

the resulting presentation is deterministic. We will discuss state-splitting later. (We might even choose to construct the Fisher cover, an attractive choice since it has the minimum possible number of states.) Then we establish the adjacency matrix D of the deterministic graph thus obtained, compute its Perron-Frobenius eigenvalue λ_D, and obtain the capacity C of the system as $C = \log \lambda_D$.

Sofic systems and their bi-infinite counterparts *sofic shifts* belong to the subject of *symbolic dynamics*. For more information on this subject and its relation to coding theory we refer to [38] and [7]. The first of these is devoted to symbolic dynamics and modulation codes, and the second one provides some background on error-correcting codes and also contains a few chapters linking symbolic dynamics to coding theory, automata theory and system theory.

1.2.5 Encoding and encoders

An *encoder* for a given constrained system $\mathcal{L}$ is a device that transforms arbitrary binary sequences of *source bits* into sequences contained in $\mathcal{L}$. Commonly, the encoder is realized as a *synchronous finite-state device*. Such a device takes *source symbols*, groups of p consecutive source bits, as its input, and translates these into q-bit words called *codewords*, where the actual output depends on the input and possibly on the *internal state*, the content of an internal memory, of the device. (Note that since an actual hardware-realization of such a device can only contain a finite amount of memory, the number of internal states is necessarily finite.) The rate of such an encoder is then $R = p/q$. (To stress the individual values of p and q we sometimes speak of a "rate $p \to q$" encoder.) Obviously, each codeword must itself satisfy the given constraint. Moreover, the encoder needs to ensure that the bit-stream made up of successive codewords produced by the encoder also satisfies the constraint.

In the simplest case, the encoder translates each source symbol into a unique corresponding codeword, according to some code table. Of course this will only produce an admissible sequence if the concatenation of any number of these codewords, in any order, also satisfies the given constraint. (Here, the *concatenation* of two words a and b is the word ab consisting of the bits of the first word followed by the bits of the second word.)

For an example, let us consider again the Fibonacci system, the collection of (0, 1)-constrained sequences where the subword "00" is not allowed to occur in the sequence. A possible encoder has $p = 1$ and $q = 2$ (so the rate of this code will be $R = 1/2$), and translates the source symbol "0" into the codeword "10" and the source symbol "1" into "11". It is now easily seen that any succession of the codewords "10" and "11" will indeed never contain the forbidden pattern "00". This simple code is called the *Frequency Modulation (FM)* or *bi-phase* code.

Note that the original source sequence can easily be obtained from the encoded sequence, namely by dropping every other bit from the encoded sequence. So *decoding* is possible. However, note also that, in practice, correct decoding requires some sort of *synchronization* between encoder and decoder. Indeed, practical decoders dispose, at each time instant, of only a small part of the entire stream of channel bits. Consequently, there are essentially two ways to group the available bits into codewords and the decoder needs to know which way to choose. We will discuss synchronization later on.

Earlier we computed the capacity C of this constraint and we found that $C = 0.6942\ldots$. So the *efficiency* R/C of this code is approximately 0.72. So although this code is very simple, its rate is far from optimal. It would be much better to have a code rate of, say, 2/3, for an efficiency of about 0.96.

So let us try to devise, in a similar way, a code with $p = 2$ and $q = 3$. The available codewords of length three are

$$010, \qquad 011, \qquad 101, \qquad 110, \qquad 111,$$

since all other words of length three contain the forbidden pattern "00". There are four source symbols (namely 00, 01, 10, and 11), so we need to pick four code words for our translation table. Unfortunately, whatever four codewords we choose, among them there will always be one ending in a "0" and one beginning with a "0", and then these two cannot be concatenated without violating our constraint. We could use the same idea, but then with *larger* values of p and q for which $p/q = 2/3$. In fact, a moment of reflection reveals that in order to succeed we need a collection of 2^p code words of length q, all of which begin with a "1" (or, equivalently, all of which end with a "1"). The number $N_q(1) = N_{q-1}$ of such codewords can be computed from (1), and if we do so we find that the smallest p and q that work are $p = 12$, $q = 18$. But then our encoder would need a enormous encoding table to memorize all translations, and would therefore require too much hardware.

The previous code construction method can be considered as a simple example of the *principal-state method* of Franaszek. Here, we search for a collection of states, referred to as *principal states*, in a presentation of the constraint, with the property that for each of these principal states there are "sufficiently many" *encoding paths* beginning at this state and ending in another principal state. In our case, the set of principal states consists of the single state "1".

As this example shows, we need a more subtle approach to the design of efficient codes which possess simple encoders. Nevertheless, the idea in this example can be used to show that, for a given sofic system with capacity C, any encoding rate $R = p/q \leq C$ can be achieved, provided that we allow arbitrarily large values of p and q. However, to obtain simple, easy-to-build encoders, we need to

keep p and q small. (Very recently, some progress has been made [34, 35] which shows that, even if p and q are large, systematic encoding may still be possible by using a smart variant of *enumerative encoding* [8]. Nevertheless, this new approach has its own problems and the above statement that p and q should better be small is a reasonable approximation of the truth.)

To do better, we need the encoder to base its encoding decision on more than a single source symbol input alone. For example, consider again the design of a rate-$2/3$ encoder for the Fibonacci system with $p = 2$ and $q = 3$. We will denote the source symbols 00, 01, 10, and 11 by 0, 1, 2, and 3, respectively. Suppose that the encoder tries to use the following simple encoding rule:

$$0 \to 011, \qquad 1 \to 101, \qquad 2 \to 110, \qquad 3 \to 111. \tag{10}$$

This works fine except when the encoder input is a "2" followed by a "0", which produces the word 110.011 in the encoder output. However, this problem can be eliminated provided that we allow the encoder a "second stage" to modify its initial outputs, by replacing, *from left to right*, each occurrence in the output of the words 110.011 and 110.110 by the words 010.101 and 010.111, respectively. (Here, the second substitution is required to avoid problems when the input sequence contains "220".)

Note that decoding of this code is possible: the decoder decodes the codeword 010 into the source symbol "2", and uses the rules (10) to decode the other codewords, except when the previous codeword was 010, in which case 101 and 111 are decoded as "0" and "2", respectively. This encoder can be realized as a finite-state device that encodes with an *encoding delay* of one source symbol. (This delay is needed to "look ahead" to the next source symbol.) The "state" of the encoder memorizes the previous source symbol and whether or not a substitution is in progress.

The above encoding method is sometimes called the *substitution method.* A similar, but more complicated example of the substitution method is the rate-$2/3$ $(1, 7)$-code in [36]. This method can be interpreted as a principal-state method where we allow the encoding paths to have different lengths. In Subsection 1.2.8 we will discuss various other code construction methods. Some of these methods are best understood as constructing *look-ahead encoders* for the constraint, an alternative encoder description that will be discussed at that time.

1.2.6 Decoding and decoders

The use of the word "code" implies that it should be possible to recover or *decode* the original sequence of p-bit source symbols from the corresponding sequence of q-bit codewords. However, in practice a stronger requirement is needed. As can

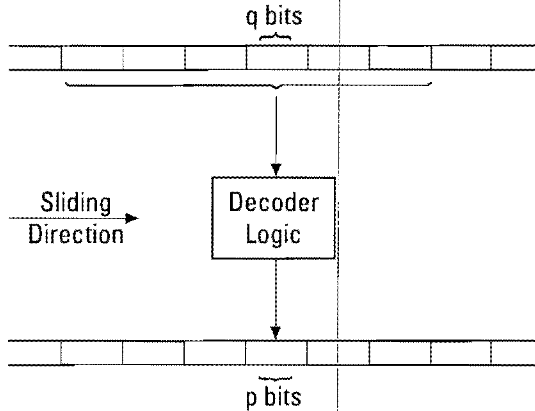

Figure 5: *A sliding-block decoder.*

be seen from Figure 1, the encoded sequence is directly transmitted over the channel, and due to the occurrence of noise on the channel the received sequence may contain errors. Surely, we do not wish that one (or a few) errors in the received sequence could cause the decoder to produce an unlimited number of errors in the decoded sequence! Therefore, we commonly require that the modulation code can be decoded by a *sliding-block decoder*. A sliding-block decoder for a rate $p \to q$ encoder takes (a sequence of) q-bit words y_n as its input, and produces a sequence of p-bit symbols x_n as its output, where each output symbol x_n only depends on a corresponding sequence $y_{n-m}, \ldots, y_{n+a}$ of $w = m + 1 + a$ consecutive inputs, for some fixed numbers m and a, $m \leq a$. We will refer to the number w as the *window size* of the decoder. (The numbers m and a are referred to as the *memory* and *anticipation* of the decoder.) The name "sliding-block decoder" refers to the image of the decoder sliding a "window" of width w over the sequence to be decoded. (Refer to Figure 5.) Note that an error in the encoded sequence will cause at most w symbol errors in the original sequence. So the *error propagation*, the amount of erroneous bits in the decoder output caused by a single input error, is limited to at most wp bits.

The window-size of the decoder is an important parameter of a code: it provides an upper bound on the error propagation of the code, and also gives a good indication of the *size* of the decoder, the amount of hardware necessary to implement the decoder. (That is, it provides an upper bound on the *complexity* of the decoding operation.) Indeed, often the size of the decoder is more important than the size of the encoder; think for example of devices such as a compact disc player which only need to handle previously stored or transmitted data and therefore do not need to have an encoder on board!

Codes with a window-size of one codeword are called *block-decodable*. For such codes, decoding a given codeword or *block* can be achieved without any knowledge of preceding or succeeding codewords. In many present-day applications, modulation codes are used in combination with symbol-error-correcting codes based on Reed-Solomon codes over some finite field GF(2^p). In that situation, the use of a rate $p \to q$ block-decodable modulation code becomes especially attractive, since then a channel-bit error affects at most one p-bit symbol of a Reed-Solomon codeword. Note that this cannot be guaranteed if a non-block-decodable code is used, even if the window-size of the code is smaller than q bits.

We have already met some block-decodable codes: the rather trivial rate $1 \to 2$ FM code and a possible rate $12 \to 18$ code for the Fibonacci system of $(0, 1)$-constrained sequences in subsection 1.2.5. (Note that the arguments given there actually show that any rate-2/3 block-decodable code for this system must have codewords of length $q \geq 18$.) Another well-known example of a block-decodable code is the *Modified Frequency Modulation (MFM)* or *Miller* code. This is a rate $1 \to 2$ $(1, 3)$-code which, like the FM code, encodes by inserting a *merging bit* between each two consecutive source bits. Here, the merging bit is set to "0" except when both surrounding bits are "0", in which case it is set to "1". (For the FM code, the merging bit is always set to "1".) The MFM code is indeed block-decodable: decoding amounts to dropping each second bit, which can be realized as an operation on 2-bit codewords.

The use of merging bits is a well-established technique for constructing block-decodable (d, k)-codes. They are often employed in combination with $(dklr)$-sequences, that is, (d, k)-constrained sequences in which the number of leading and trailing zeroes is restricted to be at most l and r respectively [4]. Here, p-bit source symbols are uniquely translated into $(dklr)$-sequences of a fixed length q', and a fixed number of $q - q'$ merging bits, chosen according to a set of *merging rules*, is employed to ensure that the resulting bit stream is a valid (d, k)-sequence. For some recent information on these methods, we refer to [53]. A similar idea is used in [33] to construct *almost-block-decodable* (d, k) codes.

As exemplified by the rate-2/3 Fibonacci code, block-decodable (d, k)-codes with a prescribed rate may require a large codeword size. In these cases, it is often possible to do much better for the corresponding (d, k) RLL constraint, provided that we allow the encoder to employ (one symbol) look-ahead. This recent, rather surprising insight is explained in detail in [32].

1.2.7 Synchronization

For correct functioning, the decoder needs to properly group the bits in the encoded sequence into its composing q-bit codewords. Therefore some form of *syn-*

chronization between encoder and decoder has to be established. We will briefly discuss some means to do so.

Usually, at the encoder side the p-bit source symbols are grouped into *frames*, each consisting of a *fixed* number of source symbols. (Usually, a frame is in fact made up from a fixed number of error-correcting codewords.) The beginning of each frame will be signalled by the occurrence of a special pattern called the *frame header*. Commonly, the frame header is required to be *unique*, that is, it should not occur at any other locations in the sequence. The use of frames and unique frame headers will enable the decoder to recognize the beginning of each frame and thus to establish the desired synchronization.

Often, we require that the header itself, and possibly even the entire bit-stream, also satisfies the imposed constraint. To achieve this, one commonly employs *incidental constraints*. As the name suggests, these are certain patterns that do satisfy the constraint but do not occur as a subword in the output of the encoder during normal operation. If we think of the encoding process of following certain *encoding paths* in a presentation of the constraint, then the incidental constraints are associated with paths that are never used for encoding purposes.

As an example, consider again the rate $2 \to 3$ code for the Fibonacci system constructed in Subsection 1.2.5. It is not difficult to verify that the encoder output never contains the pattern 0.110. We can profit from this fact to design a unique frame header: for example, we could choose as a header the codeword sequence 110.110.101, which possesses the added advantage that it can be inserted anywhere in the codeword stream without violating our constraint, to signal the beginning of each frame.

The presence of noise on the channel may cause bit-errors in a frame header, which may cause the decoder to occasionally miss a header. This problem is usually dealth with by using a PLL-like device similar to the one employed by the receiver to avoid clock-drift of its internal bit-clock.

1.2.8 Code-construction methods

What we would like is a systematic method to design a sliding-block decodable code for a given constrained system $\mathcal{L}$, presented by some deterministic graph $G = (V, A)$, say, at a given rate $p \to q$. Over the years, a great many of such construction methods have been devised. We will review a large number of them here; along the way, we will meet some techniques that we already encountered earlier. Almost all of these methods employ *approximate eigenvectors* to guide the construction. We will first explain what these are and, especially, why this is so. To this end, consider a code meeting our requirements. We will assume that an encoder for our code takes p-bit symbols as its input, and produces q-bit

symbols or codewords as its output. Since we are now interested in admissible sequences of *codewords*, it is more convenient to consider the *qth power graph* G^q of G. This graph has the same states as G, and transitions that consist of paths of length q in G, with as label the q-bit codeword generated by the path. It is now fairly obvious that G^q essentially generates the same sequences as G, but, like our encoder, does so with q bits or one codeword at a time. Note that the adjacency matrix of G^q is D^q, where D is the adjacency matrix of D. We will refer to the system presented by G^q as the *qth power* of the original constraint.

Our encoder translates each sequence of source symbols into a sequence of codewords which, in turn, corresponds to a path in the power graph G^q. We will refer to the collection $\mathcal{C}$ of all such codeword sequences as the *code system* of the code, and to the collection $\mathcal{P}$ of all finite paths in G contained in such encoding paths as the *encoding paths* of the code.

Recall now our investigation in Subsection 1.2.3 of the numbers $\pi_s^{(n)}$ of paths of length n in G that start in state s. We will do something similar here for the numbers $\phi_s^{(n)}$ of encoding paths of length n in G^q that start in state s. Intuitively, it is obvious that these numbers grow exponentially in 2^p, the number of source symbols. ("Encoding paths for different source sequences tend to be different".) Indeed, given our assumptions on $\mathcal{L}$, on G, and on the code, it can be shown that for each state s the limit

$$\phi_s = \lim_{n\to\infty} \phi_s^{(n)}/2^{pn} \tag{11}$$

exists (see [27]). We will write ϕ for the vector with as entries the numbers ϕ_s.

The second observation is that an encoding path of length n starting in s consists of a transition in G^q from s to some state t, say, followed by an encoding path of length $n-1$ starting in t. Therefore the vector $\phi^{(n)}$ with as entries the numbers $\phi_s^{(n)}$ satisfies

$$\phi^{(n)} \le D^q \phi^{(n-1)}. \tag{12}$$

(Note that we have an inequality here since not each path of length n obtained in this way needs to be an encoding path.) If we now combine Equations (11) and (12), we conclude that

$$2^p \phi \le D^q \phi. \tag{13}$$

A non-negative integer vector ϕ that satisfies the inequality (13) is usually called a $[D^q, 2^p]$-*approximate eigenvector*, or a $[G^q, 2^p]$-*approximate eigenvector* if we wish to stress the connection with the presentation. We think of the numbers ϕ_s as weights associated with the states. In terms of these weights, the inequalities state that the sum of the weights of the successors of a state is at least 2^p times the weight of this state. The above discussion makes precise the idea that in code

construction, these weights are indicators for the *relative encoding power* from the states. This idea will be further illustrated by some of our later examples.

Approximate eigenvectors were first introduced by Franaszek. In a series of pioneering papers [13], [14], [15], [16], [17], he systematically employed approximate eigenvectors to develop a number of code construction methods. For example, the *principal-state method* already mentioned is based on the observation that the existence of a binary (0, 1-valued) $[D^q, 2^p]$-approximate eigenvector is equivalent to the existence of a collection of states, the *principal states*, with the property that from each principal state there can be found at least 2^p distinct transitions in G^q, that is, paths of length q in G, that end in another principal state. (To see this, let the principal states correspond to the states with approximate eigenvector-weight one, and now recall that the (s, t)-entry of D^q counts the number of paths of length q from s to t.)

The collection of paths between the principal states immediately allows the construction of a rate $p \to q$ encoder. In each principal state, the 2^p possible source symbols are assigned to q-bit codeword labels of the paths leaving this state. Then the encoder uses, e.g., an encoding table in each principal state to do the translation. (So the encoder does not need to employ look-ahead.) Usually, the encoding tables are implemented by using *enumerative methods* (see [8], [30].

Here, the encoding moves from one principal state to another, and indeed avoids the states with weight zero in G^q. We propose to call such codes *fixed-length principal-state codes*.

If the constraint is of finite type, then any assignment of source symbols to codewords will lead to a sliding-block decodable code: in that case, the sequence of principal states traversed by the encoder can be reconstructed from the sequence of codewords. However, it would of course be preferable to assign a *fixed* source symbol to each codeword, thus making the code block-decodable, and if the constraint is *not* of finite type, this is almost a requirement. In general, finding such a *source-to-codeword assignment* is difficult (in fact, deciding whether such an assignment exists is an NP-complete problem), but in most cases that arise in practice this can be done. The possible remaining freedom in choosing the assignments can be used to optimize other aspects of the code (see, e.g., the design of an 8b10b channel code in [30]).

The method of using merging bits as discussed in Subsection 1.2.6 presents an alternative approach to this labeling problem. Here, we consider only codewords that consist of some merging bits followed by a (fixed-length) $(dklr)$-sequence, and we assign source symbols to the $(dklr)$-sequences. This replaces the labeling problem by the problem of assuring that in each principal state sufficiently many transitions with codeword labels *of this special form* are present. The optimal selection of principal states for (d, k) block-codes has been determined in [19].

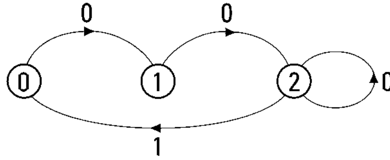

Figure 6: *A presentation of the* $(2, \infty)$*-constraint.*

We will mention two further examples of simple code construction methods that can also be interpreted as principal-state methods. Both of these are designed for the construction of DC-balanced codes. In this context, we usually think of the state of a sequence as the current value of its running digital sum (RDS). Recall that a collection of $\{-1, 1\}$-valued sequences is *balanced* if the RDS of the sequences only take on a limited number of values.

The *disparity* $d(x)$ of a q-bit codeword $x = x_1, \cdots, x_q$, where $x_i \in \{-1, 1\}$, is defined as $d(x) = \sum_{i=1}^{q} x_i$. An obvious way to obtain a DC-balanced code is to use only codewords x with disparity $d(x) = 0$, the *zero-disparity* words. Here the RDS will be zero at the beginning of each new codeword, so this zero-state acts as the single principal state. An obvious extension of the above idea is to use also low-disparity codewords. Here we associate with each source symbol either a zero-disparity codeword, or two codewords with opposite disparities; while encoding we use this freedom-of-choice to keep the RDS close to zero. A particularly simple example of such codes are the *polarity-bit* codes. These are rate $(q-1) \to q$ codes where each source word $x = x_1, \ldots, x_{q-1}$ has two alternative encodings as $x_1, \ldots, x_{q-1}, 1$ or $-x_1, \ldots, -x_{q-1}, -1$, of opposite disparity. Decoding of a codeword can be achieved by inverting all its bits if the last bit, the *polarity bit*, equals -1, followed by deleting this last bit. Here the principal states correspond to the values of the RDS for which $|RDS| \leq q/2$.

The principal-state method is completed by a simple algorithm to find a binary approximate eigenvector or, more generally, an approximate eigenvector with all components less than or equal to any given number M, provided that such a vector exists. Briefly stated, to find such an approximate eigenvector, we initially set $\phi_s = M$ for all states s, and then repeatedly perform the operation

$$\phi \leftarrow \min \{\phi, \lfloor 2^{-p} D^q \phi \rfloor\}$$

(where both the rounding-operation $\lfloor\ \rfloor$ and taking the minimum are done component wise), until either $\phi = 0$ (in which case no such approximate eigenvector exists) or no further change in the vector ϕ occurs (in which case ϕ is an approximate eigenvector with the desired property).

The principal-state method may lead to prohibitively large values of p and q.

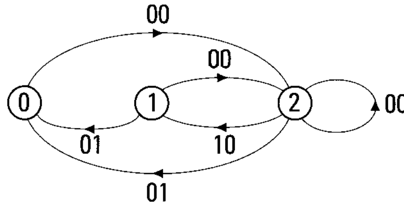

Figure 7: *A presentation of* G^2.

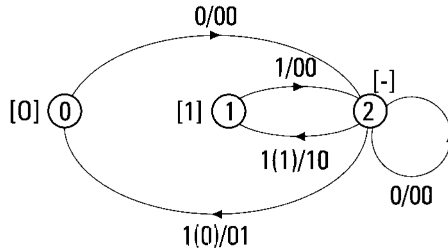

Figure 8: *A look-ahead encoder.*

For example, the minimum codeword length of a fixed-length principal-state rate-2/3 $(1, 7)$-code and a rate-$1/2$ $(2, 7)$-code are 33 and 34, respectively. Sometimes this problem can be avoided if we allow the coding paths between principal states to have varying lengths. As an example, we consider the design of a rate-$1/2$ $(2, \infty)$-code. (The forbidden subwords are given by "11" and "101".) The minimum codeword length of a *fixed-length* rate-$1/2$ code for this constraint is 14 (see, e.g., [30]). The constraint can be presented by a three-state graph G with states named 0, 1, and 2, see Figure 6. Here, sequences ending in state 0 or 1 have precisely 0 or 1 trailing zeroes, respectively, and a sequence ending in state 2 has at least 2 trailing zeroes. The 2nd power graph G^2 of G needed in the construction of a rate $1 \to 2$ code is depicted in Figure 7. The vector ϕ which has $\phi_0 = \phi_1 = 1$ and $\phi_2 = 2$ can be seen to be a $[G^2, 2]$-approximate eigenvector.

Let the set of principal states consist of the single state 2. Now consider the possible encoding paths, that is, paths in G^2 starting and ending in state 2. By inspection of G^2, we obtain the three paths

$$2 \xrightarrow{00} 2, \qquad 2 \xrightarrow{10} 0 \xrightarrow{00} 2, \qquad 2 \xrightarrow{01} 1 \xrightarrow{00} 2,$$

from which a code can now be designed. The encoding rules are specified as

$$\begin{array}{rcl} 0 & \longrightarrow & 00 \\ 10 & \longrightarrow & 01.00 \\ 11 & \longrightarrow & 10.00 \end{array}$$

Table 1: *A two-state encoder for a* $(2, \infty)$*-code.*

State	Input	Output	Next state
σ	0	00	σ
σ	1	00	τ
τ	0	01	σ
τ	1	10	σ

A *look-ahead encoder* that implements these encoding rules is described in Figure 8. In this figure, two labelings are assigned to each transition. One of these labels is the original codeword label; the other is the source symbol labeling and determines the encoding path. For example, the label "0/00" of the transition $0 \longrightarrow 2$ signifies that when in state 0 the source symbol "0" is translated into "00" while the encoder moves from state 0 to state 2. More interestingly, the label "1(0)/01" on the transition $2 \longrightarrow 0$ indicates that when in state 2 the source symbol 1 is translated into "01" while the encoder moves from state 2 to state 0, *provided that the upcoming source symbol is a "0"*. (So here we see the encoder "looking ahead" for the next source symbol before deciding on its action.)

Note that the encoder will possibly move to state 0, but will only do so when the next source symbol to encode is a "0". Therefore all source sequences starting with a "0" (and only these) have to be encodable form state 0. This is indicated by the *prefix list* [0] associated with state 0. For a similar reason, the list [1] is associated with state 1. The *empty list* [−] associated with state 2 indicates that no such condition is imposed in state 2, the principal state. (The list [−] could also be represented as [0, 1].) These prefix lists are not needed for the specification of the encoder; they are merely added to make it easy to verify that the encoding process will never get stuck. The following remark is important in connection with this type of construction method: it is no coincidence that the sizes of the lists [0], [1], and $[-] \equiv [0, 1]$ associated with the states 0, 1, and 2 correspond to the weights $\phi_0 = 1$, $\phi_1 = 1$, and $\phi_2 = 2$ of the $[G^2, 2]$-approximate eigenvector ϕ.

A look-ahead encoder can always be transformed in a systematic way into a conventional finite-state encoder. (The idea is to replace each state s by a *collection* of states (s, w) where w is one of the words in the prefix list associated with this state; the operation is completed by combining, *merging*, equivalent states.) For example, the same encoding rules can be implemented by the two-state encoder as described in Table 1. (Cf. [30], pp. 124, where a three-state encoder is suggested.) A more pictorial representation of this encoder is given in Figure 9. Note that this code can be encoded with a look-ahead of one symbol (Figure 8) or, equivalently, with an encoding delay of one codeword (Figure 9). The code can be decoded with a sliding-block decoder that has a window-size of two codewords.

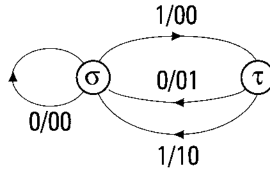

Figure 9: *An alternative description of the encoder.*

In general, for this method to succeed we need a (finite) collection $\mathcal{P}$ of paths π in G^q between principal states such that in each principal state s, we have that

$$\sum 2^{-|\pi|} \geq 1, \tag{14}$$

where the sum is over all paths π starting in s (end ending in another principal state) and where $|\pi|$ denotes the *length* (number of transitions in G^q) of π. This condition, the Kraft-McMillan inequality for prefix codes (see, e.g., [9]), is necessary and sufficient for the existence of a prefix code (such as the source words 0, 10, and 11 in the above example) with word lengths equal to the path lengths. A similar, but more complicated example of the above method is the rate-1/2 (2, 7)-code in [10].

Codes for constraints of finite type found by this method are always sliding-block decodable. We propose to call such codes *variable-length principal-state codes*. The *substitution method* encountered in Subsection 1.2.5 can be seen as a special case of this method.

The principal states in the above methods form a sort of "resting places" for the encoder; here, starting from a principal state any source sequence can be encoded (so these may serve as *initial states* of the encoder) and we are assured that a next such resting place is reached within a finite number of steps. However, codes of this type at a given rate $p \to q$ do not always exist or, if they do, they do not always provide the "simplest" possible code. The *bounded-delay encodable codes* introduced by Franaszek (see also [37]) do not suffer from this drawback; indeed, in its full generality the *bounded-delay method* to construct such codes is *universal*. That is, *each* sliding-block decodable code for a constraint of finite type with a suitable (finite-state, finite-type codeword labeling, bounded encoding delay) encoder can, in principle, be obtained by this method.[5] However, Franaszek never succeeded in turning his method into an algorithm and he showed only much later [18] that this construction method is as powerful as the ACH method. This

[5]In fact, more is true: in principle, any sliding-block decoder, and any encoder that is not a-priory impossible to construct by this method, can indeed be constructed. For further details, see [4] and Chapter 5, in particular Appendix B.

mehod, which we discuss in a moment, was only recently shown to be universal ([2], [25, 26], see also Chapters 4 and 5).

The encoding of a bounded-delay encodable code may depend on the present state, on a bounded number of upcoming source symbols (so *look-ahead* may be employed), and on a bounded number of previous states (the *history*) in the encoding path. The construction method as described by Franaszek employs an approximate eigenvector in an essential way. The idea is to construct in each state a suitable collection of *encoding trees*, also called *independent path sets or IP's*, each consisting of a set of encoding paths for this state, by exhaustive search among all such trees of a fixed maximal depth. The details of the method are not easy to explain, and, as mentioned before, the method cannot be guaranteed to succeed in an intrinsic way.

In [32], a technique called *state-combination* is developed to construct block-decodable codes that can be encoded by employing one-symbol look-ahead, a special case of bounded-delay encodable codes. (This method, especially suited to the construction of (d, k) RLL codes, was earlier referred to in Subsection 1.2.6.) Here, we first search for an approximate eigenvector with all components equal to zero, one, or two (working with the qth power graph). Then we repeatedly look for pairs of states, each having weight one, that can be combined into one new state, which is then given weight two, such that the new weights again constitute an approximate eigenvector, now for the new graph. (Here, a given state gets a transition to such a pair of combined states only if the original graph contained a transition from this state to each of the two combined states, with the same label.) We do so until no further combinations can be made. When our luck is in, there will be sufficiently many transitions from each weight-two state (the "principal states" in the resulting graph) leading to other such states. Finally, to obtain a *block-decodable* code, we have to solve a source-to-codeword assignment problem as referred to earlier. The method is well-suited for computer implementations. Although it will be clear from the above description that success of the method is not guaranteed, it has been successfully applied to construct many codes that improve upon existing ones. The method has been further developed and extended in [25]. (See also [23].)

A breakthrough in code construction was achieved by the invention of the state-splitting method or *ACH algorithm* [1]: this method employs an approximate eigenvector to construct a sliding-block decodable code with a synchronous finite-state encoder, for any given constraint of finite type, at any given rate $p \to q$ for which $p/q \leq C$, the capacity of the constraint. It does so in a number of steps bounded by the sum of the components of the approximate eigenvector.

The method acts on a pair (G, ϕ) of a graph G presenting the qth power of our constraint (that is, transitions of G are labeled with q-bit codewords) and

a $[G, 2^p]$-approximate eigenvector ϕ, and transforms this pair into another such pair by an operation which we will call *weighted state-splitting*. (This operation is called ae-consistent state-splitting in [42], to which we refer for an excellent overview of the method.) This transformation, which is guaranteed to succeed, moreover lowers at least one of the components of the approximate eigenvector, the weights, except when all non-zero weights are equal.[6] So with each transformation the approximate eigenvector gets "smaller", until finally a pair (G, ϕ) is reached where all nonzero components of ϕ are equal. If that is the case, then the approximate eigenvector-inequalities for ϕ are easily seen to imply that in each state with a nonzero weight there are at least 2^p transitions leading to other such states, that is, we have obtained an encoder for our constraint.

State-splitting, on which the transformation is based, is easy to understand. Think of a state as a collection of future opportunities (the transitions leaving this state) from which we are free to choose one. Now divide or *split* the state, this collection of opportunities, in two parts called *sub-states* (each having a part of the original transitions). Before this splitting operation, we could move from another state to this state without worrying about which opportunity (which transition) to utilise later, but now we have to choose first, before moving, from *which* of the two parts we wish to pick our next opportunity. We do not lose opportunities, but we have to choose earlier.

In terms of the graph, this translates as follows. If s is a state, and A_s is the collection of transitions from this state, then to split this state, we do the following. First, partition the set A_s into two *non-empty* parts A_{s_1} and A_{s_2}. Then, in G we replace the state s by two states s_1 and s_2 (the sub-states of s), we replace each transition α in part A_{s_i}, $i = 1, 2$, by a transition from s_i with the same label and ending in the same state as α, and we replace each transition β ending in s by two transitions β_1 and β_2, with the same label and beginning in the same state as β, with β_i ending in s_i, $i = 1, 2$.

It should be evident from the preceding discussion that the new graph thus obtained presents the same sequences as the original graph. Moreover, it is not difficult to see that if the original graph is of finite type, then the new graph is again of finite type; the memory remains the same and the anticipation increases by at most one. Note that if the final graph has memory m and anticipation a, then the encoder obtained from this graph by deleting all states with zero weight and assigning suitable source labels to the remaining transitions has a decoding window of size at most $m + 1 + a$.

Weighted state-splitting amounts to the same, except that now we also distribute the weight of the state that is split over the sub-states (where each of

[6] This gives the general picture, but is not quite true. More details will be given later.

the sub-states should receive *something*); moreover, we require that the resulting weights constitute an approximate eigenvector for the new graph. Note that this is a *local* requirement which only needs to be checked for the two new states.

It is not obvious that we can always find a state for which weighted state-splitting is possible; certainly this need not be possible in each state. However, in [1] it is shown that such a state can always be found among the states with maximum weight, provided that not all non-zero weights are equal, and an algorithm is given to find such a state.[7]

The ACH algorithm is again universal ([2], [26, 25], see also Chapters 4 and 5). However, this method is not without drawbacks. The main problem is the potentially large amount of freedom-of-choice afforded by the algorithm, first in the choice of the approximate eigenvector, then both in the choice of the states to be split and in the actual splits themselves. Different choices (almost always) lead to different codes and, by the same universality that we so much appreciated earlier, may result in any of the (infinitely many) available codes.

As a rule-of-thumb, we usually choose the *smallest possible* approximate eigenvector; a logical choice since this minimizes the a-priory upper bound on the number of state-splitting steps. This works very well in practice, although cases are known where this choice does not produce the best possible code [25].

Still this solves only part of the problem. Some heuristics to guide the state-splitting process have been developed (see, e.g., [42]) in order to obtain codes with a simple *encoder* (simple in terms of the number of encoder states), and again this seems to work reasonably well in practice. (Methods to estimate the minimum possible number of encoder states have been developed in [41].)

However, in practical applications we are often interested in codes with a small *decoding window*, and the ACH algorithm provides no clues how to find such codes. A variant of this algorithm [25] combines ideas from the bounded-delay method of Franaszek with state-splitting to address this problem.

We will end this overview with a discussion of a very recent enumerative method, originally developed to design *high-rate* codes such as $(0, k)$-codes with large k [34, 35]. But in fact, as we will see, this method is sufficiently general to describe many variable-length codes as well. We will first describe the method as a means to design simple-to-implement fixed-length block-decodable codes with a (large) codeword size q, say. Our starting point is a deterministic graph $G = (V, A)$ presenting our constraint. We will denote the bit-label of a transition α by $L(\alpha)$.

[7] In fact, this is only true when, after each splitting step, states with zero weight are deleted and the graph is restricted to an irreducible sink component. Note that after such an operation, the weights still constitute an approximate eigenvector for the remaining part of the graph.

The encoding- and decoding rules of our code will be described in terms of non-negative integers $N_n(v)$, $n = 0, 1, \ldots, q$, and $N_n(\alpha)$, $n = 1, \ldots, q$, associated with the states v and transitions α of our presentation G. Here, all numbers $N_0(v)$ will be either zero or one, i.e.,

$$N_0(v) \in \{0, 1\}, \tag{15}$$

for all states v. (These numbers will need to satisfy various other constraints that will be discussed later.)

Our encoder will in fact encode *numbers* N with $0 \leq N < 2^p$; the *binary representation* of these numbers will then correspond to the p-bit source symbols. Our encoding process will be described recursively in terms of encoding rules for numbers in each state. Here, the number $N_n(v)$ will specify the interval of numbers that can be encoded from state v by (the n-bit label sequence of) an encoding path of length n; the first transition α of this path will lower the number to be encoded by the amount of $N_n(\alpha)$. This leads to the following recursive encoding rule: the n-bit encoding $E_v^{(n)}(N)$ of a number N, $0 \leq N < N_n(v)$, from state v is given as

$$E_v^{(n)}(N) = L(\alpha) E_{\mathrm{end}(\alpha)}^{(n-1)}(N - N_n(\alpha))$$

("the label of α, followed by an $(n-1)$-bit encoding"), where α is a transition starting in state v chosen such that

$$0 \leq N - N_n(\alpha) < N_{n-1}(\mathrm{end}(\alpha)).$$

Recall that the numbers $N_0(v)$ are each either zero or one. Therefore, if the encoding of a number N succeeds, then the encoding process will necessarily terminate with the number zero, in one of the *terminal states* t for which $N_0(t) = 1$. We will use T to denote the set of these terminal states. As a consequence, if the encoder has followed the path $\alpha_n, \ldots, \alpha_1$, then the number N can be recovered as the sum

$$N = \sum_{i=1}^{n} N_i(\alpha_i) \tag{16}$$

of numbers subtracted at the successive transitions. So decoding is possible provided that the encoding path can be retrieved from the label sequence, which is possible if the constraint is of finite type. (If this is not the case, then in addition we should require that paths generating identical sequences also produce identical decodings. A detailed discussion of that case is outside the scope of this review.)

As we observed, the encoding always terminates in a state from T. Therefore, the next encoding of a p-bit number *starts* in such a state; these states serve as the

principal states in this method. So in each of these states, we must ensure that at least 2^p numbers can be encoded, that is, we require that

$$N_q(t) \geq 2^p,$$

for all states t from T.

Now we will address the question which further conditions the numbers $N_n(v)$ and $N_n(\alpha)$ have to satisfy. We first note that at the end of each transition α from state v, at most $N_{n-1}(\text{end}(\alpha))$ numbers are guaranteed to be encodable into an $(n-1)$-bit code sequence. Since we require encoding into n-bit code sequences of $N_n(v)$ numbers in state v, we necessarily have for all states v and all n, $n = 1, \ldots, q$, that

$$N_n(v) \leq \sum N_{n-1}(\text{end}(\alpha)), \tag{17}$$

where the sum is over all transitions α leaving v.

On the other hand, once we have a collection of numbers $N_n(v)$ satisfying (17), we can easily determine suitable numbers $N_n(\alpha)$ as follows. In each state v, we choose a linear order "$<$" of the set A_v of transitions leaving v, and then we define

$$N_n(\alpha) = \sum_{\beta \in A_v, \beta < \alpha} N_{n-1}(\text{end}(\beta)).$$

Note that the for the first transition γ in each set A_v we have $N_n(\gamma) = 0$ for all n. In particular, if we assume (which in fact we always did, but do not really need here) that the graph is deterministic and has binary labels, then in each state v either one transition α leaves this state and we take $N_n(\alpha) = 0$ for all n, or two transitions, α and β, say, leave this state and we could take $N_n(\alpha) = 0$ and $N_n(\beta) = N_{n-1}(\text{end}(\alpha))$, for all n.

If all the conditions mentioned up to now are satisfied, then we obtain a rate $p \to q$ fixed-length block-decodable code as desired. We propose the name *enumerative codes* for codes that employ encoding and decoding of this type.

As an example, consider the $(0, k)$ constraint. This constraint can be presented by a graph with $k+1$ states $0, 1, \ldots, k$, transitions $v \xrightarrow{0} v+1$, $v = 0, \ldots, k-1$, and transitions $v \xrightarrow{1} 0$, $v = 0, \ldots, k$.

It is not difficult to see that the inequalities (17) imply that the numbers $N_n = N_n(0)$ must satisfy

$$N_n \leq N_{n-1} + \cdots N_{n-k-1}, \tag{18}$$

$n \geq k+1$, and

$$N_n \leq N_0(n) + N_{n-1} + \cdots + N_0, \tag{19}$$

$1 \leq n \leq k$. On the other hand, if we choose, e.g.,

$$N_0(0) = \cdots = N_0(r) = 1, N_0(r+1) = \cdots = N_0(k) = 0,$$

then given numbers N_n that satisfy (18) and (19), we let

$$N_n(0) = N_n,$$

for all $n \geq 1$, and, for states $v = 1, \ldots, k$, we let

$$N_n(v) = N_{n-1} + \cdots + N_{n-k-1+v},$$

for $n \geq k + 1 - v$, and

$$N_n(v) = N_0(v + n) + N_{n-1} + \cdots + N_0,$$

for $1 \leq n \leq k - v$. Then the numbers $N_n(i)$ satisfy (17).

Next, let $N_n(\alpha) = 0$ if α has label "1" and let $N_n(\alpha) = N_{n-1}$ if α has label "0". Then all requirements are satisfied and if $N_q(v) \geq 2^p$, $v = 0, \ldots, r$, we obtain a code with rate $p \to q$. It is also easy to see directly that the encoded sequence satisfies the k-constraint. Indeed, at any stage n we encode numbers N with $0 \leq N < N_n$; since encoding into a "0" at stages $n-1, n-2, \ldots$ subtracts $N_{n-1}, N_{n-2}, \ldots$, the inequality (18) implies that we will do this at most k times. (More details concerning $(0, k)$-codes of this type can be found in [34] and [45].

An encoder and decoder for a code of this type need not necessarily be very simple. To achieve that, we need one further idea. We have seen that *any* collection of numbers $N_n(v)$ that satisfy both (15) and (17) can be used to devise an enumerative code. We now try to satisfy these conditions with numbers *of a simple form*. In particular, we will choose these numbers in such a way that, for each fixed n, all numbers $N_n(v)$ can be written as

$$N_n(v) = \nu_v^{(n)} 2^{e_n} \tag{20}$$

with

$$0 \leq \nu_v^{(n)} < 2^m.$$

Here m will be a fixed number determined by the desired implementation complexity. Note that now both the encoding- and decoding operations can be implemented as addition or subtraction on numbers represented by m+$\max_n(e_n - e_{n-1})$ bits, provided that we implement the decoding rule (16) in *reversed* order, that is, starting with $N_1(\alpha_1)$ and ending with $N_q(\alpha_q)$. (During read-back, the encoding path will be recovered in the order $\alpha_q, \ldots, \alpha_1$.)

We will now describe how to find the largest numbers $N_n(v)$ of the form (20) and meeting our requirements. The construction is best described in terms of a vector-rounding operation. Given a number m and a vector ν, we define the rounded vector

$$\lfloor \nu \rfloor_m$$

and the width $e(\nu)$ of ν as follows. If the largest entry of ν has an e-bit representation, then we set

$$e(\nu) = \max(0, e - m),$$

and the vth entry of $\lfloor \nu \rfloor_m$ will consist of the number represented by the m most significant bits in the e-bit representation of ν_v, the corresponding entry of ν.

We now recursively define the vectors $\nu^{(n)}$, with entries the numbers $\nu_v^{(n)}$, and the numbers e_n in (20) by letting $e_0 = 0$, $\nu_v^{(0)} = N_0(v)$,

$$e_n = e_{n-1} + e(D\nu^{(n-1)}),$$

and

$$\nu^{(n)} = \lfloor D\nu^{(n-1)} \rfloor_m.$$

Once the construction is understood, it is fairly easy to see that it indeed produces the largest numbers of the form (20) that satisfy our requirements.

It is important to observe that both the sequence of vectors $\nu^{(n)}$ and the sequence of numbers $e_n - e_{n-1}$ ultimately become *periodic*. Indeed, since all entries of these vectors consist of m-bit numbers, there are only finitely many different vectors in the sequence. So there are numbers s and c such that $\nu^{(c+s)} = \nu^{(c)}$. But then we obviously have that $\nu^{(n+s)} = \nu^{(n)}$ and $e_{n+s+1} - e_{n+s} = e_{n+1} - e_n$ for *all* $n \geq c$. From the second of these relations we can derive that $e_{n+s} = r + e_n$ for all $n \geq c$, where $r = e_{c+s} - e_c$, As a consequence, for large q the rate of the code thus obtained tends to $R_m = r/s$, a limit to the achievable rate that depends only on m.

It is instructive to apply this construction to the Fibonacci system again. With $m = 1$, the numbers $N_n = N_n(0)$ thus obtained are given by

$$N_{2n} = N_{2n+1} = 2^n,$$

for all $n \geq 0$. A moment of reflexion reveals that the rate $2p \to p$ enumerative code thus obtained is essentially the FM code from Subsection 1.2.5! Similarly, for $m = 3$ we obtain that $N_0 = N_1 = 1$ and

$$N_{3n-1} = 2^{2n-1}, \qquad N_{3n} = 3 \cdot 2^{2n-2}, \qquad N_{3n+1} = 5 \cdot 2^{2n-2},$$

for $n \geq 1$, which gives us a rate $(2n - 1) \to (3n - 1)$ enumerative code for all $n \geq 1$.

We are now ready to explain how this type of encoding can be implemented as operating on a *bit-stream* instead of on p-bit blocks. To this end, we think of an input sequence

$$x_{-1}, x_{-2}, x_{-3}, \ldots$$

(note the use of negative numbers as indices here) as representing the fractional number

$$N = 0.x_{-1}x_{-2}x_{-3}\cdots.$$

(Note that $0 \le N < 1$.) Again, we represent the numbers $N_n(v)$, $n = -1, -2, \ldots$ in the form (20), but now we choose for the numbers $\nu_v^{(n)}$ and e_n the "periodic" solution of the recursive relations.

To illustrate the above we consider again the $(2, \infty)$-constraint for which we constructed a variable-length code earlier. The graph G presenting this constraint was described in Figure 6. Now, for $n = 0, -1, -2, \ldots$, define

$$N_{2n}(0) = 2^{n-1}, \qquad N_{2n-1}(0) = 2^{n-2},$$

$$N_{2n}(1) = N_{2n-1}(1) = 2^{n-1},$$

$$N_{2n}(2) = 2^{n}, \qquad N_{2n-1}(2) = 2^{n-1} + 2^{n-2},$$

$$N_n(0 \longrightarrow 1) = N_n(1 \longrightarrow 2) = N_n(2 \longrightarrow 2) = 0,$$

and

$$N_{2n}(2 \longrightarrow 0) = 2^{n-1} + 2^{n-2}, \qquad N_{2n-1}(2 \longrightarrow 0) = 2^{n-1}.$$

These numbers satisfy all our requirements, and, since $N_0(2) = 1$, we can encode all fractional numbers N, $0 \le N < 1$, starting from state 2. The rate of the resulting code is $R = R_2 = 1/2$. In fact, it is not difficult to see that this code essentially is the same as the code that we constructed earlier!

As remarked earlier, a simple hardware implementation can be found to decode such codes, provided that decoding takes place in bit-reversed order, starting with the last received bit and working our way back to the first received bit. This is acceptable for a block-code variant of such a code, but not at all for the periodic variant discussed above. Nevertheless, in certain cases simple decoding in normal bit order is still possible. For example, the $(2, \infty)$ code constructed above has this property since previously we showed that this code is sliding-block decodable.

The additional requirement that the code must be decodable in normal bit order, or, equivalently, that the code must be sliding-block decodable, further restricts the choice of the numbers $N_n(v)$ that define the code. A procedure similar to the one discussed earlier can be used to find "periodic" solutions that meet these additional requirements, but some experiments for the case of (d, k)-codes have shown that in most cases the efficiency of such "automatically generated" codes is very low. Further developments have to be awaited to see if these *periodic enumerative codes* are of more than theoretical interest only.

Fixed-length enumerative codes suffer from catastrophic error-propagation within each block. (Here we assume that their periodic counterpart is not sliding-block decodable.) If the codeword size is very large, such codes are therefore used

in combination with special error-correcting schemes designed to avoid excessive error-propagation [34, 35]. Such schemes are typified by the fact that, essentially, the modulation and ECC encoding in Figure 1 have been interchanged.

Up to now, we have discussed various methods to construct modulation codes. In a given situation, which method should we choose; which method is the best? This seemingly innocent question has caused some controversies, as each method has its own adherends. The answer, of course, is "the method that best suits the case at hand", and this depends mainly on the number of q-bit codewords per state, that is, on the maximum number of transitions leaving a state in the power graph G^q. (Here we assume that the constraint is presented by a graph G and that we search for a code at a rate $p \to q$.)

If this number is relatively *small*, then use an ACH-type method. (In my opinion, to date the best method of this kind is the one described in [25].) On the other hand, if this number is relatively *large*, then such methods are less suitable. (The main reason lies in the increasing number of choices for the state-splitting steps as referred to earlier.) In that case, use a method based on merging-bits, or a method for (almost) block-decodable codes such as [32] or [26], or enumerative encoding, or any other more heuristic method that does the job.

1.3 Overview of the contents of this thesis

This thesis concerns performance evaluations and construction methods for modulation codes. In Chapter 1 of this thesis (the present chapter), we first discuss the schematic form of a modern digital transmission- or storage system in which modulation codes are employed.

After a brief review of the role other parts of such a system, we present a broad overview of the subject of modulation codes. Here we discuss some of the practical considerations behind the use of constrained sequences in data transmission and storage, and numerous aspects of the modulation codes designed to intermediate between arbitrary and constrained sequences: type of constraints to code for, coding rate and capacity, encoders, decoders, synchronization, and code-construction methods.

In the next two chapters, which are co-authored by K.A.S. Immink, we are interested in performance evaluations of certain type of codes. In Chapter 2, we compare two DC-balanced codes, namely the polarity-bit code already encountered in Subsection 1.2.8 and the Knuth code, a special type of zero-disparity code (see again Subsection 1.2.8).

As mentioned in Subsection 1.2.1, the sum-variance of a DC-balanced code at a given rate is a good performance criterium for such a code. The sum-variance

of a polarity-bit code is known to be a simple function of the codeword size, and hence of the rate, of the code. Things are not so simple for a Knuth code. My main contribution is an exact evaluation by combinatorial means of a good approximation to the sum-variance of a Knuth code. The result shows that the sum-variance of a Knuth code at a given rate is significantly larger than that of a polarity-bit code at the same rate, thus suggesting that polarity-bit codes have a greater spectral effectiveness than Knuth codes.

Magnetic recording systems commonly employ (d, k)-constrained sequences to increase the efficiency of the recording channel. If such a constraint is used in combination with a synchronization method based on frames and unique headers as discussed in Subsection 1.2.7, then the requirement that the pattern used as header is not allowed to occur in the bit-stream except at the beginning of a frame imposes an additional constraint on the sequences. In Chapter 3, we investigate the loss of capacity entailed by imposing such an additional constraint. My main contribution is a method based on generating functions to compute the capacity, the maximum possible coding rate, of this combined constraint, for any given header.

In the following three chapters, we introduce and investigate the class of bounded-delay codes or BD codes, the class of sliding-block decodable codes for constraints of finite type that can be encoded with a bounded encoding delay or, equivalently, by employing a bounded amount of look-ahead.

The ACH algorithm is one of the most important code construction methods for sliding-block decodable codes. As explained in Subsection 1.2.8, a drawback of this method lies in the great freedom of choosing the state-splitting steps, especially since the algorithm offers few clues how to strive for a code with a small decoding window.

The work in Chapter 4 addresses this problem. In this chapter, we first present a precise definition and a thorough discussion of look-ahead encoders. Then we develop an algorithm especially designed to construct codes with a small decoding window. Our algorithm employs an approximate eigenvector and combines the weighted state-splitting from the ACH algorithm with ideas from the bounded-delay method of Franaszek. It is based on a new "local" construction method for BD codes where in each state of the weighted graph a certain partition problem has to be solved. This partition problem can be interpreted as the problem of designing a generic "local" look-ahead encoder in the state; this problem is easy to solve as long as the number of transitions leaving the state is not too large.

Once all these partition problems are successfully solved, it is easy to construct the desired BD code; here we can use the remaining freedom in the construction to optimize the decoding window of the resulting code. This part of the method is fully automatic and can be easily implemented in a computer program

if desired.

It may happen, however, that one or more of the "local" problems have no solution. Such instances can always be removed by employing weighted state-splitting to the successors of states where such difficulties occur. Moreover, these very difficulties now offer additional clues to guide the state-splitting process. Our approach makes it easy to obtain both various well-known and new codes with a small decoding window, as is illustrated by a number of examples.

In Chapter 5, we investigate block-decodable BD codes or BDB codes. We present a universal construction method for such codes; each such code can be described and constructed in a specific way from the Fisher cover (see Subsection 1.2.4) for the constraint. We use this result to show that the code construction method in the previous chapter as well as the ACH algorithm are essentially universal (most of these results were first obtained by other means in [2]) and to solve some basic decision problems concerning block-decodable codes.

Our construction method for BDB codes is particularly well-suited for the construction of one-symbol look-ahead BDB codes. This is illustrated by the work in Chapter 6, where we employ the method to construct a block-decodable rate $8 \to 12$ $(1, 8)$ RLL code. Such a code is attractive for use in combination with symbol-error-correcting codes such as Reed-Solomon codes based on an eight-bit alphabet.

We have seen that many of the known code-constructions method make essential use of an approximate eigenvector, where different choices of this vector in general lead to different codes. In Chapter 7, we try to clarify this intimate relation of codes and approximate eigenvectors. First we offer a construction which, given a (code system of a) modulation code for a constraint of finite type and a presentation of finite type for this constraint, produces an approximate eigenvector for that presentation. Then we show that the approximate eigenvectors from which the ACH algorithm can construct our code each involve weights that are at least as large as the corresponding weights in this special approximate eigenvector. We offer arguments which justify calling this vector "*the approximate eigenvector*" of the code.

Chapters 2, 3, 4, 5, and 6 have appeared earlier as [21], [31], [25], [26], and [24], respectively (up to some minor changes). Chapter 7 is based on [27], which is accepted for publication.

References

[1] R.L. Adler, D. Coppersmith, and M. Hassner, "Algorithms for sliding block codes. An application of symbolic dynamics to information theory", *IEEE*

Trans. Inform. Theory, vol. IT-29, no. 1, pp. 5–22, Jan. 1983.

[2] J. Ashley and B.H. Marcus, "Canonical encoders for sliding block decoders", *Siam J. Disc. Math.*, vol. 8, no. 4, pp. 555–605, Nov. 1995.

[3] M.-P. Béal, *Codage symbolique*, Paris: Masson, 1993.

[4] G.F.M. Beenker and K.A.S. Immink, "A generalized method for encoding and decoding run-length-limited binary sequences", *IEEE Trans. Inform. Theory*, vol. IT-29, no. 5, pp. 751–754, Sept. 1983.

[5] S. Benedetto and G. Montorsi "Unveiling turbo codes: some results on parallel concatenated coding schemes", *IEEE Trans. Inform. Theory*, vol. 42, no. 2, pp. 409–429, June 1996.

[6] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: turbo codes", *Proc. ICC '93*, pp. 1064-1070.

[7] A.R. Calderbank (Ed.), *Different aspects of coding theory*, Providence, RI: AMS, 1995.

[8] T.M. Cover, "Enumerative source coding", *IEEE Trans. Inform. Theory*, vol. IT-19, pp. 73–77, Jan. 1973.

[9] T.M. Cover and J.A. Thomas, *Elements of information theory*, New York: Wiley, 1991.

[10] J.S. Eggenberger and P. Hodges, "Sequential encoding and decoding of variable word length, fixed rate data codes", U.S. Patent 4,115,768, 1978.

[11] M. Feteke, *Math. Zeitschr.*, vol. 17, pp. 233, 1923.

[12] G.D. Forney, Jr., "Generalized minimum distance decoding", *IEEE Trans. Inform. Theory*, vol. 12, pp. 125–131, 1966.

[13] P.A. Franaszek, "Sequence-state coding for digital transmission", *Bell Syst. Tech. J.*, vol. 47, pp. 143–157, Jan. 1968.

[14] P.A. Franaszek, "On future-dependent block coding for input-restricted channels", *IBM J. Res. Develop.*, vol. 23, pp. 75–81, Jan. 1979.

[15] P.A. Franaszek, "Synchronous bounded delay coding for input restricted channels", *IBM J. Res. Develop.*, vol. 24, pp. 43–48, Jan. 1980.

[16] P.A. Franaszek, "A general method for channel coding", *IBM J. Res. Develop.*, vol. 24, pp. 638–641, Sept. 1980.

[17] P.A. Franaszek, "Construction of bounded delay codes for discrete noiseless channels", *IBM J. Res. Develop.*, vol. 26, pp. 506–514, July 1982.

[18] P.A. Franaszek, "Coding for constrained channels: A comparison of two approaches", *IBM J. Res. Develop.*, vol. 33, pp. 602–608, Nov. 1989.

[19] J. Gu and T. Fuja, "A new approach to constructing optimal block codes for runlength-limited channels", *IEEE Trans. Inform. Theory*, vol. IT-40, no. 3, pp. 774–785, 1994.

[20] J. Gu and T. Fuja, "A note on look-ahead in the encoders of block-decodable (d, k) codes", preprint.

[21] H.D.L. Hollmann and K.A.S. Immink, "Performance of efficient balanced codes", *IEEE Trans. Inform. Theory*, vol. 37, no. 3, pp. 913–918, May 1991.

[22] H.D.L. Hollmann, "The general solution of write equalization for RLL (d, k) codes", *IEEE Trans. Inform. Theory*, vol. 37, no. 3, Part II, pp. 856–862, May 1991, special issue on "Coding for storage devices".

[23] H.D.L. Hollmann, "Bounded-delay encodable codes for constrained systems from state combination and state splitting", *Benelux Symp. Inform. Theory*, Veldhoven, May 1993.

[24] H.D.L. Hollmann, "A block-decodable $(d, k) = (1, 8)$ run-length-limited rate $8/12$ code", *IEEE Trans. Inform. Theory*, vol. 40, no. 4, pp. 1292–1296, July 1994.

[25] H.D.L. Hollmann, "On the construction of bounded-delay encodable codes for constrained systems", *IEEE Trans. Inform. Theory*, vol. 41, no. 5, pp. 1354–1378, Sept. 1995.

[26] H.D.L. Hollmann, "Bounded-delay encodable, block-decodable codes for constrained systems", *IEEE Trans. Inform. Theory*, vol. 42, no. 6, Nov. 1996, special issue on "Codes and complexity".

[27] H.D.L. Hollmann, "On an approximate eigenvector associated with a modulation code", accepted for publication in *IEEE Trans. Inform. Theory*.

[28] J.E. Hopcroft and J.D. Ullmann, *Formal languages and their relation to automata*, Reading, MA: Addison-Wesley, 1969.

[29] J.E. Hopcroft and J.D. Ullmann, *Introduction to automata theory, languages, and computation*, Reading, MA: Addison-Wesley, 1979.

[30] K.A.S. Immink, *Coding techniques for digital recorders*, Englewood Cliffs, NY: Prentice-Hall, 1991.

[31] K.A.S. Immink and H.D.L. Hollmann, "Prefix-synchronized run-length-limited sequences", *IEEE J. Sel. Areas Commun.*, vol. 10, no. 1, pp. 214–222, Jan. 1992.

[32] K.A.S. Immink, "Block-decodable run-length-limited codes via look-ahead technique", *Philips J. Res.*, vol. 46, no. 6, pp. 293–310, 1992.

[33] K.A.S. Immink, "Constructions of almost block-decodable runlength-limited codes", *IEEE Trans. Inform. Theory*, vol. 41, no. 1, pp. 284–287, Jan. 1995.

[34] K.A.S. Immink, "A practical methos for approaching the channel capacity of constrained channels" submitted to *IEEE Trans. Inform. Theory*.

[35] K.A.S. Immink, "EFM coding: squeezing the last bits" submitted to *IEEE Trans. Consumer Electr.*.

[36] G. Jacoby and R. Kost, "Binary two-thirds rate code with full word look-ahead", *IEEE Trans. Magnet.*, vol. MAG-20, no. 5, pp. 709–714, Sept. 1984.

[37] A. Lempel and M. Cohn, "Lookahead coding for input-restricted channels", *IEEE Trans. Inform. Theory*, vol. IT-28, pp. 933–937, Nov. 1982.

[38] D. Lind and B. Marcus, *An introduction to symbolic dynamics and coding*, Cambridge, MA: Cambridge University Press, 1995.

[39] J.H. van Lint, *Coding theory*, New York: Springer, 1971.

[40] F.J. MacWilliams and N.J.A. Sloane, *The theory of error-correcting codes*, Amsterdam: Elsevier, 1977.

[41] B.H. Marcus and R. Roth, "Bounds on the number of states in encoder graphs for input-constrained channels", *IEEE Trans. Inform. Theory*, vol. IT-37, no. 3, pp. 742–758, May 1991.

[42] B.H Marcus, P.H. Siegel, and J.K. Wolf, "Finite-state modulation codes for data storage", *IEEE J. Sel. Areas Commun.*, vol. 10, no. 1, pp. 5–37, Jan. 1992.

[43] H. Minc, *Nonnegative matrices*, New York: Wiley, 1988.

[44] H. Nyquist, "Certain factors affecting telegraph speed", *Bell Syst. Tech. J.*, vol. 3, p. 324, 1924.

[45] L. Pátrovics and K.A.S. Immink, "Encoding of $dklr$-sequences using one weight set", *IEEE Trans. Inform. Theory*, vol. 42, no. 5, pp. 1553–1554, Sept. 1996.

[46] G. Pólya and G. Szegö, *Aufgaben und Lehrsätze aus der Analysis*, Berlin: Springer, 1970.

[47] E. Seneta, *Non-negative matrices and Markov chains (Second ed.)*, New York: Springer, 1981.

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

[49] C. Shannon, "Communication in the presence of noise", *Proc. IRE*, vol. 37, pp. 10–21, 1949.

[50] R.C. Schneider, "Write equalization in high-density magnetic recording", *IBM J. Res. Develop.*, vol. 29, no 6, Nov. 1985.

[51] R.C. Schneider, "Write equalization for generalized (d, k) codes", *IEEE Trans. Magnet.*, vol. MAG-24, no. 6, pp. 2533–2535, Nov. 1988.

[52] L.M.G. Tolhuizen, "Cooperating error-correcting codes and their decoding", Doctoral thesis, Eindhoven Un. Techn., Eindhoven, June 1996.

[53] J.H. Weber and K.A. Abdel-Ghaffar, "Cascading Runlength-Limited Sequences", *IEEE Trans. Inform. Theory*, vol. IT-39, no. 6, pp. 1976–1984, Nov. 1993.

[54] B. Weiss, "Subshifts of finite type and sofic systems", *Monats. Math.*, vol. 77, pp. 462–474, 1973.

[55] H.W. Wong-Lam, H.D.L. Hollmann, "A general solution of write equalization for digital magnetic recording", *IEEE Trans. Magnet.*, vol. MAG-27, no. 5, pp. 4377–4381, Sept. 1991.

[56] A.D. Wyner, "Capabilities of bounded discrepancy decoding", *Bell Syst. Tech. J.*, vol. 54, pp. 1061–1122, 1965.

Chapter 2

Performance of Efficient Balanced Codes

Abstract— **Recently, Knuth presented coding schemes in which each codeword contains equally many "zeros" and "ones". The construction has the virtue that the codes can be efficiently encoded and decoded. In this correspondence, we address the problem of appraising the spectral performance of codes based on Knuth's algorithm.**

Index Terms — **dc-balanced codes, Knuth codes, sum variance.**

2.1 Introduction

Binary sequences with spectral nulls at zero frequency have found widespread application in optical and magnetic recording systems. These *dc-balanced codes*, as they are often called, have a long history and their application is certainly not confined to recording practice. Since the early days of digital communication over cable, dc-balanced codes have been employed to counter the effects of low-frequency cut-off due to coupling components, isolating transformers, etc. In optical recording, as explained in [3], dc-balanced codes are employed to circumvent or reduce interaction between the data written on the disc and the servo systems that follow the track. A present-day application of dc-free codes [4] is the digital audio tape recorder which uses an 8b10b code to circumvent the effects of crosstalk from adjacent tracks, and to minimize over-write noise.

© 1991, IEEE. Reprinted, with permission, from IEEE Transactions on Information Theory vol. 37, no. 3, pp. 913–918, May 1991.

Co-authored by K.A. Schouhamer Immink.

This work was presented in part at the IEEE International Workshop on Information Theory, Veldhoven, The Netherlands, 1990.

Practical coding schemes devised to achieve suppression of low-frequency components are mostly constituted by block codes. The (bipolar) source digits are grouped in source words of m digits; the source words are translated using a conversion table known as a codebook into blocks of n digits. The essential principle of operation of a channel encoder that translates arbitrary source data into a dc-free channel sequence is remarkably simple. The approaches which have actually been used for dc-balanced code design are basically three in number: zero-disparity code, low-disparity code, and polarity bit code.

The *disparity* of a codeword [6] is defined as the sum of its digits; thus the codewords $-1, -1, -1, 1, 1, -1$ and $1, -1, -1, 1, 1, 1$ have disparity -2 and $+2$, respectively. Of special interest are zero-disparity codewords. The obvious method for the construction of dc-balanced codes is to only employ zero-disparity codewords.

A logical step, then, is to extend this mechanism to the *low-disparity* code, where the translations are not one-to-one. The source words operate with two alternative translations (or modes) which are of equal or opposite disparity; each of the two modes is interpreted by the decoder in the same way. The zero-disparity words are uniquely allocated to the source words. Other codewords are allocated in pairs of opposite disparity. During transmission, the choice of a specific translation is made in such a way that the accumulated disparity, or the *running digital sum*, of the encoded sequence, after transmission of the new codeword, is as close to zero as possible. The running digital sum (RDS) is defined as the accumulated sum of the transmitted digits, counted from the start of the transmission. Both of the basic approaches to dc-balanced coding are due to Cattermole [5], [6] and Griffiths [7].

A third coding method (in fact a special case of low-disparity codes), known as the *polarity bit code*, was devised by Bowers [8] and Carter [9]. They proposed a slightly different construction of dc-balanced codes as being attractive because no look-up tables are required for encoding and decoding. In their method, each group of $(n - 1)$ source symbols are supplemented by the symbol "1". The encoder has the option to transmit the resulting n-bit words without modification or to invert all symbols. Like in the low-disparity code, the choice of a specific translation is made in such a way that the accumulated disparity is as close to zero as possible. The last symbol of the codeword, called the polarity bit, is used by the decoder to identify whether the transmitted codeword has been inverted or not.

Quite recently [2], a new algorithm for generating zero-disparity codewords was presented by Knuth. The method is based on a simple correspondence between the set of all m-bit source words and the set of all $(m + p)$-bit balanced codewords. The translation is in fact achieved by selecting a bit position within the m-bit word which defines two segments, each having one half of the total

block disparity. A zero-disparity block is now generated by the inversion of all the bits within one segment. The remaining p bits of the codeword contain a balanced encoding of the bit position that defines the two segments. For a precise description of this method, we refer to Section 2.2.

The outline of this paper is as follows. In order to get some insight into the efficiency of Knuth's construction technique we shall evaluate the spectral properties of its code streams. Of course, the spectrum may be evaluated for any given code structure by resorting to numerical computation. The theory provided in [10] furnishes efficient procedures for the computation of the power spectral density function of block-coded signals produced by an encoder that can be modelled by a finite-state machine. However, the computational load of this procedure is enormous if $m \gg 1$. Fortunately, the structure of Knuth codes allows us to derive a simple expression for (an approximation to) the *sum variance* of these code. This quantity plays a key role in the spectral performance characterization of dc-balanced codes, as explained in Section 2.3. We shall evaluate this expression in Section 2.4. In Section 2.5, we compare the sum variance of Knuth codes with the sum variance of the polarity bit codes, for fixed redundancy.

2.2 Balancing of Codewords

Most schemes for generating dc-balanced sequences use look-up tables, and are therefore restricted to codewords of medium size. An alternative and easily implementable encoding technique for zero-disparity codewords which is capable of handling (very) large blocks was described by Knuth [2]. The method is based on the idea that there is a simple correspondence between the set of all m-bit binary source words and the set of all $(m + p)$-bit codewords. For the sake of convenience, we will assume in the sequel that both m and p are even. (Similar constructions as described here are possible if one or both of p, m are odd.) Then the translation can in fact be achieved by selecting a bit position within the m-bit word that defines two segments, each having one half of the total block disparity. A zero-disparity block is now generated by the inversion of all the bits within one segment. The position information which defines the two segments is encoded in the p bits by a balanced word.

Let $z_k(x)$ be the running digital sum of the first k, $k \leq m$, bits of the binary source word $x = (x_1, \ldots, x_m)$, $x_i \in \{-1, 1\}$, or

$$z_k(x) = \sum_{i=1}^{k} x_i, \tag{1}$$

and let $x^{[k]}$ be the word x with its last $m - k$ bits inverted. (Note that the quantity

$z_m(x)$ is the *disparity* of x.) For example, if

$$x = (-1, 1, 1, 1, -1, 1, -1, 1, 1, -1),$$

then the disparity of x equals 2 and

$$x^{[4]} = (-1, 1, 1, 1, 1, -1, 1, -1, -1, 1).$$

If we let $\sigma_k(x)$ stand for the disparity $z_m(x^{[k]})$ of $x^{[k]}$, then the quantity $\sigma_k(x)$ is

$$\begin{aligned} \sigma_k(x) &= \sum_{i=1}^{k} x_i - \sum_{i=k+1}^{m} x_i \\ &= -z_m(x) + 2\sum_{i=1}^{k} x_i. \end{aligned} \quad (2)$$

As an immediate consequence, $\sigma_0(x) = -z_m(x)$, (all symbols inverted) and $\sigma_m(x) = z_m(x)$ (no symbols inverted). If x is of even length m, then we may conclude that every word x can be associated with at least one k for which $\sigma_k(x) = 0$, or $x^{[k]}$ is balanced. The value of the first such k is encoded in a balanced word of length p, p even. (If m and p are both odd, a similar construction is possible.) The maximum codeword length $m + p$ is governed by

$$m \leq \binom{p}{p/2}. \quad (3)$$

Some other modifications of the basic scheme are discussed in [2] and [11].

2.3 Spectrum and sum variance of sequences

Let $\{x_i\}_{i\geq 0}$ denote a cyclo-stationary channel sequence. The power spectral density function of the sequence is given by [16]

$$H(\omega) = R(0) + 2\sum_{i=1}^{\infty} R(i)\cos i\omega, \qquad -\pi \leq \omega \leq \pi, \quad (4)$$

where $R(i) = E\{x_j x_{j+i}\}$, $i = 0, \pm 1, \pm 2, \ldots$, is the auto-correlation function of the sequence. In the sequel it is assumed that $\{x_i\}_{i\geq 0}$ is composed of cascaded codewords x of length n. Let $x^{(k)} = (x_1^{(k)}, \ldots, x_n^{(k)})$, $x_i^{(k)} \in \{-1, 1\}$, be the kth element of a set S of codewords. The Fourier transform $X^{(k)}(\omega)$ of the codeword $x^{(k)}$ is defined by [15]

$$X^{(k)}(\omega) = \sum_{i=1}^{n} x_i^{(k)} e^{-ji\omega}, \qquad -\pi \leq \omega \leq \pi, \quad (5)$$

where $j = \sqrt{-1}$. For all i, $1 \le i \le n$, let the number of codewords $x^{(k)} \in S$ with $x_i^{(k)} = 1$ be equal to the number of codewords with $x_i^{(k)} = -1$ (i.e., the sum of all codewords in S is the all-zero vector). If, in addition, codewords are randomly chosen from S to form an infinite sequence, then it is rather straightforward to show, following [16] and [17], that the power spectral density function $H(\omega)$ of the concatenated sequence is

$$H(\omega) = \frac{1}{n|S|} \sum_{x^{(k)} \in S} |X^{(k)}(\omega)|^2. \tag{6}$$

Note that, due to our assumptions, $|S|$ has to be even. We will assume that the codewords are equiprobable.

From now on, we assume that the set of codewords S forms a *dc-balanced code*, in other words, the disparity of all the members of S is zero. The width of the spectral notch is a relevant quantity since it specifies the frequency region of suppressed components around the spectral null. Let $H(\omega)$ denote the power spectral density function of the sequences formed by cascading the codewords. Then the width of the spectral notch is defined by a quantity called the *cut-off* frequency. The cut-off frequency, denoted by ω_0, is defined by [13], [14]

$$H(\omega_0) = \frac{1}{2}. \tag{7}$$

Another quantity used in the performance evaluation of codes with a null at the zero frequency is the running digital sum [12]. The running digital sum Z_j is defined by

$$Z_j = \sum_{i=0}^{j} x_i. \tag{8}$$

Specifically, the variance of the running digital sum, in short sum variance, plays a key role in the spectral performance characterization of dc-balanced codes. The sum variance of the encoded sequence is defined as

$$s^2 = E\{Z_i^2\}, \tag{9}$$

where $E\{.\}$ denotes the expectation operator. It was found by Justesen [13] that for dc-free sequences the product of cut-off frequency ω_0 and the sum variance of the sequence is approximately $1/2$, or

$$2\omega_0 s^2 \approx 1. \tag{10}$$

It has been found that this relationship between the sum variance and the actual cut-off frequency is accurate within a few percent [12]. On the other hand, the

sum variance is relevant in its own right as it gives the variance of the intersymbol interference when the channel is ac-coupled.

As in the case of $H(\omega)$, it can be shown that, under similar conditions, the sum variance s^2 of the concatenated sequence is [12]

$$s^2 = \frac{1}{n|S|} \sum_{x^{(k)} \in S} \sum_{j=1}^{n} [z_j(x^{(k)})]^2, \tag{11}$$

where $z_j(x) := \sum_{i=1}^{j} x_i$ and $|S|$ denotes the *cardinality* of S. (Observe that $|S| = 2^m$.)

In principle, the above equations can be invoked to compute the spectrum and sum variance of sequences generated by Knuth's method. In that case, S consists of the collection of $(m+p)$-bit codewords obtained from the set of all m-bit source words as described in Section 2.2. Naturally, the above operation of enumerating all codewords is, for large codeword sets, a considerable computational load. (Recall that $|S| = 2^m$.) The computational load can be alleviated by making some approximations. Ignoring the contribution from the p-bit vector (note that Knuth's algorithm is especially designed for large codewords, i.e., $p \ll m$), the sum variance can be approximated as

$$s^2 \approx \frac{1}{n|S|} \sum_{y \in S} \sum_{j=1}^{m} [z_j(y)]^2. \tag{12}$$

This approximation is justified mathematically by the facts that

1) $z_m(y) = 0$, for $y \in S$, and
2) $(1/n)\sum_{j=1}^{p}[z_j(u)]^2 = O(p^3/m) = o(m)$, $m \to \infty$, for a balanced p-bit word u.

In the next section we shall evaluate the right-hand side in (12) exactly.

2.4 A counting problem

In this section we shall provide an exact evaluation of the right-hand side of the approximation for s^2 in (12). We first introduce some useful notation. Throughout this section, m denotes an even, positive integer. The collection of all binary words $x = (x_1, \ldots, x_m)$, $x_i \in \{-1, 1\}$, will be denoted by R_m. For $x \in R_m$, we define $z(x) = (z_0(x), \ldots, z_m(x))$ by

$$z_k(x) := \sum_{i=1}^{k} x_i, \tag{13}$$

$k = 0, \ldots, m$. (Note that $z_0(x) = 0$ by convention. Note also that $z_k(x) \equiv k \bmod 2$.) Put

$$S_m := \{z(x) \mid x \in R_m\}.$$

For each integer k, $-m \leq k \leq m$, we let i

$$R_m(k) := \{x \in R_m \mid z_m(x) = k\}$$

and

$$S_m(k) := \{z \in S_m \mid z_m = k\}.$$

Let $x \in R_m$. The quantity $z_k(x) - (1/2)z_m(x)$ starts at $-(1/2)z_m(x)$ for $k = 0$, ends at $(1/2)z_m(x)$ for $k = m$, and increases by ± 1 if k increases by 1. Since m is even, we conclude that there exists a smallest integer $l = l(x)$, $0 \leq l(x) \leq m - 1$, for which $z_l(x) = (1/2)z_m(x)$. Define $\widehat{x}$ by

$$\widehat{x} := (x_1, \ldots, x_l, -x_{l+1}, \ldots, -x_m), \tag{14}$$

with $l = l(x)$ as defined above. Observe that in Knuth's method a word x is encoded as a word with initial part $\widehat{x}$. Observe also that $\widehat{x} \in R_m(0)$, in other words, $\widehat{x}$ is balanced.

Next, let $z \in S_m$, with $z = z(x)$, say. (Note that x is uniquely determined by z.) By abuse of notation, we set $l(z) := l(x)$, and we let $\widehat{z}$ be defined by $\widehat{z} := z(\widehat{x})$. In other words,

$$\widehat{z}_k := \begin{cases} z_k, & \text{if } 0 \leq k \leq l(z); \\ z_m - z_k, & \text{if } l(z) \leq k \leq m. \end{cases} \tag{15}$$

Note that $\widehat{z} \in S_m(0)$ by definition. Moreover, observe that $\widehat{z}$ can be obtained from z by a reflexion of the part $(z_l, \ldots, z_m)$ of z with respect to the line $y = (1/2)z_m$. (See Figure 1.)

With the above definitions, the approximation (12) can now be expressed as

$$s^2 \approx \frac{1}{n2^m}\lambda(m), \tag{16}$$

where the quantity $\lambda(m)$ is defined as

$$\lambda(m) := \sum_{z \in S_m} \sum_{j=1}^{m} \widehat{z}_j^2. \tag{17}$$

Our aim is to prove the following result.

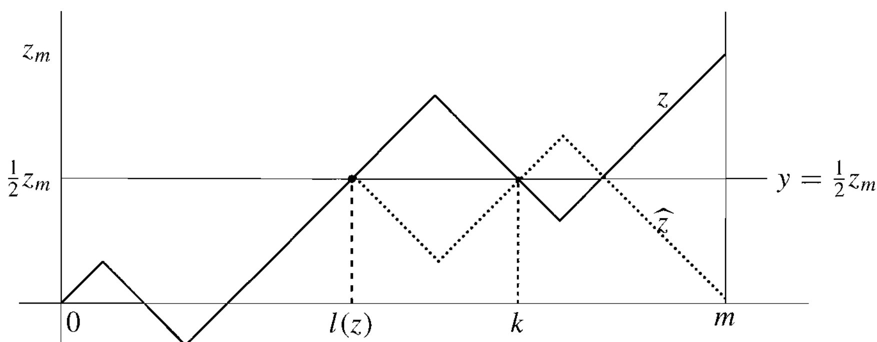

Figure 1: *The relation between z and $\widehat{z}$.*

Theorem 1: $\lambda(m) = m(3m+2)2^{m-4}$.

The proof of the above theorem will depend on a number of lemmas and uses induction on m. (It is easily verified that Theorem (1) indeed holds for small values of m.) In the course of our computations, we will frequently make use of the following two results. Let η_{ij} and ρ_{ij}, $1 \le i, j \le m$, be defined by

$$\eta_{ij} := 2^{-m} \sum_{x \in R_m} x_i x_j, \qquad \rho_{ij} := \binom{m}{m/2}^{-1} \sum_{x \in R_m(0)} x_i x_j. \tag{18}$$

Lemma 2: We have

$$\eta_{ij} = \begin{cases} 1, & \text{if } i = j; \\ 0, & \text{otherwise.} \end{cases} \tag{19}$$

Proof: Evident. □

Lemma 3: We have

$$\rho_{ij} = \begin{cases} 1, & \text{if } i = j; \\ -1/(m-1), & \text{otherwise.} \end{cases} \tag{20}$$

Proof: By symmetry, ρ_{ij} only depends on whether or not $i = j$ holds. Moreover, from the definition of $R_m(0)$ we find that $\sum_{j=1}^{m} \rho_{ij} = 0$. Since it is immediate that $\rho_{ii} = 1$, the result now follows. □

Lemmas 2 and 3 indicate that it is easy to compute a sum over S_m or $S_m(0)$ of polynomial functions of the z_j only (i.e., not involving the $\widehat{z}_j$), simply by writing out the sum in terms of the x_j.

Our next result simplifies the expression (17) for $\lambda(m)$.

Lemma 4: We have

$$\lambda(m) = \sum_{z \in S_m} z_m \sum_{j=1}^{m} \widehat{z}_j.$$

Proof: It will be sufficient to show that

$$\sum_{z \in S_m} (\widehat{z}_j^2 - z_m \widehat{z}_j) = 0 \tag{21}$$

holds, for all j, $1 \leq j \leq m$. To show this, we first observe that

$$\left(\widehat{z}_j - \frac{1}{2} z_m\right)^2 = \left(z_j - \frac{1}{2} z_m\right)^2, \tag{22}$$

(see also Figure 1), and thus

$$\widehat{z}_j^2 - z_m \widehat{z}_j = z_j^2 - z_m z_j. \tag{23}$$

So we are finished if we can show that

$$\sum_{z \in S_m} \left(z_j^2 - z_m z_j\right) = 0. \tag{24}$$

But that is easy, either by using Lemma 2, or by observing that the total contribution to the sum in (24) of $z(x)$ and $z(x')$ is 0, where

$$x' := (x_1, \ldots, x_j, -x_{j+1}, \ldots, -x_m).$$ □

At this point, there are several ways to proceed. In view of Lemma 2, it would be sufficient to derive a closed-form expression for the quantity

$$\sum_{z \in S_m} z_m \sum_{j=1}^{m} (z_j - \widehat{z}_j). \tag{25}$$

(This approach is, in a sense, the most logical way to proceed.) It is indeed possible to evaluate this quantity (using Andre's reflexion principle, see for example [18]) in terms of a complicated double summation involving the product of certain binomial coefficients. Unfortunately, although we knew what the outcome should be we were unable to find a direct proof.

Here, we will follow another approach that we now describe. For each integer $k, 0 \leq k \leq m$, we let

$$S_m^{(k)} := \{z \in S_m \mid z_k = \frac{1}{2} z_m\},$$

and for $z \in S_m^{(k)}$, we define $z^{(k)} = (z_0^{(k)}, \ldots, z_m^{(k)})$ by

$$z_j^{(k)} := \begin{cases} z_j, & \text{if } 0 \leq j \leq k; \\ z_m - z_j, & \text{if } k < j \leq m. \end{cases} \tag{26}$$

Observe that $z^{(k)} \in S_m(0)$ for all $z \in S_m^{(k)}$, and $z^{(k)} = \widehat{z}$ if $k = l(z)$. So we may write

$$\begin{aligned} \lambda(m) &= \sum_{z \in S_m} z_m \sum_{j=1}^{m} \widehat{z}_j \\ &= \sum_{k=0}^{m} \sum_{z \in S_m^{(k)}} z_m \sum_{j=1}^{m} z_j^{(k)} - \sum_{k=0}^{m} \sum_{0 \leq l < k} \sum_{\substack{z \in S_m^{(k)} \\ l(z)=l}} z_m \sum_{j=1}^{m} z_j^{(k)}. \end{aligned} \tag{27}$$

(To see this, note that each $z \in S_m$ contributes $z_m \sum_{j=1}^{m} z_j^{(k)}$ to the first summation in (27) for each k such that $z \in S_m^{(k)}$. By definition of $l(z)$, the smallest such k equals $l(z)$ and gives a contribution equal to $z_m \sum_{j=1}^{m} \widehat{z}_j$, while the contributions for all $k > l$ occur also in the second summation in (27).) We will evaluate the two parts of (27) for $\lambda(m)$ separately.

Lemma 5:

$$\begin{aligned} \sum_{k=0}^{m} \sum_{z \in S_m^{(k)}} z_m \sum_{j=1}^{m} z_j^{(k)} &= 2 \sum_{u \in S_m(0)} \left(\sum_{j=1}^{m} u_j \right)^2 \\ &= \frac{1}{6} m^2 (m+1) \binom{m}{m/2}. \end{aligned}$$

Proof: For fixed k, the map $z \mapsto z^{(k)}$ defines a one-to-one correspondence between $S_m^{(k)}$ and $S_m(0)$. Indeed, note that the inverse of this map is the map

$u \mapsto (u_1, \ldots, u_k, 2u_k - u_{k+1}, \ldots, 2u_k - u_{m-1}, 2u_k)$.

Since $z_m = 2z_k = 2z_k^{(k)}$ for all $z \in S_m^{(k)}$ by definition of $S_m^{(k)}$, the first equation follows. To evaluate the second sum, write

$$\sum_{u \in S_m(0)} \left(\sum_{j=1}^{m} u_j \right)^2 = \sum_{x \in R_m(0)} \left(\sum_{j=1}^{m} \sum_{i=1}^{j} x_i \right)^2 \tag{28}$$

and use Lemma 3, together with the two well-known series

$\sum_{k=0}^{m} k = (1/2)m(m+1)$ and $\sum_{k=0}^{m} k^2 = (1/6)m(m+1)(2m+1)$.

We leave all further details to the reader. □

Lemma 6:

$$\sum_{k=0}^{m} \sum_{0 \le l < k} \sum_{\substack{z \in S_m^{(k)} \\ l(z)=l}} z_m \sum_{j=1}^{m} z_j^{(k)} = \sum_{\substack{n=2 \\ n \equiv 0 \bmod 2}}^{m-2} \binom{n}{n/2} \left\{ \lambda(m-n) + \frac{1}{2} n(m-n) 2^{m-n} \right\}.$$

Proof: Fix k and l, with $0 \le l < k$, and put $n := k - l$. Let $z \in S_m^{(k)}$, with $l(z) = l$. (Observe that this is possible only if n is even.) With each such z, we associate a pair of sequences $a(z)$ and $b(z)$, defined as follows.

$$a_j(z) := \begin{cases} z_j, & \text{if } 0 \le j \le l; \\ z_{j+n}, & \text{if } l < j \le m-n, \end{cases} \tag{29}$$

$$b_j(z) := z_{j+l} - \frac{1}{2} z_m, \qquad 0 \le j \le n. \tag{30}$$

Observe that $b(z)$ represents the part of z between indices l and k, and since $z_k = z_l = (1/2) z_m$ by our assumptions on z, it follows that $b(z) \in S_n(0)$. Observe also that $a(z)$ can be obtained by removing from z the part of z between indices $l + 1$ and k, and thus $a(z) \in S_{m-n}$, $l(a(z)) = l(z) = l$ and $a_{m-n}(z) = z_m$. (See again Figure 1.) From these observations, and from the definition of $z^{(k)}$, we find that

$$z_m \sum_{j=1}^{m} z_j^{(k)} = a_{m-n}(z) \left\{ \sum_{j=1}^{m-n} \widehat{a_j(z)} + \sum_{i=1}^{n} \left[b_i(z) + \frac{1}{2} a_{m-n}(z) \right] \right\}. \tag{31}$$

Conversely, each pair a and b for which $a \in S_{m-n}$, $l(a) = l$ and $b \in S_n(0)$ determines, through (29) and (30), a unique $z \in S_m^{(k)}$ with $l(z) = l$ such that $a(z) = a$ and $b(z) = b$. By this observation, we find from (31) that

$$\begin{aligned} \sum_{\substack{z \in S_m^{(k)} \\ l(z)=l}} z_m \sum_{j=1}^{m} z_j^{(k)} &= \sum_{\substack{a \in S_{m-n} \\ l(a)=l}} \sum_{b \in S_n(0)} a_{m-n} \left\{ \sum_{j=1}^{m-n} \widehat{a_j} + \sum_{i=1}^{n} b_i + \frac{1}{2} n a_{m-n} \right\} \\ &= \binom{n}{n/2} \sum_{\substack{a \in S_{m-n} \\ l(a)=l}} \left\{ a_{m-n} \sum_{j=1}^{m-n} \widehat{a_j} + \frac{1}{2} n a_{m-n}^2 \right\}, \end{aligned} \tag{32}$$

where the last equation follows from the trivial observation that

$$\sum_{b \in S_n(0)} b_i = 0, \tag{33}$$

$1 \leq i \leq n$. (Indeed, $b \in S_n(0)$ if and only if $-b \in S_n(0)$, and b and $-b$ together contribute 0 to the sum in (33).)

Finally, observe that

$$\sum_{a \in S_{m-n}} a_{m-n} \sum_{j=1}^{m-n} \widehat{a_j} = \lambda(m-n)$$

by definition of $\lambda(m-n)$, and

$$\sum_{a \in S_{m-n}} a_{m-n}^2 = (m-n)2^{m-n}.$$

(The last statement can easily be proved using Lemma 2). Then, since $z_m = 0$ whenever $z \in S_m^{(m)}$, the contribution from $k = m$ to the sum in the lemma is 0, so the lemma now follows from (32) by summation over l and n. (Substitute $k = n - l$.) □

If we now combine Lemmas 5 and 6 with the expression for $\lambda(m)$ in (27), and if we write $m = 2M$ and $n = 2k$, we find the following result.

Corollary 7:

$$\sum_{k=0}^{M-1} \binom{2k}{k} \left\{ \lambda(2M-2k) + 2k(M-k)4^{M-k} \right\} = \frac{2}{3} M^2 (2M+1) \binom{2M}{M}.$$

By the induction hypothesis of Theorem 1, we may assume that

$$\lambda(2M-2k) = (2M-2k)(6M-6k+2)4^{M-k-2}, \tag{34}$$

for $k = 1, \ldots, M-1$. Therefore, in order to prove Theorem 1, it remains only to be shown that substitution of the relation (34), for *all* values $k = 0, \ldots, M-1$, into the expression in Corollary 7 results in an identity. This can be shown to be true by use of the identity

$$\sum_{k=0}^{M-1} (k)_e \binom{2k}{k} 4^{M-k-1} = \frac{(M)_{e+1}}{2(2e+1)} \binom{2M}{M}, \tag{35}$$

$e = 0, 1, 2, \ldots$, where $(k)_e := k(k-1)\ldots(k-e+1)$ for $e \geq 1$ and $(k)_0 := 1$, which easily follows from the well-known expansion

$$F(x) := (1-x)^{-1/2} = \sum_{k=0}^{\infty} \binom{2k}{k} \left(\frac{x}{4}\right)^k \tag{36}$$

by calculating $(1-x)^{-1} F^{(e)}(x)$. (The instances $e = 0, 1, 2$ of (35), which we need here, can also be derived for example from [1, vol. 1, page 613, number 19].) Further details are left to the reader.

2.5 Performance Appraisal

We are now in a position to compare the performance of Knuth codes with that of other schemes. Specifically, we will compare the performance of Knuth codes with that of the polarity bit scheme. Both schemes, Knuth codes and polarity bit codes, have the property that they can be encoded and decoded without look-up tables. (To be precise, in the case of Knuth codes the encoding of the value of l into a p-bit balanced word can be effectuated either by using a *small* look-up table or algorithmically by enumerative encoding [19].)

The rate of Knuth codes as a function of p is at most

$$R = \frac{m}{m+p}, \tag{37}$$

where $m = \binom{p}{p/2}$. Let n_p denote the length of a codeword in the polarity bit code. The rate of the polarity bit code is

$$R = 1 - \frac{1}{n_p}. \tag{38}$$

The sum variance of a bit stream generated by the polarity bit format is [12]

$$s_p^2 = \frac{2n_p - 1}{3}. \tag{39}$$

It seems fair to compare the performance of both methods at the same rate. Therefore we choose

$$1 - \frac{1}{n_p} = \frac{m}{m+p}, \tag{40}$$

or

$$n_p = \left\lceil \frac{m+p}{p} \right\rceil, \tag{41}$$

where $\lceil x \rceil$ denotes the smallest integer $k \geq x$. Application of the outcomes of our analysis provides the results shown in Table 1. We may observe that the sum variance of Knuth codes is significantly larger than the sum variance of the polarity bit codes, for approximately the same redundancy.

2.6 Conclusions

We have compared two methods for the generation of dc-balanced sequences. The two methods, Knuth's code and polarity bit code, have the virtue that they can be used without look-up tables. The spectral performance of the two methods has

Table 1: *Rate and sum variance of Knuth codes and polarity bit codes versus codeword length $n = m + p$.*

p	m	$1-R$	s_K^2	n_p	s_p^2
6	20	0.2308	3.875	5	3
8	70	0.1026	13.250	10	6.33
10	252	0.0382	47.375	27	17.67
12	924	0.0128	173.375	78	51.67
14	3432	0.0041	643.625	247	164.33
16	12870	0.0012	2413.250	806	537
18	48620	0.0004	9116.375	2703	1801.67

been evaluated with a parameter called the sum variance. Under the premise that the sum variance can serve as a quantity to judge the width of the spectral notch, we conclude that codes based on Knuth's algorithm offer less spectral suppression than polarity bit codes with the same redundancy.

Acknowledgement

It is a great pleasure to thank our colleagues J.H. van Lint, Sr. and L. Tolhuizen for some fruitful discussions.

References

[1] A.P. Prudnikov, Yu.A. Brychkov, and O.I. Marichev, *Integrals and series*, Gordon and Breach Science Publishers.

[2] D.E. Knuth, "Efficient Balanced Codes", *IEEE Trans. Inform. Theory*, vol. IT-32, pp. 51–53, Jan. 1986. See also P.S. Henry, "Zero disparity coding system", US Patent 4,309,694, Jan. 1982.

[3] J.P.J. Heemskerk and K.A.S. Immink, "Compact Disc: System Aspects and Modulation", *Philips J. Res.*, vol. 40, No. 6, pp. 157–164, 1982.

[4] S. Fukuda, Y. Kojima, Y. Shimpuku, and K. Odaka, "8/10 Modulation Codes for Digital Magnetic Recording", *IEEE Trans. Magnet.*, vol. MAG-22, pp. 1194–1197, Sept. 1986.

[5] K.W. Cattermole, "Principles of Digital Line Coding", *Int. J. Electron.*, vol. 55, pp. 3–33, July 1983.

[6] K.W. Cattermole, *Principles of Pulse Code Modulation*, London: Iliffe Books Ltd, 1969.

[7] J.M. Griffiths, "Binary Code Suitable for Line Transmission", *Electron. Letters*, vol. 5, pp. 79–81, 1969.

[8] F.K. Bowers, US Patent no. 2,957,947, 1960

[9] R.O. Carter, "Low-disparity Binary Coding System", *Electron. Letters*, vol. 1, pp. 65–68, 1965.

[10] G.L. Cariolaro and G.P. Tronca, "Spectra of Block Coded Digital Signals", *IEEE Trans. Commun.*, vol. COM-22, pp. 1555–1563, Oct. 1974.

[11] N. Alon, E.E. Bergmann, D. Coppersmith, and A.M. Odlyzko, "Balancing Sets of Vectors", *IEEE Trans. Inform. Theory*, vol. IT-34, no. 1, pp. 128–130, Jan. 1988.

[12] K.A.S. Immink, "Performance of Simple Binary DC-constrained Codes", *Philips J. Res.*, vol. 40, pp. 1–21, 1985.

[13] J. Justesen, "Information Rate and Power Spectra of Digital Codes", *IEEE Trans. Inform. Theory*, vol. IT-28, pp. 457–472, May 1982.

[14] J.N. Franklin and J.R. Pierce, "Spectra and efficiency of binary codes without dc", *IEEE Trans. Commun.*, vol. COM-20, pp. 1182–1184, Dec. 1972.

[15] E. Gorog, "Redundant Alphabets with Desirable Frequency Spectrum Properties", *IBM J. Res. Develop.*, vol. 12, pp. 234–241, May 1968.

[16] B.S. Bosik, "The Spectral Density of a Coded Digital Signal", *Bell Syst. Tech. J.*, vol. 51, pp. 921–932, Apr. 1972.

[17] G.L. Pierobon, "Codes for Zero Spectral Density at Zero Frequency", *IEEE Trans. Inform. Theory*, vol. IT-30, no. 2, pp. 435–439, Mar. 1984.

[18] W. Feller, *An Introduction to Probability Theory and its Applications, Volume I*, New York: Wiley, 1968.

[19] T. Cover, "Enumerative source coding", *IEEE Trans. Inform. Theory*, vol. IT-19, no. 1, pp. 73–76, Jan. 1973.

Chapter 3

Prefix-synchronized Runlength-limited Sequences

Abstract — **Runlength-limited sequences, or (d, k) sequences, are used in recording systems to increase the efficiency of the channel. Applications are found in digital recording devices such as sophisticated computer disk files and numerous domestic electronics such as stationary- and rotary-head digital audio tape recorders, the Compact Disc and floppy disk drives. In digital recorders, the coded information is commonly grouped in large blocks, called *frames*. Each frame starts with a synchronization pattern, or marker, used by the receiver to identify the frame boundaries and to synchronize the decoder. In most applications, the sync pattern follows the prescribed (d, k) constraints, and it is *unique*, that is, in the encoded sequence, no block of information will agree with the sync pattern except specifically transmitted at the beginnings of the frames. Usually, the above arrangement is termed a *prefix-synchronized format*. The prefix-synchronized format imposes an extra constraint on the encoded sequences, and will therefore result in a loss of capacity. There is an obvious loss in capacity resulting from the fact that the sender is periodically transmitting sync patterns, and there is a further loss in capacity since the sender is not permitted to transmit patterns equal to the sync pattern during normal transmission. The relative reduction in channel capacity associated with the imposition of this additional channel constraint is calculated. Examples of (d, k) codes that permit the prefix-synchronization format, based for the purpose of illustration on the sliding-block coding algorithm, are presented.**

© 1994, IEEE. Reprinted, with permission, from IEEE Jounal on Selected Areas in Communications, vol. 10, no. 1, pp. 214–222, Januari 1994.

Co-authored by K.A. Schouhamer Immink.

This work was presented in part at the 12th Information Theory Symposium in the Benelux, Veldhoven, The Netherlands, May 1991, and the IEEE 1991 International Symposium on Information Theory, Budapest, Hungary, June 24-28, 1991.

3.1 Introduction

Since the early 1970s, coding methods based on (d, k)-constrained sequences have been widely used in such high-capacity storage systems as magnetic and optical disks or tapes. Properties and applications of (d, k)-constrained sequences, or runlength-limited sequences as they are often called, are surveyed in [1] . A binary sequence is said to be (d, k) constrained if the number of "zeros" between pairs of consecutive "ones" lies between d and $k, k > d$. An encoder has the task of translating arbitrary user information into a constrained channel sequence. Commonly, the user data is partitioned into words of length m and under the coding rules, these m-tuples are translated into n-tuples, called codewords. A certain number of codewords constitute a *frame*. All frames are marked with a unique sequence of symbols, referred to as a *marker* or *synchronization pattern*, in short, sync pattern. The frames are chosen so as to assure the non-occurrence of this marker except when specifically transmitted at the beginnings of each frame. A receiver restores the channel bit clock—usually a phase-locked loop is used for this purpose—and subsequently the synchronization pattern is used to identify the frame and codeword boundaries. The sync pattern and the related synchronization circuitry must be designed with great care, as the effect of sync slippage is devastating since entire blocks of information might be lost, which is just as bad as a massive dropout.

Prefix-synchronization codes can be designed by judiciously discarding a number of potential codewords in such a way that the given marker does not occur in a codeword and also does not occur anywhere in the coded sequence as juxtapositions of codewords. Alternatively, potential sync patterns can be found by making use of "by-products" of code implementation. Since the capacity of (d, k)-constrained sequences is irrational (see Ashley and Siegel [2]), it is clear that code implementations which, by necessity, operate at rates of the form m/n, where m and n are integers, can never attain 100% of the capacity. It was noted by Howell [3] that implemented codes differ from maxentropic, "ideal", sequences by the addition of certain constraints, which he called incidental constraints. He found that certain bit patterns, which could readily be used as a basis for constructing sync patterns, are absent in sequences generated by the popular $(1, 7)$ and $(2, 7)$ codes.

The main objections one may have to this simple approach are, first, that the sync pattern is found in a rather heuristic and incidental fashion and, second, we have (generally speaking) no idea how far we are away from a sound engineering compromise between redundancy and efficiency and, specifically, how the choice of a certain sync pattern may affect the complexity of the encoder and decoder. Given that both the choice of the (d, k) parameters and the choice of the sync

pattern involve a careful evaluation, it is clearly appropriate to explore approaches and strategies of a more general, rather than specific, applicability in which the issues of sync pattern selection and code design are beneficially combined.

In 1960, frame synchronization of unconstrained binary sequences in a general setting was addressed by Gilbert [4] (see also Stiffler [5], where a detailed description is given of the synchronization issue). Gilbert showed, among other things, that to each given frame length, there corresponds some optimum length of the sync pattern. Gilbert's work was extended, in 1978, by Guibas and Odlyzko [6], who provided elegant combinatorial arguments to establish closed-form expressions for generating functions. In this paper, we will concentrate on the frame synchronization problem of (d, k) sequences. We commence, in Section 3.2, with a brief description of $(d, k)-$constrained sequences, and proceed with the examination of the channel capacity. It will be shown that for certain sync patterns, called *repetitive-free* sync patterns, the capacity can be formulated in a simple manner as it is solely a function of the (d, k) parameters and the length of the sync pattern. For each forbidden pattern and (d, k) constraints, methods for enumerating constrained sequences are given. In the final sections, we deal with design considerations of schemes for encoding and decoding. Examples of prefix-synchronized (d, k) codes, based for the purpose of illustration on the sliding-block coding algorithm, are presented.

3.2 Preliminaries

Sequences (with a leading "one") that meet prescribed (d, k) constraints may be thought to be composed of *phrases* of length (duration) $j+1, d \leq j \leq k$, denoted by T_{j+1}, of the form 10^j, where 0^j stands for a sequence of j consecutive "zeros". The sync pattern is composed of p phrases, $T_{s_1}, T_{s_2}, \ldots, T_{s_p}$, and since it is assumed that the sync pattern obeys the prescribed (d, k) constraints, we have $s_i \in dk$, where $dk = \{d+1, \ldots, k+1\}$. The p-tuple $\boldsymbol{s} = (s_1, \ldots, s_p)$ is used as a shorthand notation to describe the sync pattern. For example, $\boldsymbol{s} = (2, 1, 3)$ refers to the sync pattern "101100". To keep notation to a minimum, $\boldsymbol{s}$ will represent both the p-tuple $(s_1, \ldots, s_p)$ and the string $10^{s_1-1} \cdots 10^{s_p-1}$, whichever one is referred to should be clear from the context. The length of the sync pattern, $L(\boldsymbol{s})$, is defined by

$$L(\boldsymbol{s}) = \sum_{i=1}^{p} s_i. \tag{1}$$

It should be appreciated that, as a result of the (d, k) constraints in force, the sync pattern is preceded by at least d "zeros", and followed by a "one" and at least d "zeroes". (Of course, unless $s_p = k + 1$, the "one" starting the phrase following

the sync pattern must be a part of the binary pattern used by the synchronization circuitry at the receiver.) It is, therefore, a matter of debate to say that the sync pattern length is $L(\boldsymbol{s})$ or $L(\boldsymbol{s}) + 2d + 1$. The adopted definition (1) is slightly more convenient, as we will see shortly.

Each frame of coded information consists of the prescribed sync pattern $\boldsymbol{s}$ and a string of l cascaded phrases $T_{i_1}, \ldots, T_{i_l}$. The frame is organized according to the format

$$(T_{s_1}, T_{s_2}, \ldots, T_{s_p}, T_{i_1}, T_{i_2}, \ldots, T_{i_l}). \tag{2}$$

The total number of binary digits contained in a frame is prescribed and is denoted by L_{frame}; that is,

$$L_{\text{frame}} = \sum_{j=1}^{p} s_j + \sum_{j=1}^{l} i_j. \tag{3}$$

The valid choices of $T_{i_1}, \ldots, T_{i_l}$ are those for which no p consecutive phrases taken from

$$(T_{s_2}, T_{s_3}, \ldots, T_{s_p}, T_{i_1}, T_{i_2}, \ldots, T_{i_l}, T_{s_1}, T_{s_2}, \ldots, T_{s_{p-1}}) \tag{4}$$

agree with the sync pattern. Dictionaries satisfying this constraint are called *prefix-synchronized runlength-limited codes*. It is relevant to enumerate the number of distinct L_{frame}-tuples given the above conditions. Following this enumeration problem, we will deal with a related problem, namely the computation of the number of constrained sequences when the frame length L_{frame} is allowed to become very large.

3.3 Enumeration of Sequences

In this section, we will address the problem of counting the number of constrained sequences. Our methods can be viewed as an extension of those of Guibas and Odlyzko [6] but can, in fact, be traced back to the work of Schuetzenberger on semaphore codes [7, remark 3], see also Berstel and Perrin, [8, sect. II-7 and notes following sec. VI]. Schuetzenberger attributes these results to Von Mises and Feller [9]. Related work can be found in [10].

First we develop some notation. Let F be the collection of all sequences T of the form

$$T = (T_{i_1}, \ldots, T_{i_n}), \qquad i_j \in dk, \qquad j = 1, \ldots, n, \tag{5}$$

composed of $n > p$ phrases of length

$$L(T) = \sum_{j=1}^{n} i_j, \tag{6}$$

with the properties that

$$\text{F1}): \quad (i_1, \ldots, i_p) = (i_{n-p+1}, \ldots, i_n) = (s_1, \ldots, s_p), \tag{7}$$

$$\text{F2}): \quad (i_j, \ldots, i_{j+p-1}) \neq (s_1, \ldots, s_p), \qquad j = 2, \ldots, n-p. \tag{8}$$

With the collection F, we associate the generating function $f(z)$ defined by

$$f(z) = \sum_{T \in F} z^{-L(T)}. \tag{9}$$

We denote by f_N the coefficient of z^{-N} in $f(z)$. Our aim will be to enumerate the number of distinct binary N-tuples in F, i.e., to determine the numbers f_N. Observe that the number $f_{L_{\text{frame}}+L(s)}$ is nothing but the number of admissible frames. Following [6], we introduce two additional collections of sequences G and H, defined as follows. The collection G consists of all sequences T as in (5) composed of $n \geq p$ phrases such that

$$\text{G1}): \quad (i_1, \ldots, i_p) = (s_1, \ldots, s_p), \tag{10}$$

$$\text{G2}): \quad (i_j, \ldots, i_{j+p-1}) \neq (s_1, \ldots, s_p), \qquad j = 2, \ldots, n-p+1 \tag{11}$$

and H consist of all sequences T as in (5) composed of $n \geq 0$ phrases such that

$$\text{H1}): \quad (i_j, \ldots, i_{j+p-1}) \neq (s_1, \ldots, s_p), \qquad j = 1, \ldots, n-p+1. \tag{12}$$

Note that by definition the empty sequence Λ is contained in H, $\boldsymbol{s} \in G$, and the three collections F, G, and H are mutually disjoint. With these collections G and H, we associate generating functions $g(z)$ and $h(z)$ defined as

$$g(z) = \sum_{T \in G} z^{-L(T)}; \qquad h(z) = \sum_{T \in H} z^{-L(T)}. \tag{13}$$

We now proceed to determine $f(z)$, $g(z)$, and $h(z)$. The idea is to derive relations between these generating functions from certain combinatorial relations between the collections F, G, and H. To start with, we observe the following. Let $T = (T_{i_1}, \ldots, T_{i_n})$ be a sequence in G. Then the sequence $T * (T_i) = (T_{i_1}, \ldots, T_{i_n}, T_i)$, $i \in dk$, is contained in F or in $G \setminus \{\boldsymbol{s}\}$, but not in both since F and G are disjoint. On the other hand, if $T = (T_{i_1}, \ldots, T_{i_n})$, $n \geq p+1$, is contained in F or in $G \setminus \{\boldsymbol{s}\}$, then the sequence $(T_{i_1}, \ldots, T_{i_{n-1}})$ is contained in G. We conclude that there is a one-to-one correspondence between sequences

$$T * (T_i), \qquad T \in G, \qquad i \in dk,$$

and sequences in

$$F \cup G \setminus \{\boldsymbol{s}\}.$$

From this observation, the following lemma is immediate.

Lemma 1:

$$g(z)P_{dk}(z) = f(z) + g(z) - z^{-L(s)},$$

where

$$P_{dk}(z) = \sum_{i \in dk} z^{-L(T_i)} = \sum_{i \in dk} z^{-i}. \tag{14}$$

Proof: We have

$$\begin{aligned} g(z)P_{dk}(z) &= \sum_{T \in G} z^{-L(T)} \cdot \sum_{i \in dk} z^{-L(T_i)} \\ &= \sum_{T \in G} \sum_{i \in dk} z^{-L(T*(T_i))} \\ &= \sum_{\widehat{T} \in F \cup G, \widehat{T} \neq s} z^{-L(\widehat{T})} \\ &= f(z) + g(z) - z^{-L(s)}. \end{aligned}$$

□

Similarly, we may derive a one-to-one correspondence between sequences

$$(T_i) * T, \qquad T \in H, \qquad i \in dk,$$

and sequences in

$$H \cup G \setminus \{\Lambda\},$$

(recall that G and H are disjoint), which leads to the next lemma.

Lemma 2:

$$P_{dk}(z)h(z) = h(z) + g(z) - 1.$$

Proof: Similar to the proof of Lemma 1. (Note that $z^{-L(\Lambda)} = z^0 = 1$). □

Before we can write down the last relationship we require some definitions.

Define the $(p-i)$-tuples $\boldsymbol{h}^{(i)} = (s_1, \ldots, s_{p-i})$ and $\boldsymbol{t}^{(i)} = (s_{i+1}, \ldots, s_p)$, so $\boldsymbol{h}^{(i)}$ and $\boldsymbol{t}^{(i)}$ consist of the $p-i$, $i = 0, \ldots, p-1$, first and last phrases of the marker $\boldsymbol{s}$, respectively. Let $L(\boldsymbol{h}^{(i)})$ be the length of the first $p-i$ phrases of $\boldsymbol{s}$, or

$$L(\boldsymbol{h}^{(i)}) = \sum_{j=1}^{p-i} s_j. \tag{15}$$

The *autocorrelation function* of $\boldsymbol{s}$, denoted by $\boldsymbol{r}$, is a binary vector of length p which is defined by $r_i = 0$ if $\boldsymbol{h}^{(i)} \neq \boldsymbol{t}^{(i)}$ and $r_i = 1$ if $\boldsymbol{h}^{(i)} = \boldsymbol{t}^{(i)}$. Obviously, $r_0 = 1$. An example may serve to explain why $\boldsymbol{r}$ is termed auto-correlation function.

i						r_i
	1	2	1	2	1	
1		1	2	1	2	0
2			1	2	1	1
3				1	2	0
4					1	1

Figure 1: *Process of forming the auto-correlation function* $\mathbf{r}$.

Let $\boldsymbol{s} = (1, 2, 1, 2, 1)$, then Figure 1 exemplifies the process of forming the auto-correlation function. If a marker tail coincides with a marker head, we have $r_i = 1$; else $r_i = 0$.

If $r_i = 0$, $1 \leq i \leq p-1$, that is, if no proper tail equals a proper head of the marker, we say that the marker is *repetitive free*.

We are now in the position to prepare Lemma 3. To that end, let i, $0 \leq i \leq p-1$, be such that $r_i = 1$, and let $T = (T_{i_1}, \ldots, T_{i_n}) \in G$. By definition of G, $i_j = s_j$ for $j = 1, \ldots, p$. Since $r_i = 1$, we have

$$
\begin{aligned}
(T_{s_1}, \quad \ldots \quad &, T_{s_i}) * T \\
&= (T_{s_1}, \ldots, T_{s_i}, T_{s_1}, \ldots, T_{s_p}, T_{i_{p+1}}, \ldots, T_{i_n}) \\
&= (T_{s_1}, \ldots, T_{s_i}, T_{s_{i+1}}, \ldots, T_{s_p}, T_{s_{p-i+1}}, \ldots, T_{s_p}, T_{i_{p+1}}, \ldots, T_{i_n}) \\
&= \boldsymbol{s} * \widehat{T}
\end{aligned}
$$

where $\widehat{T} = (T_{s_{p-i+1}}, \ldots, T_{s_p}, T_{i_{p+1}}, \ldots, T_n)$. Moreover, $\widehat{T}$ is a proper suffix of T, hence $\widehat{T} \in H$.

Conversely, let $\widehat{T} = (T_{i_1}, \ldots, T_{i_n}) \in H$. Consider the word $U = \boldsymbol{s} * \widehat{T} = (U_{i_1}, \ldots, U_{i_{n+p}})$. Let j be the largest number such that $(U_{i_j}, \ldots, U_{i_{j+p-1}}) = \boldsymbol{s}$. Note that, since $\widehat{T} \in H$, $1 \leq j \leq p$ holds. From the way in which j was defined, it follows that $T := (U_{i_j}, \ldots, U_{i_{n+p}}) \in G$ and moreover that $i := j - 1$ satisfies $r_i = 1$.

From the above we conclude that there exists a one-to-one correspondence between sequences

$$(T_{s_1}, \ldots, T_{s_i}) * T, \qquad 0 \leq i \leq p-1, \qquad r_i = 1, \qquad T \in G$$

and sequences

$$\boldsymbol{s} * \widehat{T}, \qquad \widehat{T} \in H.$$

From this observation, the following lemma easily follows.

Lemma 3:

$$(1 + F_r(z))g(z) = z^{-L(\boldsymbol{s})}h(z),$$

where the polynomial $F_r(z)$ is defined as

$$F_r(z) = \sum_{i=1}^{p-1} r_i z^{L(\boldsymbol{h}^{(i)})-L(\boldsymbol{s})}. \tag{16}$$

Proof: If $r_i = 1$, then $\boldsymbol{h}^{(i)} = \boldsymbol{t}^{(i)} = (s_{i+1}, \ldots, s_p)$, whence

$$L(\boldsymbol{h}^{(i)}) - L(\boldsymbol{s}) = -L(s_1, \ldots, s_i). \tag{17}$$

Note also that

$$r_0 z^{L(\boldsymbol{h}^{(0)})-L(\boldsymbol{s})} = 1. \tag{18}$$

Therefore,

$$\begin{aligned}
(1 + F_r(z))g(z) &= \left(\sum_{0 \le i \le p-1} z^{-L(s_1,\ldots,s_i)} \right) \sum_{T \in G} z^{-L(T)} \\
&= \sum_{\substack{0 \le i \le p-1 \\ r_i = 1}} \sum_{T \in G} z^{-L((s_1,\ldots,s_i)*T)} \\
&= \sum_{\widehat{T} \in H} z^{-L(\boldsymbol{s}*\widehat{T})} \\
&= z^{-L(\boldsymbol{s})}h(z).
\end{aligned}$$

□

From the relations between $f(z)$, $g(z)$, and $h(z)$ as described in Lemmas 1–3, we may derive the following result.

Theorem 4:

$$z^{L(\boldsymbol{s})} f(z) = \frac{F_r(z)(P_{dk}(z) - 1) - z^{-L(\boldsymbol{s})}}{P_{dk}(z) - 1 - z^{-L(\boldsymbol{s})} - (1 - P_{dk}(z))F_r(z)}. \tag{19}$$

Proof: From Lemmas 2 and 3, an expression for $g(z)$ not involving $h(z)$ may be derived. If this expression for $g(z)$ is combined with Lemma 1, the theorem follows easily. □

Corollary 5: The number of admissible frames of length $N = L_{\text{frame}}$ is the coefficient of z^{-N} in the power series expression of the RHS of (19) .

It is immediate from Theorem 4 that the number of constrained sequences is solely a function of the marker length $L(\boldsymbol{s})$ if the marker is repetitive free. Although Theorem 4 is useful for enumerating the number of constrained sequences,

it shows its greatest utility in investigating the asymptotic behaviour of the number of constrained sequences when the sequence length is allowed to become very large.

This asymptotic behaviour is directly related to the (noiseless) capacity to be discussed in the ensuing section.

3.3.1 Capacity

The number of distinct (d, k) sequences of length n, denoted by $N_{dk}(n)$, grows, for sufficiently large n, exponentially:

$$N_{dk}(n) \simeq a_1 2^{C(d,k)n}, \tag{20}$$

where a_1 is a constant independent of n, and the rate of growth $C(d, k)$ is termed the (noiseless) *capacity* of the (d, k)-constrained channel. The capacity $C(d, k)$ is the base-2 logarithm of the largest real root of the characteristic equation [11]

$$P_{dk}(z) = \sum_{i \in dk} z^{-i} = 1. \tag{21}$$

For the d-constrained case, $k = \infty$, we have the characteristic equation

$$P_{d,\infty}(z) = \sum_{i=d+1}^{\infty} z^{-i} = \frac{z^{-d}}{z-1} = 1. \tag{22}$$

Using the above characteristic equations, it is not difficult to compute the capacity of (d, k)-constrained channels. The results of the computations have been tabulated in [1] .

The capacity $C(d, k, \boldsymbol{s})$ of (d, k) sequences where the marker $\boldsymbol{s}$ is forbidden can be found from the next theorem.

Theorem 6: $C(d, k, \boldsymbol{s}) = \log_2 \lambda$, where λ is the largest real root of

$$P_{dk}(z) - z^{-L(\boldsymbol{s})} - (1 - P_{dk}(z))F_r(z) = 1. \tag{23}$$

Proof: Follows from Theorem 4. Note that numerator and denominator of the RHS of (19) have no common factors. □

An upper and lower bound to the capacity $C(d, k, \boldsymbol{s})$ are given in the next corollary.

Corollary 7: For given sync pattern length $L(\boldsymbol{s})$,

$$\log_2 \lambda_l \leq C(d, k, \boldsymbol{s}) \leq \log_2 \lambda_u,$$

where λ_l is the largest real root of

$$P_{dk}(z) - z^{-L(s)} = 1$$

and λ_u is the largest real root of

$$P_{dk}(z) - z^{-L(s)} - (1 - P_{dk}(z)) \sum_{i=1}^{p-1} z^{-i(d+1)} = 1.$$

The lower bound is attained if s is repetitive free and the upper bound is attained if s is of the form $(d+1, \ldots, d+1)$.

Proof: Let

$$q(z) := P_{dk}(z) - 1 - z^{-L(s)} - (1 - P_{dk}(z))F_r(z),$$

$$a(z) := P_{dk}(z) - 1 - z^{-L(s)},$$

and

$$b(z) := P_{dk}(z) - 1 - z^{-L(s)} - (1 - P_{dk}(z)) \sum_{i=1}^{p-1} z^{-i(d+1)}.$$

We start with the lower bound. Since $a(\lambda_l) = 0$, we have

$$q(\lambda_l) = -(1 - P_{dk}(\lambda_l))F_r(\lambda_l) = \lambda_l^{-L(s)} F_r(\lambda_l) \geq 0.$$

Since $q(z) = -1$ for $z \to \infty$, and since λ is the largest real zero of $q(z)$, this implies $\lambda_l \leq \lambda$. Equality holds if $r = (1, 0, \ldots, 0)$, that is, if the sync pattern s is repetitive free.

The upper bound follows from a similar argument. Since $1 \leq \lambda_l \leq \lambda$, we have $\lambda^{-1} \leq 1$. Moreover, $L(s_1, \ldots, s_i) \geq i(d+1)$, hence

$$F_r(\lambda) \leq \sum_{i=1}^{p-1} \lambda^{-L(s_1,\ldots,s_i)} \leq \sum_{i=1}^{p-1} \lambda^{-i(d+1)}.$$

From $q(\lambda) = 0$, it follows that

$$(1 - P_{dk}(\lambda))(1 + F_r(\lambda)) = -\lambda^{-L(s)} < 0,$$

whence $1 - P_{dk}(\lambda) < 0$. Therefore,

$$b(\lambda) = (1 - P_{dk}(\lambda))(F_r(\lambda) - \sum_{i=1}^{p-1} \lambda^{-i(d+1)}) \geq 0.$$

Table 1: *Capacity of the* $(1,3)$*-constrained channel.*

s	$C(1,3,s)$	Σv_i	$\max\{v_i\}$
1000	.40569	-	-
1010	.46496	-	-
10100	.45316	-	-
100010	.48673	-	-
100100	.50630	18	4
101010	.51737	30	6
1000100	.50902	11	3
1001010	.50902	20	4
1010010	.51606	33	6
10001000	.52934	11	3
10001010	.52352	12	3
10010010	.52352	18	4
10010100	.52785	26	4
10100010	.52669	32	6
10101010	.53691	30	6

Since $b(z) = -1$ for $z \to \infty$, and since λ_u is the largest real zero of $b(z)$, this implies $\lambda \leq \lambda_u$. □

Tables 1–3 present $C(d,k,s)$ for selected (d,k) parameters. (The right-hand columns are related to the code design to be discussed in a later section.) From Table 1 we observe, for example, that $s = (3, 3)$ is the shortest sync pattern that admits of a code with rate $1:2$, while the shortest repetitive-free sync patterns that admit of the same code rate are $s = (4, 3)$ and $(3, 2, 2)$ (and, of course, their reversed counterparts, which are not in the table). In the Appendix, an analysis is given to approximate $C(d,k,s)$. In the next section, we will discuss what the consequences are in terms of encoder and decoder complexity.

3.4 Code Design

In this section, we use the preceding theory to provide some examples of code design based on the sliding-block code algorithm [12] . The starting point of the sliding-block code algorithm is a description of the channel constraints by a labeled directed graph. An N_s-state (labeled) graph is characterized by an $N_s \times N_s$ adjacency matrix D with non-negative integer entries and a labeling output function ζ that maps the transitions (edges) into the channel alphabet. In the next section, we will describe the graph that represents the (d,k) and synchronization

Table 2: *Capacity of the* $(1, 7)$*-constrained channel.*

s	$C(1,7,s)$	Σv_i	$\max\{v_i\}$
1000000	.66262	-	-
1000010	.66262	-	-
1000100	.66262	-	-
1001010	.66262	-	-
1010010	.66422	-	-
10000000	.66903	99	19
10000010	.66903	129	19
10000100	.66903	147	19
10001000	.67047	164	19
10001010	.66903	160	19
10010010	.66903	275	30
10010100	.66998	252	27
10100010	.66965	279	27
10101010	.67302	142	15
100000010	.67296	65	11
100000100	.67296	76	11
100001000	.67296	88	11
100001010	.67296	86	11
100010010	.67296	102	11
100010100	.67296	102	11
100100010	.67296	156	16
100100100	.67448	153	16
100101010	.67296	152	16
101000010	.67320	198	19
101001010	.67320	202	19

constraints. Thereafter, as a straightforward application, we present some examples of code design.

3.4.1 Graph description

We will now proceed to show that the constraints can be represented by an $(L + k + 1 - s_1)$-state graph. Admissible sequences emitted by the graph exclude the sync pattern as well as those sequences which violate the (d, k) constraints. Let the $(L + k + 1 - s_1)$ states be denoted by $\sigma_1, \ldots, \sigma_L, \hat{\sigma}_1, \ldots, \hat{\sigma}_{k+1-s_1}$. The graph occupies state σ_i, $1 \leq i \leq L$, if it is possible to form a marker by extending the generated sequence with $L - i + 1$ symbols, or stated alternatively, if the i

Table 3: *Capacity of the* $(2,7)$*-constrained channel.*

s	C(2,7,s)	Σv_i	$\max\{v_i\}$
1000000	.48913	-	-
1000100	.48913	-	-
10000000	.49791	-	-
10000100	.49791	-	-
10001000	.50202	160	20
100000100	.50395	67	10
100001000	.50395	108	14
100100100	.50841	124	14
1000000100	.50811	61	10
1000001000	.50811	70	10
1000010000	.50952	80	10
1000100100	.50811	85	10
1001000100	.50887	134	14
10000000100	.51097	23	4
10000001000	.51097	39	6
10000010000	.51097	62	8
10000100100	.51097	79	10
10001000100	.51097	96	10
10001001000	.51148	86	10
10010000100	.51134	138	14

most recently emitted symbols equal the i first symbols of the marker (and if i is maximal with this property). The graph is in the state $\hat{\sigma}_i$, $1 \leq i \leq k+1-s_1$, if, to form a marker, the generated sequence must be extended with $L+1$ symbols and if the most recently emitted "zero" runlength is $s_1 - 1 + i$. The structure of the graph can be described as follows. Let $\boldsymbol{m} = (m_1, \ldots, m_p)$ denote the p-tuple formed by the p most recent phrases generated by the graph. Evidently, $m_i \in dk$, $i \leq p-1$, and $m_p \in \{1, \ldots, k+1\}$. Let l be the *largest* integer for which $1 \leq l \leq p(s_1, \ldots, s_{l-1}) = (m_{p-l+1}, \ldots, m_{p-1})$ and $m_p \leq s_l$ hold. (The first condition is considered to be vacuously true if $l = 1$). If no such integer can be found, then set $l = 0$. Then, if $l > 0$, the graph is in the state

$$\sigma_j, \quad j = \sum_{i=p-l+1}^{p} m_i,$$

else it is in the state $\hat{\sigma}_j$, $j = m_p - s_1$. The various transitions between the states and the output labeling function are implied by the definition of the states and

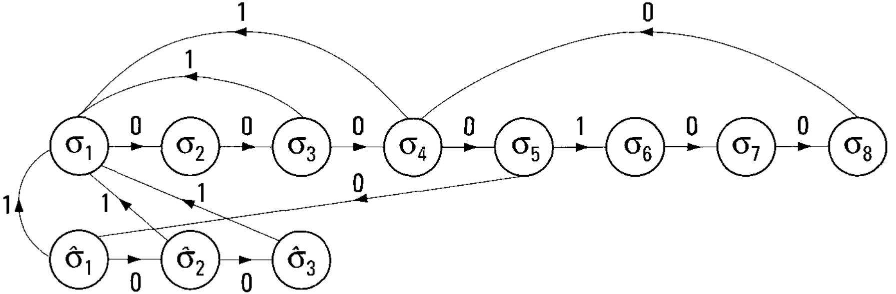

Figure 2: *Eleven-state source that describes the (2,7) constraints and does not generate the pattern "10000100(1)". Note that the last "one", written in parentheses, indicates the start of a new phrase.*

the constraints in force. Figure 2, for example, depicts an 11-state graph that represents the (2, 7) constraints and does not generate the marker $\boldsymbol{s} = (5, 3)$.

3.4.2 Results

The graph described in the previous section serves as the input to the sliding-block algorithm. The sliding-block code design is guided by the *approximate eigenvector inequality*. Let the code rate be $m/n < C(d, k)$, where m and n, $m < n$, are positive integers. An approximate eigenvector $\boldsymbol{v}$ is a non-negative integer vector satisfying

$$D^n \boldsymbol{v} \geq 2^m \boldsymbol{v}. \tag{24}$$

If λ is the largest positive eigenvalue of the adjacency matrix D, then by the Perron-Frobenius theory, (see e.g., Varga [13]) there exists a vector $\boldsymbol{v}$ whose entries v_i are non-negative integers satisfying (24), where $n/m \leq \log_2 \lambda$. The following algorithm, taken from Franaszek [14], (see also Adler *et al.* [12]) is an approach to finding such a vector.

Choose an initial vector $\boldsymbol{v}^{(0)}$ whose entries are $v_i^{(0)} = \beta$, where β is a nonnegative integer. Define inductively

$$v_i^{(u+1)} \equiv \min\left(v_i^{(u)}, \text{ entier}\left(2^{-m} \sum_{j=1}^{N_s} [D^n]_{ij} v_j^{(u)}\right)\right), \tag{25}$$

where entier(x) denotes the largest integer less or equal x. Let

$$\boldsymbol{v} \equiv \boldsymbol{v}^{(u)},$$

where u is the first integer such that $\boldsymbol{v}^{(u+1)} = \boldsymbol{v}^{(u)}$. There are two situations: (a) $\boldsymbol{v} > 0$ and (b) $\boldsymbol{v} = 0$. Case a) means that we have found an approximate

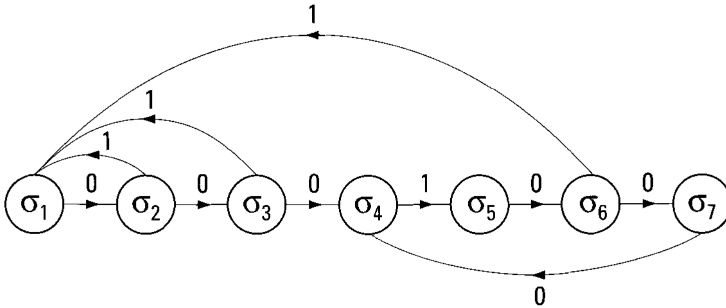

Figure 3: *Seven-state source that describes the (1,3) constraints and does not generate a pattern "1000100(1)".*

eigenvector, and in case b) there is no solution, so we increase β and start from the top again. There may be multiple solutions for the vector v. The choice of the vector may affect some significant parameters of the code so constructed. The largest component of v is related to the error propagation in the decoding process, and the parameter Σv_i is a factor that has an influence on the encoder complexity. After finding an approximate eigenvector using (25), it is sometimes possible to find a better eigenvector by a systematic trial and error method. Adler *et al.* [12] suggested reducing one of the components of v and applying (25) again.

At least in principle, it is now possible to run a computer program in order to evaluate the suitability of a certain sync pattern and to assess the concomitant complexity of the encoder and decoder hardware. When the sync pattern is long, however, the evaluation of each valid sync pattern quite easily becomes an engineering impossibility. We opted for a slightly different approach. A rough estimate of the encoder and decoder complexity can be based on two simple indicators: a) Σv_i and b) $\max\{v_i\}$. It has been shown by Adler *et al.* [12] that the size of the encoder is upper bounded by Σv_i and the error propagation properties are related to the number of rounds in the splitting process which is, in turn, connected to $\max\{v_i\}$. Results of computations are collected in Tables 1–3 for rate-1/2, (1, 3), rate-2/3, (1, 7), and rate-1/2, (2, 7) codes, respectively. In general terms, the figures reflect our intuition that as the capacity of the constrained channel increases, it becomes easier to implement the code. Some outcomes, however, are intriguing as they contradict this general rule. Specifically, we observe that both parameters Σv_i and $\max\{v_i\}$ suggest that the rate 1/2, (1, 3) code with sync pattern (3, 3) is easier to implement than the same code with the sync pattern (2, 3, 2), while one might expect otherwise as the capacity of the latter is significantly greater. It is not at all obvious whether this is an artifact of the indicators or whether other parameters play a role. Further study, for example using the tools

Table 4: *Code book of rate* $1/2$, $(1,3)$ *code.*

σ_i	$g(\sigma_i,0)$	$h(\sigma_i,0)$	$g(\sigma_i,1)$	$h(\sigma_i,1)$
1	1	10	4	10
2	1	00	5	00
3	2	01	3	01
4	1	00	3	10
5	3	01	3	10

presented by Marcus and Roth [15], is needed to answer these questions. In the next section, we will describe a worked example of the design of a rate 1/2, (1, 3) code.

3.4.3 Worked example

We consider the design of a rate $1/2$, $(1,3)$ code which does not generate the marker $s = (4,3)$ or "1000100(1)". The capacity of the channel is (see Table 1) $C(1,3,s=(4,3)) = 0.50902$, so that this code achieves an efficiency of $0.5/0.50902 = 98.2\%$. Figure 3 shows the graphical representation of a seven-state source whose output sequences obey the given constraints.

The adjacency matrix D associated with this source is

$$D = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}. \tag{26}$$

Invoking (25), we find the approximate eigenvector $\boldsymbol{v}^{\mathrm{T}} = (2,3,2,1,1,2,0)$. After the previous spadework, it is now a simple matter to assemble, using the sliding-block coding algorithm, a rate $1/2$, $(1,3)$ sliding-block encoder. The encoder is defined by two characterizing functions: the next-state function $g(\sigma_i, \bar{\beta}_u)$ and the output function $h(\sigma_i, \bar{\beta}_u)$, where $\bar{\beta}_u$, $u = 1,\ldots,2^m$, are the 2^m source words, and σ_i, $i = 1,\ldots,N_e$, are the N_e encoder states. The output and next-state function are shown in Table 4.

The implementation of the rate $1/2$, $(1,3)$ encoder is elementary; the encoder comprises a three-stage register to store the actual state (three bits are sufficient to represent the five encoder states), and a $(1+3) \longrightarrow (2+3)$ logic array for looking up the 2-b codeword and the 3-b next-state. Decoding can be accomplished by a

Table 5: *Coding rules MFM code.*

Source	*Output*
0	$x0$
1	01

sliding-window decoder with a window length of eight channel bits, thus limiting error propagation to at most four decoded user bits. No attempt has been made to improve this parameter.

3.4.4 Incidental constraints

Runlength-limited codes operate at rates of the form m/n, where m and n are integers. It was shown by Ashley and Siegel [2] that, save a trivial exception, the capacity of (d, k)-constrained sequences is irrational. It is, therefore, clear that code implementations can never attain 100% of the capacity. It was noted by Howell [3] that implemented codes differ from maxentropic, "ideal", sequences by the addition of a few constraints, which he called incidental constraints. Howell [3] described in detail the incidental constraints of three popular (d, k) codes, namely the rate 2/3, (1, 7) code [16], the rate 1/2, (2, 7) code [17], and the rate 1/2, (1, 3) code. He found that certain bit patterns do not occur in sequences generated by these codes. The codes above exhibit other incidental constraints. They are, however, not relevant for synchronization purposes and are therefore not discussed here. An important beneficial consequence of the presence of incidental constraints is that prefix-synchronization can often be established by exploiting the incidental constraints so that information capacity is not wastefully dissipated. It is an instructive exercise to compare the three above codes, where we could base the sync pattern on the incidental constraints, with codes found with the more direct approach outlined in the theory developed.

1) *Rate* 1/2, (1, 3) *Code:* Modified Frequency Modulation (MFM), a rate 1/2, (1, 3) code, has proved very popular from the viewpoint of simplicity and ease of implementation, and has become a *de facto* industry standard in flexible and "Winchester"-technology disk drives. MFM is essentially a state-dependent block code with codewords of length $n = 2$. The encoding rules underlying the MFM code are shown in Table 5. The symbol indicated with "x" is set to "zero" if the preceding symbol is "one"; else it is set to "one". It can be verified that MFM sequences have a minimum and maximum runlength of $d = 1$ and $k = 3$, respectively.

These rules are easily translated into a two-state encoder. Decoding of the MFM code is simply accomplished by discarding the redundant first bit in each

received 2-b block. It is not difficult to check that under MFM rules, the output sequence "10 00" is not valid, but "01 00 01", however, is valid, so that "1000" cannot serve as a sync pattern. The shortest sequence that does not occur in an MFM sequence is the 12-b sequence "100010010001". Perusal of Table 1 reveals that the shortest markers giving room for the design of a rate $1/2$ code are of length six. The rate $1/2$, $(1, 3)$ code described in Section 3.4.3 has the advantage that its sync word is 8-b long instead of twelve bits required in the MFM code. On the other hand, the MFM code has a simpler implementation and the decoder window length is shorter, namely two bits, and is thus more favorable in terms of error propagation.

2) *Rate* $1/2$, $(2, 7)$ *Code:* A great many kinds of rate-$1/2$ codes are known for the parameters $d = 2$ and $k = 7$. We will concentrate on the variable-length variant invented by Franaszek [17] and modified by Eggenberger and Hodges [18]. The code can be encoded with a six-state encoder (Howell [3] showed how to devise a five-state encoder) and decoded with a sliding-block decoder of length eight. Howell [3] found that this (2, 7) code does not produce 12-b sequences of the form $10^7 10^2 1$, so that the sequence $s = (8, 3)$ can be used as a marker in this scheme. In Table 3, we observe that there are nine candidate sync patterns s of length $L(s) < 11$ which permit the construction of a rate $1/2$ code; the shortest marker has length eight. Rate-$1/2$ codes which embed each of these candidate markers were generated using the sliding-block coding algorithm. Approximately 10 rounds of splitting steps are required for the codes with the markers of length 8 and 9; the corresponding encoders utilize approximately 30 states. The codes for the markers of length 10 require 8 splitting rounds. The simplest code found uses the marker $s = (7, 3)$; the code requires 23 encoder states and can be decoded with a decoder window of 23 bits.

3) *Rate* $2/3$, $(1, 7)$ *Code:* The rate $2/3$, $(1, 7)$ code described by Jacoby and Kost [16] is a lookahead code. It can be encoded by a five-state encoder and decoded by a decoder with a window of seven bits. Howell [3] has observed that the set of sequences generated by this code is invariant under time reversal. The 16-b pattern $10^6 10^7 1$ and its reverse $10^7 10^6 1$ do not occur in sequences generated by the $(1, 7)$ code. We have evaluated a number of shorter sync patterns, but could not find a code that rivaled the Jacoby/Kost code in complexity.

3.5 Conclusions

We have investigated the prefix-synchronization of runlength-limited sequences. In particular, we have analyzed the capacity of (d, k)-constrained channels giving room for the prefix-synchronization format. It has been shown that for certain

sync patterns, termed repetitive-free sync patterns, the capacity can be formulated in a simple manner, as it is solely a function of the (d, k) parameters and the length of the sync pattern. For presentation purposes, the complexity consequences of short sync patterns have been investigated for the rate-1/2 codes with parameters $(1, 3)$ and $(2, 7)$, and for the rate 2/3, $(1, 7)$ code. We have not found a code that rivaled the $(1, 7)$ and $(2, 7)$ codes whose sync patterns are based on their respective incidental constraints. A worked example of a rate 1/2, $(1, 3)$ code giving room for the inclusion of a unique sync pattern of length seven has been presented.

Appendix: Approximations to the capacity bounds

In this Appendix, we derive approximations to the lower and upper bound to the capacity $C(d, k, \boldsymbol{s})$. Corollary 7 asserts that for given marker length $L(\boldsymbol{s})$, $\log_2 \lambda_l \le C(d, k, \boldsymbol{s}) \le \log_2 \lambda_u$, where λ_l is the largest real root of

$$P_{dk}(z) - z^{-L(\boldsymbol{s})} = 1$$

and λ_u is the largest real root of

$$P_{dk}(z) - z^{-L(\boldsymbol{s})} - (1 - P_{dk}(z)) \sum_{i=1}^{p-1} z^{-i(d+1)} = 1.$$

We commence with the lower bound. To that end, let μ be the largest real root of the characteristic equation

$$P_{dk}(z) = \sum_{i \in dk} z^{-i} = 1$$

and let $\lambda_l = \mu + \Delta\lambda_l$ be the largest real root of

$$P_{dk}(z) - z^{-L(\boldsymbol{s})} = 1.$$

Then

$$P_{dk}(\mu + \Delta\lambda_l) - (\mu + \Delta\lambda_l)^{-L(\boldsymbol{s})} = 1.$$

Differentiating and working out yields

$$\Delta\lambda_l \simeq -\frac{\mu}{\overline{T}} \mu^{-L(\boldsymbol{s})}, \qquad L(\boldsymbol{s}) \gg 1,$$

where the average runlength $\overline{T}$ is

$$\overline{T} = \sum_{i \in dk} i \mu^{-i}.$$

Thus,

$$\begin{aligned} C(d,k,s) &> \log_2(\mu + \Delta\lambda_l) \\ &= C(d,k) + \log_2(1 + \Delta\lambda_l/\mu) \\ &\simeq C(d,k) - \frac{1}{ln(2)\overline{T}}\,\mu^{-L(s)}. \end{aligned}$$

In a similar fashion, we find for the upper bound $\lambda_u = \mu + \Delta\lambda_u$:

$$\Delta\lambda_u \simeq -\frac{\mu}{\overline{T}}\,\frac{\mu^{-L(s)}}{1 + \sum_{i=1}^{p-1} \mu^{-i(d+1)}}.$$

We have

$$1 + \sum_{i=1}^{p-1} \mu^{-i(d+1)} \simeq \frac{1}{1 - \mu^{-(d+1)}}, \qquad L(\mathbf{s}) \gg 1,$$

and from (22) we infer that, for $k \gg 1$,

$$\mu^{-(d+1)} \simeq 1 - \mu^{-1},$$

so that

$$\Delta\lambda_u \simeq -\frac{1}{\overline{T}}\,\mu^{-L(s)}, \qquad L(s) \gg 1, \quad k \gg 1.$$

Thus, for the relative capacity loss we find the following approximations:

$$\frac{1}{ln(2)\overline{T}}\,\mu^{-L(s)-1} < C(d,k) - C(d,k,s) < \frac{1}{ln(2)\overline{T}}\,\mu^{-L(s)}, \qquad L(s) \gg 1.$$

References

[1] K.A.S. Immink, "Runlength-Limited Sequences", *Proc. IEEE*, vol. 78, no. 11, pp. 1745–1759, Nov. 1990.

[2] J. Ashley and P.H. Siegel, "A Note on the Shannon Capacity of Run-Length-Limited Codes", *IEEE Trans. Inform. Theory*, vol. IT-33, no. 4, pp. 601–605, July 1987.

[3] T.D. Howell, "Statistical Properties of Selected Recording Codes", *IBM J. Res. Develop.*, vol. 33, no. 1, pp. 60–73, Jan. 1989.

[4] E.N. Gilbert, "Synchronization of Binary Messages", *IEEE Trans. Inform. Theory*, vol. IT-6, no. 5, pp. 470–477, Sept. 1960.

[5] J.J. Stiffler, *Theory of Synchronous Communications,* Engelwood Cliffs: Prentice-Hall, 1971.

[6] L.J. Guibas and A.M. Odlyzko, "Maximal Prefix-Synchronized Codes", *SIAM J. Appl. Math.,* vol. 35, no. 2, pp. 401–418, 1978.

[7] M.P. Schuetzenberger, "On the Synchronizing Properties of Certain Prefix Codes", *Inform. Contr.*, vol. 7, pp. 23–36, 1964.

[8] J. Berstel and D. Perrin, *Theory of Codes,* Orlando: Academic Press, 1985.

[9] W. Feller, *An Introduction to Probability Theory and Its Applications, Volume I,* New York: Wiley, 1959.

[10] D. Lind, "Perturbations of shifts of finite type", *Siam J. Disc. Math.*, vol. 2, pp. 350–365, 1989.

[11] C.E. Shannon, "A Mathematical Theory of Communication", *Bell Syst. Tech. J.*, vol. 27, pp. 379–423, July 1948.

[12] R.L. Adler, D. Coppersmith, and M. Hassner, "Algorithms for Sliding Block Codes. An Application of Symbolic Dynamics to Information Theory", *IEEE Trans. Inform. Theory*, vol. IT-29, no. 1, pp. 5–22, Jan. 1983.

[13] R.S Varga, *Matrix Iterative Analysis,* Engelwood Cliffs: Prentice-Hall, 1962.

[14] P.A. Franaszek, "Construction of Bounded Delay Codes for Discrete Noiseless Channels", *IBM J. Res. Develop.*, vol. 26, no. 4, pp. 506–514, July 1982.

[15] B.H. Marcus and R.M. Roth, "Bounds on the Number of States in Encoder Graphs for Input-constrained Channels", *IEEE Trans. Inform. Theory*, vol. IT-37, no. 3, pp. 742–758, May 1991.

[16] G.V. Jacoby and R. Kost, "Binary Two-thirds Rate Code with Full Word Look-Ahead", *IEEE Trans. Magnet.*, vol. MAG-20, no. 5, pp. 709–714, Sept. 1984; see also M. Cohn, G.V. Jacoby, and C.A. Bates III, U.S. Patent 4,337,458, June 1982.

[17] P.A. Franaszek, "Run-length-limited Variable Length Coding with Error Propagation Limitation", U.S. Patent 3,689,899, Sept. 1972.

[18] J.S. Eggenberger and P. Hodges, "Sequential Encoding and Decoding of Variable Word Length, Fixed Rate Data Codes", U.S. Patent 4,115,768, 1978.

Chapter 4

On the construction of bounded-delay encodable codes for constrained systems

Abstract — **We present a new technique to construct sliding-block modulation codes with a small decoding window. Our method, which involves both state splitting and look-ahead encoding, crucially depends on a new, "local" construction method for bounded-delay codes. We apply our method to construct several new codes, all with a smaller decoding window than previously known codes for the same constraints at the same rate.**

Index Terms — **Bounded-delay code, local encoder structure, ACH algorithm, look-ahead coding, state splitting.**

4.1 Introduction

In various applications it is desirable to translate or *encode* sequences of source data into sequences that obey certain constraints. For example, storage systems based on magnetic and optical disks and tapes often employ (d,k)*-constrained sequences*, binary sequences in which any two consecutive "ones" are separated by at least d and at most k "zeroes". (See e.g. [15], [21].) In most cases of practical interest, the collection of allowable sequences, the *constrained system*, can be described by a *finite-state transition diagram* (FSTD), a labeled digraph

---

© 1995, IEEE. Reprinted, with permission, from IEEE Transactions on Information Theory, vol. 41, no. 5, pp. 1354–1378, September 1995.

The material in this paper was presented in part at the IEEE Symposium on Information Theory, Trondheim, Norway, 1994.

consisting of a finite number of *states* and a finite number of *transitions* between states, each labeled with a *code word.* The constrained system *represented* or *generated* by the FSTD consists of all sequences that result by reading off the labels from the transitions that make up a directed path or *walk* in the FSTD.

A (modulation) *code* for a constrained system consists of an *encoder*, a one-to-one map that transforms sequences of *source symbols* into allowable sequences, together with a *decoder*, the inverse of the encoder map. In practical applications the encoder takes the form of a synchronous finite-state machine. Such a device accepts arbitrary source symbols from a given *source alphabet* B as input, and produces a code word depending on this input and the current state of the encoder. (Such codes are called *synchronous*.) Typically, each source symbol is made up of p user bits, and each code word is composed of q channel bits, for some fixed numbers p and q. The number p/q is called the (binary) *rate* of the code.

Since modulation codes are normally used in a noisy environment, it is important that the decoder limits the propagation of input errors. All codes considered in this paper will have a *sliding block decoder.* Such a decoder retrieves the original source symbol at the input of the encoder from the corresponding code word along with a fixed number m of preceding code words and a fixed number a of subsequent code words. We refer to these $m+1+a$ code words as the *decoding window* for this source symbol, and we say that the code has a *decoding-window size* of $m+1+a$ code words. Note that a single error in the input to a sliding-block decoder affects at most $m+1+a$ source symbols, so the amount of *error propagation* (the maximum number of erroneous bits in the output sequence produced by the decoder caused by a single bit error at the input) remains bounded. The decoding-window size also has a direct bearing on the amount of hardware needed to implement the decoder. For these reasons, it is an important parameter of a code and should be as small as possible.

Throughout the years, various code construction methods have been devised. The effectivity of these methods depends (amongst others) on the desired rate p/q of the code. In this paper we concentrate on methods which are most effective when p and q are relatively small. (For a more general overview, see e.g. [15].) A classical technique is the Bounded-Delay (BD) method developed by Franaszek in a sequence of papers [6], [7], [8], [9]. This method constructs codes that employ *look-ahead encoding* (or, equivalently, involve an *encoding delay*) and aims to minimize the decoding window. A drawback of the BD method is that code construction may involve large and complex search problems. A breakthrough occurred with the discovery of the ACH state splitting algorithm [2]. This algorithm, which in its simplest form is polynomial-time and of low complexity, is guaranteed to produce a code for a constrained system of *finite type*, at any (rational) rate not exceeding the *Shannon capacity* of the constraint. However, the

method allows a great deal of freedom of choice and it is far from evident how to use this freedom to obtain a code with a small decoding window.

Both the BD method and the ACH algorithm produce codes for a given constrained system by transforming a given FSTD that represents these constraints. Moreover, in both cases the transformations are guided by an *approximate eigenvector* that assigns certain *weights* to the states of the given FSTD. The ACH algorithm does so by a process called (forward) *state splitting*. Here, a (forward) state split essentially subdivides a state by partitioning the set of transitions leaving that state. Our description of BD codes in Section 3 shows that the BD-method may be considered as *backward state splitting*, where states are now subdivided by partitioning the set of transitions *entering* a state. We will refer to these transformations by saying that states are subdivided according to their future (as in the ACH algorithm) or to their history (as in the BD-method).

In this paper, we propose a code construction technique that *combines* aspects of these two methods, generalizing the approach in [17]. The method aims specifically at the design of codes with a *small decoding window*. The construction involves both state splitting and look-ahead encoding (thus subdivides states according to both future and history), and is again guided by an approximate eigenvector. Our technique crucially depends on a new construction method for BD codes from what we call a *local encoder structure*. This method allows the decomposition of the difficult *global* code construction problem into a number of independent, relatively easy, *local* problems, one for each state.

The existence of a valid encoder structure in a given state (whose construction makes up the said local problem for this state) depends only on the weights assigned to this state and its successors. In particular, if all weights are "sufficiently small", then a local encoder structure can always be found. However, it may happen that for some states no valid encoder structure exists. In that case, we reduce the state-weights by state splitting operations, where the state splitting process is driven by the requirement that a local encoder structure can be formed in each of the states. As in the ACH algorithm, we strive to keep the number of *rounds* of state splitting (see e.g. [19]) as small as possible, since this number has a direct bearing on the size of the decoding window of the resulting code.

Once we have obtained a complete local encoder structure, it is fairly easy to construct the required BD code. In fact, this construction still affords some freedom of choice, which we can employ to minimize the decoding window and error propagation of the resulting code. Only at this point the specific code word labeling of the FSTD is taken into account.

In all examples that we consider (and some of these involve fairly complex constraints), the appropriate choice of state splitting operations is fairly evident. However, we do not offer a clear-cut algorithm on which to base these choices.

Nevertheless, since our technique can be considered as an extension of ACH, *any* sequence of valid state splittings will eventually produce a code whenever ACH would. In our view, the method is best implemented *interactively*; the user chooses the state splits but the remainder of the work is left to the computer program.

Our method would not be very exciting if the required amount of state splitting was comparable to that required in straightforward ACH. Fortunately, this does not seem to be the case. Indeed, the motivation behind this method stems from the following remarkable observation. An encoder produced from the ACH algorithm by M rounds of state splitting (i.e. M future transitions are taken into account) has a decoding window of size $M+c$, for some constant c depending only on the actual labeling of the FSTD. But sometimes it may happen that decoding requires only the $K + c$ *last* code words from the decoding window, for some $K < M$. (An extreme example of this behaviour occurs for bounded-delay encodable, block-decodable (BDB) codes [12], where $K = 1$ and M can be arbitrarily large. This will be explained elsewhere.) In other words, the *effective* window size may be much smaller than the ordinary window size. (For this reason the lower bound on the decoding-window size in [18] has to be applied with care.) We will refer to this property by saying that the code has the *reduced window property.* At first sight, coaching ACH into producing a code with the reduced window property seems difficult: the algorithm allows much freedom and there is no evident criterium on which to base the choice of the state splitting steps. Is there a way to profit from this phenomenon in a systematic way? We will show that in the case described above, our code construction technique would require at most K rounds of state splitting (and often much less) to produce a *shifted* version of the ACH code, i.e., corresponding code sequences are equal up to a fixed shift.

In practice, our method performs very satisfactorily, as is witnessed by many examples. In particular, in this paper we derive the following new codes:

– A rate-2/3 $(d, k) = (1, 6)$-constrained code with decoding-window size of 5 symbols (in fact, only 14 bits), with an error propagation of only 9 bits. (To the best of our knowledge, the previous record is held by a code described in [1], which has an error propagation of 11 bits.)

– A rate-2/5 $(d, k, s) = (2, 18, 2)$-code, with a decoding-window size of 3 symbols. (The previous record was 4 symbols, see [19].)

– A rate-2/3 $(d, k) = (1, 9)$-constrained code with a decoding-window of size 2 symbols or 6 bits.

All of these codes were constructed by hand in a few hours. Moreover, many known good codes can also be found by our method, often with surprising ease.

Finally, we observe that our approach allows a uniform description to be given of various code construction methods such as ACH [2], BD method [8], variable length codes, prefix-free list codes, variable-length state splitting [3], [11], and a

method described by Cohn and Lempel in [17].

The paper is organized as follows. We begin in Section 2 by reviewing the code construction problem. Here we also recall some definitions and establish most of our notation. In Section 3, we discuss look-ahead encoders and Bounded Delay (BD) codes. Local encoder structures are introduced in Section 4. In Appendix A, we give a proof that a BD code can be constructed from a given local encoder structure, and we discuss the complexity of local encoder construction. In Section 5, we describe our code construction method. This construction delivers an encoder that employs look-ahead. In Section 6, we explain how such an encoder can be transformed into a conventional encoder. In Section 4.7, we illustrate our methods by constructing three new codes, and in Section 8, we offer some conclusions. Finally, we briefly review the ACH state splitting algorithm in Appendix B. Here we also establish the connection between ACH and our methods. Some further details concerning this connection are provided by Appendix C

4.2 Codes for constraints of finite type

In this section, we consider the coding problem for constrained systems in more detail. Along the way we establish some notation and introduce a number of concepts that will be used later on. The reader is advised to skim this section on first reading, and to return here later if necessary.

Suppose that we have to design a (synchronous) code for a constrained system $\mathcal{L}$, with a given *source alphabet* $\boldsymbol{B}$ of size $N = |\boldsymbol{B}|$, at a given (binary) rate $R = p/q$. For convenience, we will assume that each *source symbol* from $\boldsymbol{B}$ consists of p user bits. Typically, $\mathcal{L}$ is described with the aid of a *finite-state transition diagram* or FSTD. Such an FSTD — which is in fact a labeled directed graph — is described by a four-tuple $\mathcal{M} = (V, A, L, \boldsymbol{F})$, where V is a finite collection of vertices or *states*, A is a finite collection of arcs or *transitions*, and $L : A \mapsto \boldsymbol{F}$ is the *labeling map* that assigns to each transition a symbol from the *labeling alphabet* $\boldsymbol{F}$. Each transition α of $\mathcal{M}$ has a *initial state* $\text{beg}(\alpha)$ and a *terminal state* $\text{end}(\alpha)$, and is referred to as a *transition from* $\text{beg}(\alpha)$ *to* $\text{end}(\alpha)$. (Note that loops and multiple transitions are allowed.) We let A_v denote the collection of transitions leaving state v (the outgoing transitions in v), that is, all transitions α with $\text{beg}(\alpha) = v$. We say that $\mathcal{L}$ is *generated* (or *represented*) by $\mathcal{M}$ if $\mathcal{L}$ consists of all words $L(\boldsymbol{\alpha}) = L(\alpha_1) \cdots L(\alpha_n)$ obtained from sequences $\boldsymbol{\alpha} = \alpha_1 \cdots \alpha_n$ of transitions in $\mathcal{M}$ for which $\text{end}(\alpha_i) = \text{beg}(\alpha_{i+1})$, $i = 1, \ldots, n-1$. Such a sequence $\boldsymbol{\alpha}$ is called a *walk* in $\mathcal{M}$ of *length* n, and $L(\boldsymbol{\alpha})$ is called the word *generated* by this walk. The collection of all walks in $\mathcal{M}$ will be denoted by Σ.

Usually, we choose $\mathcal{M}$ to be *deterministic*. Here, a FSTD is called determinis-

tic if in each state the outgoing transitions carry distinct labels. (Any given FSTD that presents $\mathcal{L}$ can easily be transformed into a deterministic presentation for $\mathcal{L}$.)

Moreover, it is common practice to choose $\mathcal{M}$ to be the *Shannon cover*, the (unique) deterministic presentation with the *minimum* number of states. Any given deterministic FSTD that presents $\mathcal{L}$ can easily be transformed into the Shannon cover by a process called *state merging*. Here, if the outgoing transitions in two states can be paired off in such a way that the two transitions in each pair carry the same label and end in the same state, then from the point-of-view of sequence generation these states accomplish the same, hence they can as well be combined, or *merged* as it is usually called, into a single state. (Note that the resulting FSTD will again be deterministic.) This process of merging states is repeated until no more merging is possible. It can be shown that the resulting FSTD, the Shannon cover, does not depend on the order in which the merging was carried out, and that, moreover, no deterministic FSTD presenting the same constrained system can have fewer states, and must be equal to the Shannon cover if it has the same number of states. Nevertheless, in what follows, we do not yet make any special assumptions about the FSTD $\mathcal{M}$ chosen to presents the constrained system $\mathcal{L}$, except that we require that $\mathcal{M}$ contains no *sources* and no *sincs*, i.e., we require that each state of $\mathcal{M}$ can both be entered and left.

The said coding problem for $\mathcal{L}$ requires that we translate or *encode* arbitrary sequences of p-bit source symbols into sequences of q-bit *code words* , where the resulting sequence is required to be in $\mathcal{L}$, that is, can be generated by some walk in $\mathcal{M}$. Usually the code word size q is a multiple of the size q_1 of the labeling symbols in $\boldsymbol{F}$, say $q = nq_1$. (Typically, $\boldsymbol{F} = \{0, 1\}$, $q_1 = 1$, and $n = q$.) Therefore it is desirable to have a representation for $\mathcal{L}$ where the labeling symbols are actually code words. Such a representation is provided by the *nth order power graph* $\mathcal{M}^n$ of $\mathcal{M}$. This FSTD has the same states as $\mathcal{M}$, but the transitions are now walks of length n in $\mathcal{M}$; a walk $\boldsymbol{\alpha} = \alpha_1 \cdots \alpha_n$ is considered as a transition in $\mathcal{M}^n$ from state $\mathrm{beg}(\boldsymbol{\alpha}) = \mathrm{beg}(\alpha_1)$ to state $\mathrm{end}(\boldsymbol{\alpha}) = \mathrm{end}(\alpha_n)$, and is labeled with the code word $L(\boldsymbol{\alpha})$ generated by $\boldsymbol{\alpha}$. So after replacing $\mathcal{M}$ by an appropriate power graph, we may assume that the labels of $\mathcal{M}$ are actually q-bit code words, and now the encoding problem is to transform an arbitrary source sequence into a sequence of code words that can be generated by some walk in $\mathcal{M}$.

Commonly, an encoder for $\mathcal{L}$ can actually be put into the form of an encoder *for* Σ, the collection of walks in $\mathcal{M}$. That is, we may think of the encoder as actually producing the *walk* that generates the sequence of code words, instead of the sequence itself. (We observe that Σ is also a constrained system; indeed, $\mathcal{M}$ represents Σ if L is taken to be the identity map.) Note that an encoder for Σ produces a transformation from source sequences into sequences of $\mathcal{L}$ through the labeling map L.

However, some restriction on L is required to ensure that this transformation is actually a *code*, that is, that decoding is possible. The proper requirement is that L is of *finite type*: there are numbers m and a, the *memory* and *anticipation*, respectively, such that walks $\alpha_{-m} \cdots \alpha_0 \cdots \alpha_a$ that generate a given code word sequence $f_{-m} \cdots f_0 \cdots f_a$ all agree in transition α_0. If this is the case, then we can reconstruct the walk $\alpha_{-m} \cdots \alpha_0 \cdots \alpha_{n+a}$ used to generate a given sequence $f_{-m} \cdots f_0 \cdots f_{n+a}$ in $\mathcal{L}$, except for the first m and last a transitions. Indeed, note that each transition α_k, $0 \leq k \leq n$, is uniquely determined by $f_{k-m} \cdots f_k \cdots f_{k+a}$, the *decoding window* for α_k.

In the context of encoder maps, a map of this type is termed *sliding-block decodable*, with *decoding-window size* $m + 1 + a$ and *window* (m, a). The use of the term "finite type" for such a labeling is not standard, but we prefer it because it avoids the suggestion that source symbols are to be recovered instead of transitions. We also introduce the notion of a map of *almost-finite type*; the definition is similar to the one for finite type, except that now we also provide the initial state of the transition α_0 to be recovered.

It is well-known that $\mathcal{L}$ is of finite type (i.e. $\mathcal{L}$ can be generated by an FSTD of finite type) precisely when $\mathcal{L}$ can be specified in terms of a finite list of *forbidden blocks* [2]. It can also be shown that $\mathcal{L}$ is of almost-finite type (see e.g.[19]) precisely when $\mathcal{L}$ can be generated by a deterministic FSTD of almost-finite type [16]. Since the process of state merging referred to above does not destroy either of these properties, it follows that $\mathcal{L}$ is of (almost-) finite type if and only if its Shannon cover is of (almost-) finite type. (Indeed, it is not difficult to show that $\mathcal{M}$ is of almost-finite type if and only if $\mathcal{M}$ has both finite local memory and finite local anticipation. For a definition of these notions, we refer to [19].)

It can be shown that a synchronous code for a constrained system $\mathcal{L}$ represented by an FSTD $\mathcal{M}$ of (almost-) finite type, that is, with a labeling of (almost-) finite type, can always be transformed into one arising from a code for Σ. (This transformation may cause a fixed amount of *encoding delay*, or equivalently, may require a fixed amount of *look-ahead*, at the encoder.) It is important to note that if the labeling map L of $\mathcal{M}$ is of finite type and the code for Σ is sliding-block decodable, then the corresponding code for $\mathcal{L}$ is again a sliding-block code. (Indeed, note that each successive recovery operation causes only a limited loss of information at both ends of the recovered sequence.) So for most purposes, we can simply forget about the particular labeling L of $\mathcal{M}$ as long as L is of finite type, and concentrate on constructing a code for Σ.

Typically, to construct a code for Σ, the FSTD $\mathcal{M}$ is first transformed into another FSTD $\mathcal{M}^*$ that is more suitable for encoding purposes, where the transformation is such that $\mathcal{M}^* = (V^\star, A^\star, L^\star, A)$ generates Σ, through a map L^* of finite type that assigns to each transition α^* in $\mathcal{M}^*$ its *parent* transition in $\mathcal{M}$.

(The map L^* is sometimes called the *ancestor map*.)

A good example is provided by the ACH algorithm; if $\mathcal{M}^*$ is obtained from $\mathcal{M}$ by M rounds of state splitting, then the map L^* has window $(0, M)$, see Appendix B. Another transformation, and one that will play an important role in this paper, involves *history graphs* of $\mathcal{M}$. In that case, the rôle of $\mathcal{M}^*$ is played by the *Tth order history graph* $\mathcal{M}_T$ of $\mathcal{M}$, for a suitable T. Here, $\mathcal{M}_T$ is the FSTD with as states the walks of length T and as transitions the walks of length $T+1$ in $\mathcal{M}$; a walk $\boldsymbol{\alpha} = \alpha_0 \cdots \alpha_T$ of length $T+1$ in $\mathcal{M}$ represents a transition in $\mathcal{M}_T$ from the state $\text{beg}(\boldsymbol{\alpha}) = \alpha_0 \cdots \alpha_{T-1}$ to the state $\text{end}(\boldsymbol{\alpha}) = \alpha_1 \cdots \alpha_T$, and has label $L^*(\boldsymbol{\alpha}) = \alpha_T$. (Observe that the graph underlying $\mathcal{M}_T$ is the Tth order edge graph $\mathcal{M}^{[T]}$ of $\mathcal{M}$; in fact, the only difference between these graphs is that a transition $\boldsymbol{\alpha}$ as above now carries label $L^*(\boldsymbol{\alpha}) = \alpha_0$. Walks in these graphs generate the same sequences, but shifted by $T+1$ symbols.) While state splitting essentially subdivides states according to their *future*, in history graphs states are subdivided according to their *history*, that is, according to how states are entered. (See also Section 3.)

Following such preparations, the code-construction process is completed by transforming $\mathcal{M}^*$ into a (look-ahead) *encoder* (see Section 3): the transitions of $\mathcal{M}^*$ are labeled with source symbols (represented by a map $\varphi : A^\star \mapsto \boldsymbol{B}$) and an encoder map $\Psi : \boldsymbol{B}^* \mapsto \Sigma^\star$ is defined that maps each source sequence $\{b_n\}$ onto a walk $\{\alpha_n^\star\}$ in $\mathcal{M}^*$ for which $\varphi(\alpha_n^\star) = b_n$. (For example, if the FSTD $\mathcal{M}^*$ is obtained from the ACH algorithm, then the encoder takes the form of a finite-state machine.)

The entire encoding process can then be presented schematically as

$$\{b_n\} \overset{\Psi}{\mapsto} \{\alpha_n^\star\} \overset{L^\star}{\mapsto} \{\alpha_n\} \overset{L}{\mapsto} \{f_n\}, \tag{1}$$

where Ψ represents a look-ahead encoder for $\Sigma^\star$, and $L^* : \Sigma^\star \mapsto \Sigma$ is a map of finite type. As will be shown in future work, this encoding scheme is *universal*, in the sense that each sliding-block code with a finite-type encoder (wrt. the labeling with code words) for a constrained system $\mathcal{L}$ (of finite type) presented by an FSTD $\mathcal{M}$ of finite type can be transformed into a code of this form. (The universality of this scheme is also proven in [4].) We observed earlier that if the labeling map L is of finite type, then the concatenation $L^* \circ L$ of L^* and L is also sliding-block decodable. Note furthermore that decoding of a code word f_n is immediate once $\alpha_n^\star$ is obtained, since $b_n = \varphi(\alpha_n^\star)$. In this paper we refer to the decoding window of L^* as the *path-decoding window*, to stress the independence of the decoding of $\Psi \circ L^*$ from the particular labeling L used to generate $\mathcal{L}$. In what follows, we will concentrate on the construction of codes *for* Σ, the constrained system of the paths from $\mathcal{M}$, mostly forgetting about the particular labeling L of $\mathcal{M}$ but assuming only that L is of finite type.

4.3 Look-ahead encoders and BD codes

In this section we investigate encoders that employ *look-ahead* (see e.g. [8], [17]). A conventional encoder for a given constrained system $\mathcal{L}$ is based on an *encoder graph*, that is, an FSTD that generates (a subset of) $\mathcal{L}$, with the additional property that precisely N transitions leave each state, where N denotes the size of the source alphabet for the code. From such an encoder graph a finite-state encoder for $\mathcal{L}$ (in fact, many different ones) can then be obtained by labeling the N outgoing transitions in each state with N different source symbols. Given the present state of the encoder, the upcoming source symbol to be encoded then determines uniquely the next transition in the encoding path, hence the next code word together with the next encoder state. (So the encoder takes the form of a synchronous finite-state machine.) Decoding is possible if the encoding path can be reconstructed from the resulting sequence of code words.

A *look-ahead encoder* also involves an FSTD that generates (a subset of) $\mathcal{L}$ together with a labeling of its transitions with source symbols. Again, each source symbol must be encoded by a transition labeled with this symbol. But in general we no longer require that each source symbol occurs as a label on the outgoing transitions from a state (nor that the transitions leaving a state are labeled with distinct source symbols). As a consequence, the encoder has to avoid entering a state from which the upcoming source symbol(s) cannot be encoded. To achieve this, the encoder is allowed to base its choice of the next transition to be included in the encoding path on the next $M + 1$ upcoming source symbols, for some fixed number M termed the *look-ahead* of the encoder. So a state is entered with knowledge of the next M upcoming source symbols only (one symbol being encoded by the entering transition), and encoding from this state must be possible for *all* source sequences starting with these M symbols. Consequently, with each state of the encoder there will be associated a finite *list of admissible prefixes* (each member being a source word of length at most M) specifying the prefixes of sequences of source symbols that can be encoded from this state. Let us consider the following simple example.

Example 1: In Figure 1 we present a typical look-ahead encoder, with source alphabet $\boldsymbol{B} = \{0, 1\}$. (The actual code words with which the transitions are labeled are not relevant for what follows and are not shown in the figure. That is, we consider the code as a code for Σ, the collection of walks in $\mathcal{M}$. See also Section 2.) The encoder has three states, named a, b, and c. In state a, no requirements are imposed on upcoming source symbols (indicated in Figure 1 by the prefix list $[-]$ containing the *empty* word only.) State b may be entered only with an upcoming source symbol equal to 0 (indicated by the prefix list $[0]$), and

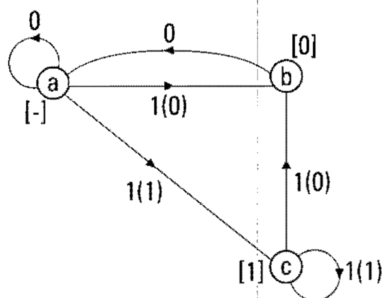

Figure 1: *A look-ahead encoder.*

similarly, the prefix list [1] is associated with state c . Each transition in Figure 1 carries a source symbol, sometimes followed by a parenthesized symbol indicating a constraint on the upcoming source symbol. For example, to encode source symbol 1 from state a, the encoder looks ahead to the *next* upcoming source symbol, and chooses the transition from a to b, labeled 1(0), if this symbol equals 0, or otherwise the transition from a to c, labeled 1(1). (A similar notation is used e.g. in [17].) The reader will have no difficulty verifying that Figure 1 does indeed present an encoder; encoding of a given source sequence, starting in a given encoder state, is possible if some prefix of the sequence is contained in the list of admissible prefixes associated with that state. This encoder never employs more than *one* symbol look-ahead, and is therefore called a *one-symbol look-ahead encoder.* If the actual labeling of $\mathcal{M}$ is of finite type and generates a constrained system $\mathcal{L}$, then the resulting code for $\mathcal{L}$ will be sliding-block decodable, as explained in Section 2. Decoding of a code word is immediate once the corresponding transition in Figure 1 is known; we express this fact by saying that the code for Σ has a *path-decoding window* of size one. □

We now seek to characterize an M-symbol look-ahead encoder in terms of the lists of admissible prefixes $\mathcal{W}_v$ associated with states v of the encoder. In what follows, we will assume that each list $\mathcal{W}_v$ consists entirely of words of length M over $\boldsymbol{B}$, where $\boldsymbol{B}$ denotes the source alphabet of the code. (Note that we can replace each $\mathcal{W}_v$ by the set of words of length M with a prefix in $\mathcal{W}_v$ without altering the collection of source sequences with a prefix in $\mathcal{W}_v$.) Now suppose that a given sequence of source symbols $b_1 \cdots b_M$ is contained in $\mathcal{W}_v$. Then for all choices of further source symbols $b_{M+1}, \ldots$, the sequence $\{b_n\}_{n\geq 1}$ has a prefix contained in $\mathcal{W}_v$ and should therefore be encodable from state v. By assumption, the encoder must base its choice of the transition leaving v that will encode b_1 on $b_1 \cdots b_M$ and b_{M+1} only. Therefore, for *each* source symbol b_{M+1} from $\boldsymbol{B}$, there must be a transition α leaving state v that encodes b_1 (that is, labeled with

source symbol b_1), with the further property that the source sequence $\{b_n\}_{n\geq 2}$ can be encoded from the terminal state end(α) of α for *all* choices of $b_{M+2}b_{M+3}\cdots$, that is, its prefix $b_2\cdots b_{M+1}$ must be contained in $\mathcal{W}_{\mathrm{end}(\alpha)}$.

Conversely, if the lists $\mathcal{W}_v$ satisfy these requirements and if at least one list $\mathcal{W}_v$ is non-empty, then M-symbol look-ahead encoding is possible. Indeed, if $\mathcal{W}_s$ is non-empty, then all source sequences can be encoded, starting in this state s, if we prefix all sequences with a fixed word in $\mathcal{W}_s$. It is easily seen that the encoding process will never get stuck. (At some stage of the encoding process several choices for the encoding transition may be possible. However, if desired the encoder can easily be made deterministic. Anyway, we do not mind much as long as all encoding alternatives are decoded into the same, original, source sequence, which will always be the case here.)

At this point, it is convenient to introduce some additional notation. We denote by $\mathcal{W}_v\boldsymbol{B}$ the set of all words $b_1\cdots b_{M+1}$ with $b_1\cdots b_M \in \mathcal{W}_v$ and $b_{M+1} \in \boldsymbol{B}$. If $\varphi(\alpha)$ denotes the source label of transition α, then we write $\varphi(\alpha)\mathcal{W}_{\mathrm{end}(\alpha)}$ to denote the collection of all words $\varphi(\alpha)b_2\cdots b_{M+1}$ with $b_2\cdots b_{M+1} \in \mathcal{W}_{\mathrm{end}(\alpha)}$. We use this notation in the following formal definition, which, in view of the preceding discussion, may also be termed a characterization, of an M-symbol look-ahead encoder.

Definition 1: An *M-symbol look-ahead encoder* with *source alphabet $\boldsymbol{B}$* for a constrained system $\mathcal{L}$ consists of a triple $[\mathcal{M}, \varphi, \mathcal{W}]$, where $\mathcal{M}$ is an FSTD representing (a subset of) $\mathcal{L}$, $\varphi : A \mapsto \boldsymbol{B}$ denotes a labeling of the transitions with symbols from $\boldsymbol{B}$, and $\mathcal{W} : V \mapsto \mathcal{P}(\boldsymbol{B}^M)$ is a map which assigns to each state v of $\mathcal{M}$ a collection $\mathcal{W}_v$ of words of length M over $\boldsymbol{B}$, termed the *list of admissible prefixes at v*, such that
a) at least one $\mathcal{W}_v$ is non-empty, and
b) for all states v, we have

$$\mathcal{W}_v\boldsymbol{B} \subseteq \bigcup_{\alpha\in A_v} \varphi(\alpha)\mathcal{W}_{\mathrm{end}(\alpha)}, \tag{2}$$

where A_v denotes the set of transitions leaving v.

The above definition may be interpreted as requiring that a) the encoder has a starting point, and b) there will always be a transition to encode the next source symbol. Note that a look-ahead encoder actually produces an encoder *for* Σ, the collection of walks in $\mathcal{M}$. Note also that decoding of a code word is immediate once the corresponding transition is known, that is, the encoder has a *path-decoding window* of size one. In Section 6 we show how to obtain an ordinary finite-state encoder from a given look-ahead encoder.

Note that sometimes we may have a choice for the encoding transition. If, in addition, the encoding transition is specified in each case, then we speak of a *fully specified* look-ahead encoder.

A fundamental notion in the theory of code construction is that of an *approximate eigenvector.* (See e.g. [6], [17], [2].) We will now show that if we assign a weight $|\mathcal{W}_v|$ to each state v of a look-ahead encoder, then these weights constitute an approximate eigenvector, a result first obtained by Franaszek [6]. We first recall the definition of an approximate eigenvector. Let $\mathcal{M} = (V, A, L, \boldsymbol{F})$ be a FSTD, and let N denote a positive integer. We say that a map $\phi : V \mapsto \boldsymbol{Z}_+$, which associates with each state v of $\mathcal{M}$ a non-negative integer *weight* ϕ_v, is an $[\mathcal{M}, N]$*-approximate eigenvector* if for each state v the sum of the weights of terminal states of the outgoing transitions is not less than N times the weight of state v, that is, if

$$N\phi_v \leq \sum_{\alpha \in A_v} \phi_{\mathrm{end}(\alpha)} \tag{3}$$

holds for each state v. (Usually, we think of ϕ as a *vector*, and the above condition is stated as

$$N\phi \leq D_{\mathcal{M}}\phi. \tag{4}$$

Here $D_{\mathcal{M}}$ denotes the $|V| \times |V|$ *adjacency matrix* of $\mathcal{M}$; $D_{\mathcal{M}}(v, w)$ equals the number of transitions from state v to state w. The above inequality is to be interpreted component-wise.)

Theorem 1: Let $[\mathcal{M}, \varphi, \mathcal{W}]$ be an M-symbol look-ahead encoder with source alphabet $\boldsymbol{B}$. Then the weights ϕ_v defined by $\phi_v = |\mathcal{W}_v|$, the size of the list of admissible prefixes associated with state v, satisfy $0 \leq \phi_v \leq |\boldsymbol{B}|^M$ and constitute an $[\mathcal{M}, |\boldsymbol{B}|]$-approximate eigenvector.

Proof: This follows immediately from (2). Indeed, we have

$$\begin{aligned}
\phi_v|\boldsymbol{B}| &= |\mathcal{W}_v\boldsymbol{B}| \\
&\leq |\bigcup_{\alpha \in A_v} \varphi(\alpha)\mathcal{W}_{\mathrm{end}(\alpha)}| \\
&\leq \sum_{\alpha \in A_v} |\varphi(\alpha)\mathcal{W}_{\mathrm{end}(\alpha)}| \\
&= \sum_{\alpha \in A_v} |\mathcal{W}_{\mathrm{end}(\alpha)}| \\
&= \sum_{\alpha \in A_v} \phi_{\mathrm{end}(\alpha)}.
\end{aligned}$$

□

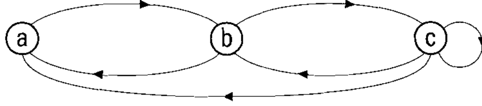

Figure 2: *An FSTD $\mathcal{M}$.*

It is not always possible to find a look-ahead encoder of the form $[\mathcal{M}, \varphi, \mathcal{W}]$ for a given FSTD $\mathcal{M}$, even if $\log_2 |\boldsymbol{B}|$ does not exceed the *capacity* of $\mathcal{M}$ (a number indicating the maximum possible rate of a code for Σ, see [20]). To gain more freedom in code construction, we might allow both the sets $\mathcal{W}_v$ and the labeling of the transitions leaving v to depend on the *history* in v, that is, on the way that the state v is entered. Such codes were investigated in [8] under the name of Bounded Delay (BD) codes. These codes can be described conveniently with the aid of the *Tth order history graph* M_T of $\mathcal{M}$ as introduced in Section 2. We think of the state $\alpha_0 \cdots \alpha_{T-1}$ of $\mathcal{M}_T$ as representing the state $v = \text{end}(\alpha_{T-1})$ when entered through the walk $\alpha_0 \cdots \alpha_{T-1}$. Similarly, we think of a (source word or code word) label of the transition $\alpha_0 \cdots \alpha_T$ in $\mathcal{M}_T$ as the label of the transition α_T leaving v, provided that v is entered through the walk $\alpha_0 \cdots \alpha_{T-1}$. We now state the formal definition of BD codes.

Definition 2: A *bounded delay (BD) code* for an FSTD $\mathcal{M}$ is a code for the collection Σ of walks in $\mathcal{M}$ that can be encoded by an M-symbol look-ahead encoder of the form $[\mathcal{M}_T, \varphi, \mathcal{W}]$, for some numbers M and T. We refer to M and T as the *look-ahead* and *path-memory* of the BD code, respectively.

As long as $\log_2 |\boldsymbol{B}|$ does not exceed the capacity of $\mathcal{M}$, we can always find a BD code for $\mathcal{M}$ with look-ahead M and path-memory T, for some numbers M and $T \leq M$. Indeed, under these assumptions the state splitting method always works [2], and the results in [10] or in Appendix B show that each code obtained by the state splitting method can also be obtained as a BD code.

We illustrate the above ideas by another example, taken from [8].

Example 2: Let $\mathcal{M}$ be the FSTD in Figure 2, and take $B = \{0, 1\}$. It can be shown that there is no look-ahead encoder with source alphabet $\boldsymbol{B}$ for $\mathcal{M}$. (A reasonably simple proof of this statement involves ideas such as stable state-sets which are outside the scope of this paper; the interested reader will find a proof in future work.)

On the other hand, Figure 3 describes a BD code for $\mathcal{M}$ with look-ahead $M = 2$ and path-memory $T = 1$. Here as well as in further examples the encoding rules are described *locally* for each state, possibly depending on the way that a state has been entered. In this example, the encoding rules for state b will depend on whether b has been entered from state a or from state c; encoding

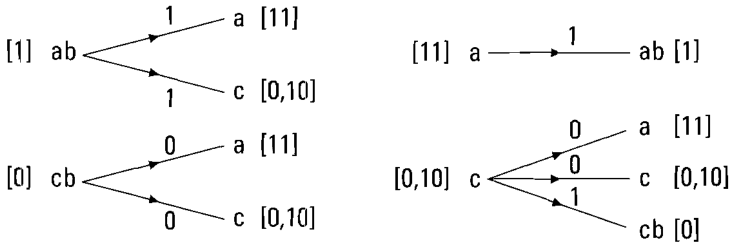

Figure 3: *A two-symbol look-ahead encoder for $\mathcal{M}^\star$.*

rules for other states will not depend on history. (Hence we have $T = 1$.) The notation in Figure 3 is (hopefully) evident; the lists of admissible prefixes $\mathcal{W}_a = \{11\}$, $\mathcal{W}_c = \{00, 01, 10\}$, $\mathcal{W}_{ab} = \{10, 11\}$, and $\mathcal{W}_{cb} = \{00, 01\}$ are represented in this Figure by the short-hand notation [11], [0, 10], [1], and [0], respectively.

To obtain this code, we first transform $\mathcal{M}$ into a FSTD $\mathcal{M}^*$ by splitting state b "backwards" into a state ab (which can only be entered from state a) and a state cb (which can only be entered from state c). Then we can construct the look-ahead encoder for $\mathcal{M}^*$ as shown in Figure 3. (Code construction for this example is discussed again in Example 3 in the next section.)

Each transition in $\mathcal{M}^*$ has a unique "parent" transition in $\mathcal{M}$. Hence an encoding path in $\mathcal{M}^*$ produces a unique encoding path in $\mathcal{M}$. On the other hand, the present transition in the encoding path in $\mathcal{M}$, together with the previous transition, uniquely determines the present transition in $\mathcal{M}^*$; in other words, the code has a *path-decoding window* of size two. Note that a look-ahead encoder for $\mathcal{M}_1$ (as required by the formal definition of a BD code in Definition 2) can be constructed immediately from Figure 3. □

Decoding a BD code for $\mathcal{M}$ requires the reconstruction of the transitions of the encoding path in $\mathcal{M}_T$, each of which is made up of $T + 1$ consecutive transitions in $\mathcal{M}$. In other words, the code for Σ has a *path-decoding window* of size $T + 1$; decoding of a particular transition in $\mathcal{M}$ also requires knowledge of the T previous transitions. For future reference, we state this observation separately as a theorem.

Theorem 2: A BD code for $\mathcal{M}$ as in Definition 2 has a path-decoding window of size $T + 1$, where T denotes the path-memory of the code.

We stress again the fact that from a BD code for $\mathcal{M}$ we can obtain a code for the constrained system $\mathcal{L}$ generated by $\mathcal{M}$, provided that the labeling map L of $\mathcal{M}$ is of finite type. (Indeed, to encode a given source sequence, read off the code word labels of the walk produced by the BD code.) A code for $\mathcal{L}$ of this type will be referred to as a *BD code for $\mathcal{L}$*. Note that if the labeling L is of finite type, then the walk in $\mathcal{M}$ can be "sliding-block" reconstructed from the

code word sequence generated by the walk, so that the corresponding BD code for $\mathcal{L}$ is sliding-block decodable.

We see that, essentially, a BD code for $\mathcal{M}$ is obtained by subdividing states of $\mathcal{M}$ according to their *history*, that is, by "backwards state splitting". In contrast, the ACH state splitting algorithm subdivides states according to their *future*. (In [4] these fundamental operations are termed *in-splitting* and *out-splitting*, respectively.) The construction methods that we shall propose later can be considered as a *mixture* of these two methods, where states are subdivided according to both their history and future.

BD codes were introduced by Franaszek in [8], where they were termed "stationary" BD codes. The more general "periodic" BD codes in that paper are, in our terminology, BD codes for higher-order power graphs (see Section 2) of $\mathcal{M}$.

4.4 Local encoder structures and BD partitions

A major drawback of BD codes is that their construction, as described in a sequence of papers by Franaszek [6]—[9], involves a complicated process of partitioning state-trees and can be difficult. In this section we describe a much simpler construction method. Its main feature is that it partitions the (difficult) global code construction problem into a number of independent (relatively easy) local construction problems, one for each state of the FSTD. The local problem in a particular state requires the construction of a *local encoder structure* in that state, and involves only the weights assigned to that state and its successors. As we will show, it is fairly easy to construct a BD code once a local encoder structure is obtained in each state.

Contrary to Franaszek's method, our method does not always succeed. Indeed, there may be states for which no local encoder structure exists. This can happen if the weights of the successors to that state do not "fit together". Fortunately, this problem can always be solved by using state splitting to break down the weights of (some of) the successors into smaller weights, as will be explained in the next section.

The motivation for our method is the following. Let us consider a BD code $[\mathcal{M}_T, \mathcal{W}, \varphi]$ for a FSTD $\mathcal{M}$, with source alphabet B and path-memory T (see Definition 2). Recall that a BD code associates with each state v of $\mathcal{M}$ various lists of admissible prefixes, one for each particular history in v (represented by walks of length T ending in v). Sometimes such a BD code is very regular, in the sense that, for each state v, these lists can all be obtained from one special list of words (representing the common "structure" of these lists) by a process that will be called *relabeling*. (Then, as we will see, a local encoder structure can be

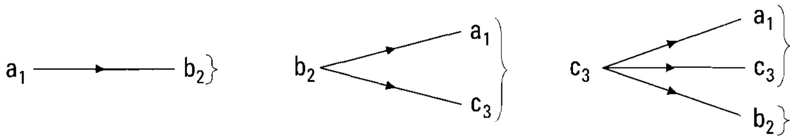

Figure 4: *The BD partition.*

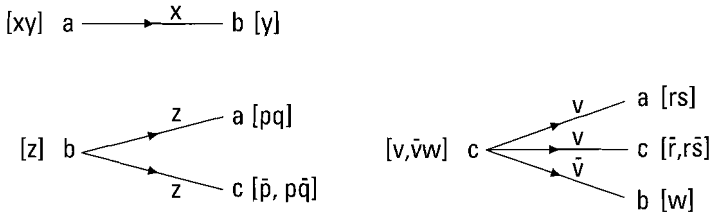

Figure 5: *The local encoder structure (1).*

defined in each state.) In that case, the cardinalities ϕ_v of these special lists will constitute an $[\mathcal{M}, |\boldsymbol{B}|]$-approximate eigenvector ϕ. Also, in each state v, the set of transitions leaving v can then be partitioned into groups, in such a way that all transitions within a group carry the same source label when entered though the same walk. (These partitions will be referred to as a *BD partition* for $\mathcal{M}$.)

Our construction method tries to reverse this process: given ϕ, we try to reconstruct, in each state v, the special lists associated with v, together with the partition of the set of transitions leaving v. An additional advantage of this approach is that if we succeed, then the BD code that we find will automatically possess some extra regularity.

Our method is best introduced by an example. In this example, the special lists will be represented with the aid of *variables* over $\boldsymbol{B}$, the source alphabet, and a relabeling will consist of a suitable substitution of source symbols for these variables.

Example 3: Consider again the FSTD $\mathcal{M}$ in Figure 2. We want to construct a BD code for $\mathcal{M}$ with source alphabet $\boldsymbol{B} = \{0, 1\}$. We first note that the weights $\phi_a = 1$, $\phi_b = 2$, and $\phi_c = 3$ constitute an $[\mathcal{M}, 2]$-approximate eigenvector. The discussion preceding this example now suggests taking look-ahead $M = 2$, and letting the lists of admissible prefixes associated with the states a, b, and c consist of one, two, and three binary words of length two, respectively. (Note that these lists may depend on the way that the corresponding state is entered.) Typically, lists associated with states a, b, and c will have the form $[xy]$, $[z] \sim [z0, z1]$, and $[v, \bar{v}w] \sim [v0, v1, \bar{v}w]$, respectively, for suitable choices of the binary variables x, y, z, v and w. (Here $\bar{v} = 1 - v$ denotes the complement of v.)

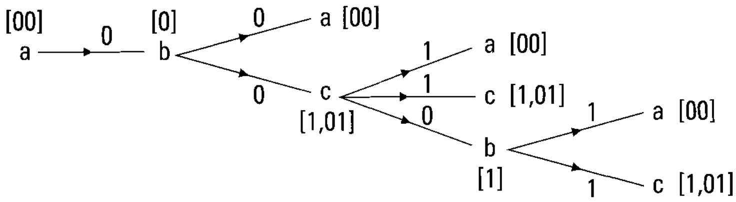

Figure 6: *Encoder construction.*

Note that we have opted for the *simplest possible structure* $[z]$ of lists associated with state b.

The first and most important step in the construction is to find in each state a partition of the outgoing transitions into groups that will always receive equal source labels. In Figure 4 we give a local description of the FSTD $\mathcal{M}$, and, in each of the states, a partition of the outgoing transitions. The indices to the state names in this figure indicate the weight of the states, a convention that will be adopted throughout this paper. Note that these partitions correspond to the form of the "local encoding rules" in Figure 5. (Here, x, y, z, p, q, r and s represent binary variables.) In fact, the choice of these partitions is immediate once we have fixed the structures of the lists of admissible prefixes as we did. Let us explain this for state c. (A similar argument applies to the other two states.) Suppose that, for a certain history in state c, the list of admissible prefixes in c is $[v, \bar{v}w]$. Then the lists associated with the terminal states of transitions leaving c and labeled with source symbol v together must contain each binary word of length two. (There are no further restrictions on upcoming sequences of source symbols that begin with a symbol v). Hence the sum of the weights of these terminal states must be at least four. Similarly, the sum of the weights of terminal states of transitions leaving c and labeled with source symbol $\bar{v}$ must be at least two. Consequently, the partition must be the one as indicated in Figure 4. A description of local encoding rules as in Figure 5 will be termed the *local encoder structure*.

Observe that all lists in Figure 5 associated with the same state are equal up to renaming variables. Therefore we might try to build an encoder based on the local encoding rules in Figure 5, substituting appropriate source symbols for the variables according to need, that is, depending on how a state is entered. (Remark that no substitution of symbols for the variables in Figure 5 gives a look-ahead encoder; in particular, no encoder exists for which the list in state b is independent of history.) This idea is further illustrated by Figure 6, where we show how the encoder is gradually developed, starting with list [00] for state a. (Read this figure from left to right.) Proceeding in this way we again obtain the BD code described earlier in Example 2. □

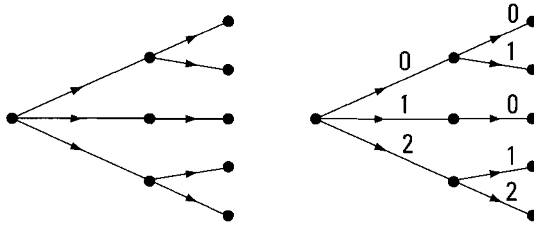

Figure 7: *The parsing tree of U.*

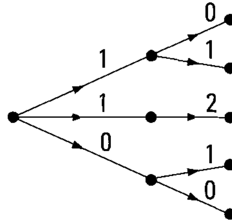

Figure 8: *A labeling of the parsing tree of U.*

After this example, the reader will surely wonder whether a valid local encoder structure as in Figure 5 can *always* be transformed into an encoder with *finite* path-memory. The answer turns out to be yes, even with path-memory $T \leq M$. Before we show this, we will present a more formal description of our method and the notions involved.

For the BD code derived in Example 3, the various lists of admissible prefixes associated with a given state may depend on history, but they all have a similar structure. The *structure* of a set of words is described by its *parsing tree*, a (directed, rooted) tree obtained by arranging these words into a tree structure.

Example 4: The parsing tree associated with the set of words

$$U = \{00, 01, 10, 21, 22\}$$

is the tree at the left in Figure 7. Indeed, if the arcs of this tree are labeled as shown at the right in Figure 7, then U is the set of words obtained by reading off the labels of paths from the root to a leaf in this tree. □

Definition 3: Let U and W be two sets of words, all of a fixed length M. We say that W is a *relabeling* of U if the arcs of the parsing tree of U can be labeled in such a way that each word in W is the label sequence of some path from the root to a leaf in this tree.

Example 5: Let U be the set of words in Example 4, and let

$$W = \{10, 11, 12, 00\}.$$

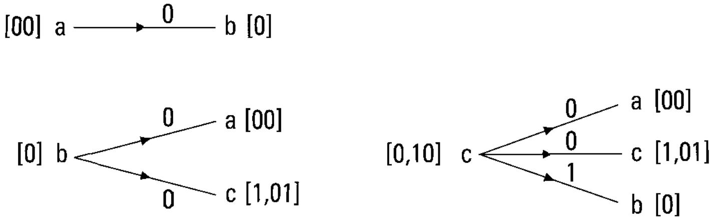

Figure 9: *The local encoder structure (2).*

Then W is a relabeling of U, as is shown by the labeling of the parsing tree of U as in Figure 8. (Note that, according to Definition 3, the word 01 need not be included in W.) □

Now we come to the formal description of a local encoder structure. To establish the relation with the ideas presented in Example 3, we first offer the following discussion. We may think of the structure of a set of words as its parsing tree, labeled with variables. Then a relabeling consists of substituting source symbols for these variables. On the other hand, a generic representation of the lists of admissible prefixes and local encoder structure (involving variables) such as in Figure 5 can be replaced by a suitable *realization* of this structure without loss of information. We illustrate this with an example.

Example 6: An alternative description of the local encoder structure in Figure 5 of Example 3 is given in Figure 9. Indeed, although this description seems less general, Figure 9 still contains all the necessary information to reconstruct the local encoder structure in the form of Figure 5.

The three lists $\mathcal{W}_a = [00]$, $\mathcal{W}_b = [0]$, and $\mathcal{W}_c = [0, 10]$ associated with the initial states a, b, and c of transitions in Figure 9 have the same structure as their generic counterparts in Example 3. We may think of the lists appearing at terminal states of transitions in Figure 9 as lists $\mathcal{W}_\alpha$ *associated with transition* α. Note that each $\mathcal{W}_\alpha$ is a relabeling of the list $\mathcal{W}_{\text{end}(\alpha)}$. The fact that Figure 9 represents "locally" valid encoding rules could be expressed in a form similar to (2). □

The above example leads to the following definition. Let $\mathcal{M}$ be an FSTD, let $\boldsymbol{B}$ denote a source alphabet, and let M be a non-negative integer. The map $\varphi : A \mapsto \boldsymbol{B}$ represents a labeling of the transitions of $\mathcal{M}$ with source symbols. Finally, let $\mathcal{W} : V \cup A \mapsto \mathcal{P}(\boldsymbol{B}^M)$ be a map that assigns to each state v and each transition α of $\mathcal{M}$ collections of words $\mathcal{W}_v$ and $\mathcal{W}_\alpha$, respectively, each consisting of words of length M over $\boldsymbol{B}$.

Definition 4: We say that a triple $[\mathcal{M}, \varphi, \mathcal{W}]$ as above constitutes a *local encoder structure* with source alphabet $\boldsymbol{B}$ and look-ahead M for $\mathcal{M}$ if

i) at least one set $\mathcal{W}_v$ is non-empty,
ii) for each transition α, the set $\mathcal{W}_\alpha$ is a relabeling of $\mathcal{W}_{\text{end}(\alpha)}$, and
iii) for each state v, we have

$$\mathcal{W}_v \boldsymbol{B} \subseteq \bigcup_{\alpha \in A_v} \varphi(\alpha)\mathcal{W}_\alpha. \tag{5}$$

The above definition should be compared with Definition 1 in the previous section. In particular, we see that if $\mathcal{W}_\alpha$ is *equal* to $\mathcal{W}_{\text{end}(\alpha)}$ for each transition α, then the local encoder structure is in fact a look-ahead encoder for $\mathcal{M}$.

We claimed earlier that a local encoder structure for $\mathcal{M}$ can be used to construct a BD code for $\mathcal{M}$. Our next theorem, which is one of the main results in this paper, states this in a precise way and, moreover, provides an upper bound on the path-memory of the resulting BD code. The proof, which can be found in Appendix A, is not difficult but involves many details. Therefore, the reader is advised to skip the proof on first reading. (The technique used in the proof to construct the desired BD code can and will be replaced by more "ad hoc" techniques in our examples.) In Appendix B, we indicate another way to obtain an encoder.

Theorem 3: Suppose that the triple $[\mathcal{M}, \varphi, \mathcal{W}]$ constitutes a local encoder structure with look-ahead M for $\mathcal{M}$. Then we can construct an M-symbol look-ahead BD code for $\mathcal{M}$, with the same source alphabet and with path-memory at most T, that is, a BD code for $\mathcal{M}$ with a path decoding window of size at most $T + 1$. Here $T \leq M$ denotes the smallest number with the property that each of the sets $\mathcal{W}_\alpha$ can be obtained from the parsing tree associated with $\mathcal{W}_v$ by relabeling up to depth T only.

In general we can obtain many different BD codes from a given local encoder structure. But of course we want to find one with smallest possible path-memory since such a code will have the smallest possible decoding window. Fortunately this optimizing problem is easy and can be dealt with by hand, or otherwise by computer. (A good illustration is provided by Example 9 in the next section.)

The construction method suggested by Definition 4 and Theorem 3 will be too general for our purposes. The problem is that we would like to construct the local encoder structure for each state *separately*, but unfortunately a particular choice for the set $\mathcal{W}_v$ in one state may affect the construction in other states. Note however that once the sets $\mathcal{W}_v$ are chosen, the requirements in Definition 4 can be verified *locally* for each state.

Therefore, with the sole exception of Example 3 in Section 8, we will restrict ourselves to the case where (the structure of) the sets $\mathcal{W}_v$ in Definition 4 is determined by the desired look-ahead M and a given approximate eigenvector ϕ, in the following way. For a given weight ϕ_v, we let $\mathcal{W}_v$ consist of the ϕ_v *smallest words*

of length M over $\boldsymbol{B}$, with respect to the *lexicographical order*. Here, we assume that the symbols of $\boldsymbol{B}$ have been ordered in some way. Then a word $x = x_1 \cdots x_M$ is smaller than a word $y = y_1 \cdots y_M$ if $x_1 \cdots x_{k-1} = y_1 \cdots y_{k-1}$ and $x_k < y_k$ holds for some k, $1 \le k \le M$. Intuitively, these sets have the "simplest" structure among all sets of words of a given size. (This structure does not depend on the chosen ordering on $\boldsymbol{B}$.) So, in Example 3 we choose $\mathcal{W}_b = \{00, 01\}$ (with structure $[z]$) and not $\{00, 11\}$ (with structure $[zx, \bar{z}y]$), given that $\phi_b = 2$ and $M = 2$. For completeness' sake we note that restricting the structure of the sets $\mathcal{W}_v$ as here does not limit the scope of our method if it is combined with state splitting, as we will do later. For example, if in Example 3 a set $\mathcal{W}_b = \{00, 11\}$ were required, then we would first split state b into two states b^0 and b^1, each of weight one. Then both $\mathcal{W}_{b^0} = \{00\}$ and $\mathcal{W}_{b^1} = \{11\}$ have the required structure. Concerning the size of the sets $|\mathcal{W}_v|$, in Theorem 4, we will show that the weights $|\mathcal{W}_v|$ obtained from a local encoder structure necessarily constitute an approximate eigenvector.

Now we turn our attention to the remaining local construction problem. So suppose that we are given a source alphabet $\boldsymbol{B}$, a number M, and an $[\mathcal{M}, |\boldsymbol{B}|]$-approximate eigenvector ϕ, and suppose furthermore that we have chosen the sets $\mathcal{W}_v$ as explained above. By construction, there is a number n_v such that the set $\mathcal{W}_v$ contains all words of length M that begin with one of the first n_v symbols from $\boldsymbol{B}$, and possibly some of the words that begin with the $(n_v + 1)$th symbol, and hence these symbols will appear as a label on the outgoing transitions in state v. Consequently, to complete the construction, the most important task that remains is to find a suitable partition of (a subset of) the outgoing transitions into n_v (or $n_v + 1$) groups of transitions that will receive the same source label. On the other hand, it is fairly easy to determine whether a given partition can indeed be completed to a local encoder structure for state v, in the sense that appropriate labels can be assigned to the groups such that the resulting labeling φ and suitable relabelings $\mathcal{W}_\alpha$ of the sets $\mathcal{W}_{\text{end}(\alpha)}$ satisfy ii) and iii) in Definition 4.

Definition 5: If a given partition of (a subset of) the outgoing transitions in state v can be completed to a local encoder structure in v in the above sense, then this partition is called a *BD partition* for state v, *relative to M and ϕ*.

It is very important to realize that whether a given partition of the outgoing transitions in a state v is indeed a BD partition for v depends only on the number M and the weights of v and its successors. Consequently, the above approach reduces the (global) code construction problem to a local one in each state, involving only the amount of look-ahead and the weights of that state and its successors.

Example 7: Consider again Example 3. The partitions given in Figure 4 are indeed BD partitions relative to $M = 2$ and weights $\phi_a = 1$, $\phi_b = 2$, and $\phi_c = 3$,

as witnessed by the local encoder structure given in Figure 9. Note that each list $\mathcal{W}_v$ in this figure consists of the ϕ_v smallest words of length 2 over $\boldsymbol{B}$ as required. □

We may even remove the dependence on M: without loss of generality we may take M to be the *smallest* number for which $\phi_v \leq |\boldsymbol{B}|^M$ holds for all states v. This can be seen as follows. First note that if we have a local encoder structure in state v with look-ahead M, then we can immediately obtain one with look-ahead $M + 1$ by defining new lists $\mathcal{W}'_v$ and $\mathcal{W}'_\alpha$ and a new labeling φ' by $\mathcal{W}'_v = 0\mathcal{W}_v$, $\mathcal{W}'_\alpha = \varphi(\alpha)\mathcal{W}_\alpha$, and $\varphi'(\alpha) = 0$, for all transitions α leaving v, where 0 denotes the smallest source symbol in $\boldsymbol{B}$. Conversely, if $\phi_v \leq |\boldsymbol{B}|^{M-1}$ holds for all states v, then, since by assumption each $\mathcal{W}_v$ consists of the ϕ_v smallest words of length M, each $\mathcal{W}_v$ will be of the form $0\mathcal{W}'_v$, and hence each $\mathcal{W}_\alpha$ will be of the form $\varphi'(\alpha)\mathcal{W}'_\alpha$, for some source symbol $\varphi'(\alpha)$. It is then easily checked that $[\mathcal{M}, \mathcal{W}', \varphi']$ again constitutes a local encoder structure, but now with look-ahead $M - 1$. (Further details are left to the reader.)

Our next theorem states some necessary conditions that a BD partition (or, more generally, the partition induced by the labeling φ of a local encoder structure) has to satisfy. The first part of this theorem generalizes Theorem 1 and explains our restriction imposed on the cardinality of the sets $\mathcal{W}_v$.

So let $[\mathcal{M}, \varphi, \mathcal{W}]$ denotes a local encoder structure with look-ahead M and source alphabet $\boldsymbol{B}$. We denote by $A_v = \cup_{b\in \boldsymbol{B}} A_v^b$ the partition of the set of outgoing transitions A_v in state v *induced* by φ; here A_v^b consists of the transitions α in A_v for which $\varphi(\alpha) = b$. (Note that some of the parts may be empty.) Let ϕ_v denote the number of words $|\mathcal{W}_v|$ contained in $\mathcal{W}_v$.

Theorem 4: With the above notation, ϕ is an $[\mathcal{M}, N]$-approximate eigenvector, where $N = |\boldsymbol{B}|$ denotes the cardinality of $\boldsymbol{B}$. Moreover, for each state v of $\mathcal{M}$ we have

$$\phi_v \leq \sum_{b\in \boldsymbol{B}} \min\left(\lfloor N^{-1} \sum_{\alpha\in A_v^b} \phi_{\mathrm{end}(\alpha)} \rfloor, N^{M-1} \right). \tag{6}$$

In particular, if $\mathcal{W}_v$ consists of the ϕ_v smallest words in $\boldsymbol{B}^M$, then the number ϕ_v^b of words in $\mathcal{W}_v$ beginning with symbol b satisfies

$$\phi_v^b \leq \lfloor N^{-1} \sum_{\alpha\in A_v^b} \phi_{\mathrm{end}(\alpha)} \rfloor. \tag{7}$$

The proof of this theorem (which is similar to the proof of Theorem 1) together with a discussion of a nice corollary can be found in Appendix A.

Our methods may be summarized as follows.

Construction method for BD codes:
Given an FSTD $\mathcal{M}$ and an $[\mathcal{M}, |\boldsymbol{B}|]$-approximate eigenvector ϕ, let M be the smallest number such that $\phi_v \leq |\boldsymbol{B}|^M$ holds for each state v of $\mathcal{M}$. Furthermore, for each state v, let $\mathcal{W}_v$ consist of the ϕ_v smallest words of length M over $\boldsymbol{B}$. Then, *for each state v separately*, try to find a BD partition of the set of transitions leaving state v, and construct the corresponding local encoder structure in this state. This last step requires that we find, for each transition α leaving v, a suitable relabeling of the set of words $\mathcal{W}_{\text{end}(\alpha)}$ and a suitable source symbol $\varphi(\alpha)$, where the partition induced by φ is precisely the BD partition. If a local encoder structure is found in each state, then Theorem 3 guarantees the existence of a BD code with look-ahead M and path-memory at most M.

Given an FSTD $\mathcal{M}$ and an $[\mathcal{M}, |\boldsymbol{B}|]$-approximate eigenvector ϕ, how difficult is it to determine whether a BD partition w.r.t. ϕ exists? This problem is discussed in detail in Appendix A. It turns out that the worst-case complexity of this problem is (at least) exponential in $d_{\max}$, the maximum number of transitions leaving a state. The reader should not worry too much about this exponential behaviour. Indeed, note that even the complexity of finding a partition of A_v satisfying the necessary condition (7) in Theorem 4 is already exponential in $|A_v|$. Nevertheless, the reader will surely agree that as long as $|A_v|$ is not too big, this last problem can easily be handled. (Here we see the reason why the desired rate p/q of the code should have small q.) For small M, say $M \leq 3$, finding a BD partition turns out to be comparable in difficulty to this last problem, and, according to our experience, is easy for values of $d_{\max}$ *at least* up to 10. This conclusion is supported by the examples in Section 8.

4.5 A general construction method

In this section we supplement the methods in the previous section to obtain a general construction method for sliding-block decodable codes for a constrained system of finite type. The many examples that we will give, both in this section and in Section 4.7, show that the method works very well and enables the design by hand of "good" codes, that is, codes with *small* decoding windows and simple encoders, even for rather complicated constraints. Some of the reasons *why* this method works so well are discussed in Appendix B.

To understand our method, the reader should have some working knowledge of state splitting. If this is not the case, we suggest consulting Appendix B or [19]. In what follows, we assume that the constraints are represented by a given FSTD $\mathcal{M}$ of finite type. We also assume that we are given an $[\mathcal{M}, N]$-approximate eigenvector ϕ, where N denotes the desired size of the source alpha-

bet $\boldsymbol{B}$. (Possible necessary preparations are discussed in Section 2. A simple algorithm to compute such an approximate eigenvector has been given by Franaszek, and can be found for example in [2].)

In the previous section we showed that a BD code for $\mathcal{M}$ can be obtained once a BD partition is found in each state. However, there may be some states for which no BD partition exists. For example, let $N = 2$, and suppose that $\mathcal{M}$ contains a state v with precisely two successor states a and b. If $M = 2$ and if v, a, and b all have weight 3, then no BD partition in state v exists. (Note also that increasing the value of the allowed look-ahead M does not solve the problem.) However, if one of the successors of v could be split into a state of weight 1 and another one of weight 2, then after this state splitting operation a BD partition for v would exist. (The reader should verify this! Precisely this situation will also be discussed in Example 8 below.) This simple example suggests the use of state splitting operations to transform the weighted FSTD $\mathcal{M}$ until our local construction method applies, and leads to the following three-stage code construction scheme.

In stage *one*, we transform the pair $\mathcal{M}$, ϕ into a new FSTD $\mathcal{M}'$ and a new $[\mathcal{M}', N]$-approximate eigenvector ϕ' by *state splitting*. Here, we try to minimize the number K of rounds of state splitting needed to make the second stage succeed. (Note that each additional round of state splitting will increase the size of the decoding window by one symbol).

In the *second* stage, we construct a local encoder structure for $\mathcal{M}'$ from a BD partition with respect to ϕ' in each state of $\mathcal{M}'$.

In the *third* and final stage, we use this local encoder structure for $\mathcal{M}'$ to construct a BD code for $\mathcal{M}'$; in fact, one that provides a code for the constrained system $\mathcal{L}$ generated by $\mathcal{M}$ with *smallest* possible decoding window.

The third stage of the above scheme is easy and could be handled by computer if desired. (This will be further illustrated by our examples.) The main difficulty of our approach is to "coordinate" the first two stages. Unfortunately, it is not easy to formulate strict rules on how to choose between various possible state splitting steps. However, we will see from our examples that for many practical applications the above guideline is sufficient. Since this scheme obviously extends the ACH-algorithm [2], we are guaranteed to find *some* code.

A nice feature of the above approach is that code-construction methods such as the ACH-algorithm [2], the BD-method of Franaszek [8], ideas described in [17], variable-length codes, and variable-length state splitting as in [3], [11] can all be thought of as special cases of the above scheme. In particular, the ideas from [17] become clearer when considered in this light. We will illustrate this with an example.

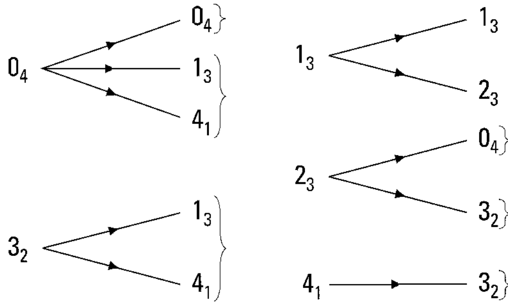

Figure 10: *The local description of $\mathcal{M}$.*

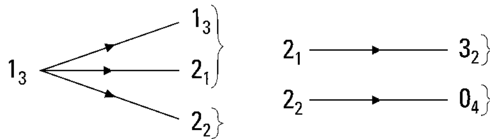

Figure 11: *The BD partitions in the remaining states.*

Example 8: Let $\mathcal{M}$ be the five-state FSTD with local description as in Figure 10. (This is the same FSTD as the one considered in Example 2 from [17].) Also shown in this figure are the weights (indicated as a subscript to the state numbers) that constitute an $[\mathcal{M}, 2]$-approximate eigenvector, and a BD partition in all states except state 1, for which no such partition exists. (Here, we take $M = 2$.) Therefore, in stage one we *must* split a successor of state 1, hence we must split state 2, and the only possibility is to split state 2 into states 2_1 and 2_2, of weight 1 and 2, respectively. Let $\mathcal{M}'$ denote the FSTD thus obtained. In Figure 11 we show the result of the state split, as well as a BD partition for states 1, 2_1 and 2_2. Since we have now found a BD partition in each state of $\mathcal{M}$, we can construct a BD code for $\mathcal{M}$ with source alphabet $B = \{0, 1\}$ of size two. Stage three is trivial here, since it is evident that we can even obtain a two-symbol look-ahead encoder based on $\mathcal{M}'$, as shown in Figure 12. The resulting code for $\mathcal{M}$ has a (path-)decoding window of size 1 (actual transition) + 1 (the split) = 2. □

We remark that the other examples in [17] can be handled in a similar way.

Up to now, we have illustrated the theory with "artificial" examples only. Our next example changes that situation; we now construct a code which has considerable practical applications.

Example 9: In this example, we will construct a rate 1/2 $(2, 7)$-code with a decoding window of size four code words or eight bits. (Recall that a (d, k)-constrained sequence is a binary sequence in which each run of zeros has a length

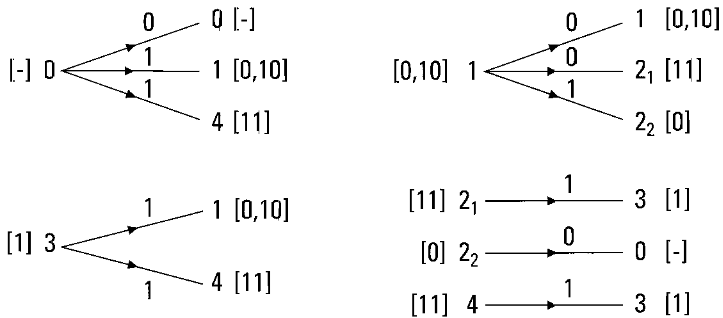

Figure 12: *A two-symbol look-ahead encoder based on $\mathcal{M}'$.*

of at least d and at most k. Here we consider the case where $d = 2$ and $k = 7$.) In [2], a rate $1/2$ code for this constraint is constructed to illustrate the ACH state splitting algorithm. We will do likewise here.

The FSTD commonly used to generate $(2, 7)$-constrained sequences has eight states numbered $0, 1, \ldots, 7$, where the numbering is such that a sequence generated by a walk ending in state v ends in precisely v zeros. Since we aim for a rate $1/2$ code for this constraint, we let $\mathcal{M}$ be the second power of this FSTD (so $\mathcal{M}$ is labeled with two-bit code words), and we take the source alphabet $\boldsymbol{B} = \{0, 1\}$. The local structure of $\mathcal{M}$ is given in Figure 13. The smallest $[\mathcal{M}, 2]$-approximate eigenvector involves weights between 1 and 4, so we choose $M = 2$. (The weights of the states are indicated in Figure 13 as subscripts to the state numbers.)

For all states except states 2 and 3 a BD partition can be found, as indicated in Figure 13. (For state 6, we may choose either one of the indicated parts.)

Since no BD partition for states 2 and 3 exists, some state splitting is required. Inspection of Figure 13 shows that we have to split either both states 4 and 5, or state 1. We choose to split state 1, into a state 1_1 with weight one and single successor state 0, and state 1_2 with weight two and single successor state 3. (Splitting state 1 seems appropriate in view of the successor situation at state 7. However, the other choice also leads to a code with an eight-bit decoding window. This code, whose derivation we leave to the reader, is only slightly less regular than the code obtained below. Yet another approach would be to split all of states 1, 4, and 5. This would of course complicate stages two and three since more BD partitions have to be considered, but even by hand these complications could still be dealt with easily.)

After splitting state 1, a BD partition can be found in each state. (In fact, in most states we need to consider *several* BD partitions. In what follows we will see how to handle this freedom of choice.) So we can now move to stage three of

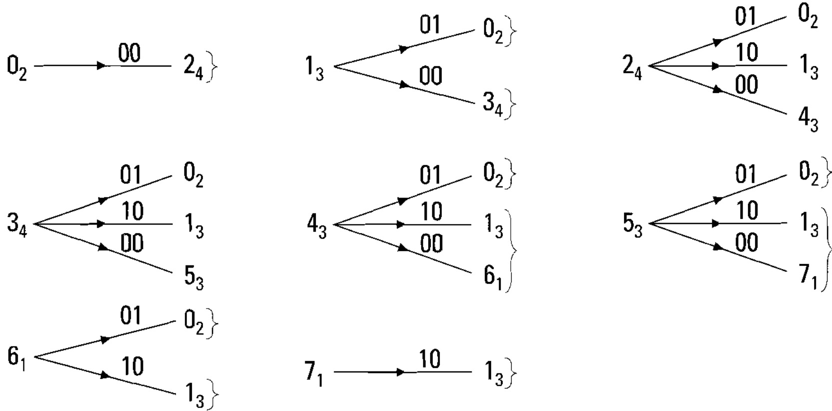

Figure 13: *The local structure of $\mathcal{M}$.*

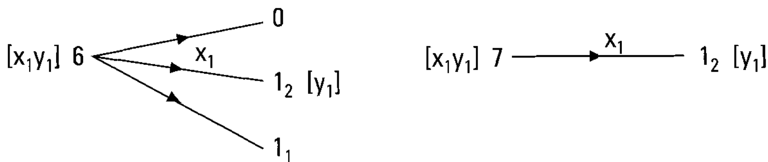

Figure 14: *The local encoder structure in states 6 and 7.*

our method. We will aim for a decoding window of only four code words. Since we did only one round of state splitting (that is, we have $K = 1$), this means aiming for a "look-back" part of the decoding window of size $D = 2$ code words only. Aiming for a certain value of D is of great help in stage three of our method, as will be seen below. (Moreover, in other cases it may help to eliminate certain BD partitions and state splits that would have to be considered otherwise.) Note that by looking back four bits only, we cannot distinguish states $v \geq 1$ according to their history, and we can only distinguish between state 0 entered from state 1 (each history results in a label sequence $\cdots 1001$) and state 0 entered from a state different from 1 (which produces $\cdots 0001$). Moreover, we cannot distinguish between states 4, 5, 6 and 7 (all these states are entered with label sequence $\cdots 0000$), so transitions leaving one of these states that are labeled with the same (two-bit) code word must receive the same label from $\boldsymbol{B}$. (We will sometimes express this last observation by saying that *consistent labeling* is required for these states.) So in states 6 and 7, we necessarily have a local encoder structure as shown in Figure 14. (In this figure, x_1 and y_1 are both binary variables. Later developments may put further constraints on the values that x_1 and y_1 may attain.) Note that, in accordance with the remarks in the preceding paragraph, we

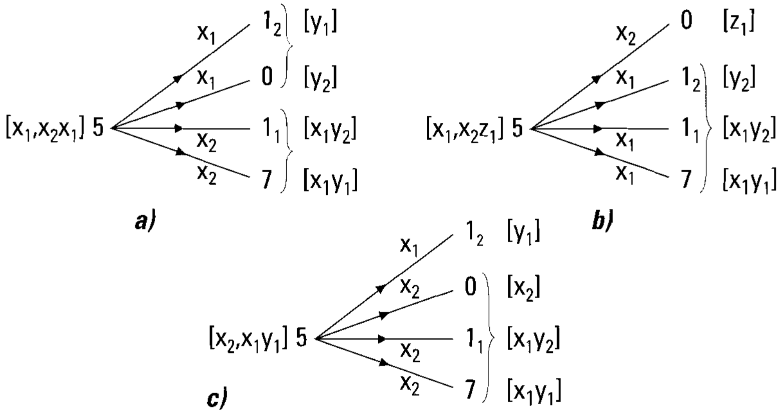

Figure 15: *The possible encoder structures in state 5.*

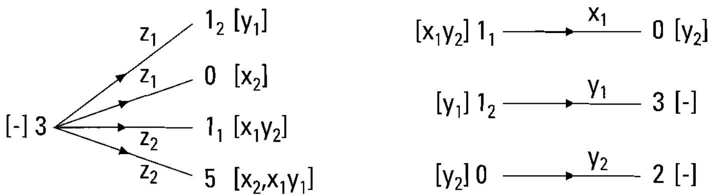

Figure 16: *The encoder structures in the remaining states.*

have assigned the *same* prefix list $[y_1]$ to both state 1_2 entered from state 7 and state 1_2 entered from state 6 (each history of state 1_2 produces 0010, irrespective of the particular state from which 1_2 is entered). Also, we have labeled both the transitions from state 6 to 1_2 and from state 7 to 1_2 with the *same* source symbol x_1, since both these transitions have code word label 10 and states 6 and 7 cannot be distinguished (both states are invariably entered with 0000). We will implicitly use a similar reasoning when we develop the local encoder structures in the remaining states.

There are three possible BD partitions in state 5, which lead to the three possible local encoder structures in Figure 15. Here, in all three cases, we require that $\{x_1, x_2\} = \{y_1, y_2\} = \{0, 1\}$. To each of the three local encoder structures in state 5, there corresponds a similar local encoder structure in state 4. There is a unique BD partition in state 3, in which one part consists of the transitions to states 1_2 and 0, and the other part consists of the transitions to states 5 and 1_1.

The local encoder structures a) and b) in Figure 15 in state 5 assign sets of words to states 1_1 and 5 that cannot be combined. Therefore in state 5 we must take the local encoder structure c) in Figure 15. This choice determines the local

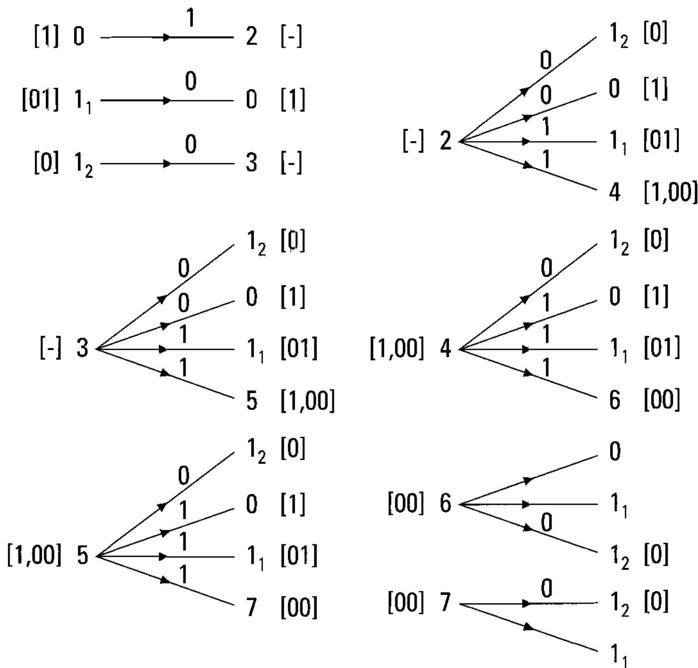

Figure 17: *A two-symbol look-ahead encoder.*

encoder structure in the remaining states as indicated in Figure 16. (The local encoder structure for state 4 is similar to the one for state 3.) Remark that the local encoder structure in state 3 imposes the constraint that $x_1 = y_1$. Finally, if we take for example $x_1 = 0 = z_1$, then we find the two-symbol look-ahead encoder in Figure 17, which indeed constitutes a rate $1/2$ $(2, 7)$-code with a decoding window of size four symbols or eight bits. We will show in Example 11 that this code has a six-state encoder graph. (The code derived in [2] has the same decoding window size but requires a seven-state encoder.) Our code is actually equivalent to one of the codes that can be obtained from the uniquely decipherable prefix list in Table II of [2], and is among the best codes known for this constraint. Observe that our method produces this code almost automatically. □

We end this section with some remarks on the problem of how to choose the state splitting steps in the first stage. One approach (which is consistent with our examples) would be to choose only state splitting steps that do not destroy BD partitions (or "partial" BD partitions) constructed earlier. Let us call such state splits *nice*. First note that we certainly obtain a ae-consistent state split with respect to the approximate eigenvector ϕ (see [19] or Appendix B) in a state if we let the successors of one of the two offsprings consist of the union of one or more

of the parts of the partial BD partition in that state. Of course, splitting a state in such a way does not destroy the partial BD partition in that state.

We mention here without proof that an ae-consistent state split with respect to ϕ of the above type in a state v of weight $\phi_v \leq N^M$ is nice if, moreover, one of the two offsprings of v has a weight ν satisfying $1 \leq \nu \leq N^k$ or $\nu \equiv \phi_v \bmod N^l$, where N^k is the highest power of N dividing ϕ_v and $N^{l-1} < \nu \leq N^l$. (The reason is that under these conditions the set of the ν smallest words of length M over $\boldsymbol{B}$ has the same structure as the set consisting of the ν largest words in $\mathcal{W}_v$, the set of the ϕ_v smallest words of length M over $\boldsymbol{B}$.)

Unfortunately, this approach works often, but not always; it is possible to construct examples where no nice state split exists. Nevertheless, a useful heuristic is to use nice state splits whenever this is possible.

We believe that the method in its present form is best implemented *interactively*; the user chooses state splitting steps and BD partitions from alternatives proposed by the program, but leaves further calculations as well as the optimizing in stage three to the computer.

In the next section we discuss how to design finite-state encoders for this type of code. Further applications of our code-construction method will be given in Section 4.7.

4.6 Finite-state encoders from look-ahead encoders

We now investigate how to obtain an encoder graph, and hence an ordinary finite-state encoder, from a given look-ahead encoder. Since a BD code is a special look-ahead encoder, the construction below also applies to BD codes.

So let $[\mathcal{M}, \varphi, \mathcal{W}]$ be an M-symbol look-ahead encoder, with source alphabet $\boldsymbol{B}$. We will think of an internal state of the encoder as being determined by a pair $(v, \boldsymbol{b})$, consisting of a state v of $\mathcal{M}$ together with a word $\boldsymbol{b} = b_1 \cdots b_M$ in $\mathcal{W}_v$ obtained by looking ahead at upcoming source symbols. Then, to encode b_1, the encoder first looks ahead one further symbol to see $b = b_{M+1}$, then picks a transition α leaving v for which $\varphi(\alpha) = b_1$ and $\boldsymbol{b}' = b_2 \cdots b_M b \in \mathcal{W}_{\mathrm{end}(\alpha)}$; as a result the encoder now moves to the internal state determined by $(\mathrm{end}(\alpha), \boldsymbol{b}')$. We may think of this internal transition as being labeled with source/transition label b/α.

In what follows, we will find it convenient to think of the encoder as moving from a set of encoding alternatives to another such set. To this end, we associate with each pair $(v, \boldsymbol{b})$ the collection $\Sigma_{v,\boldsymbol{b}}$ consisting of all walks $\boldsymbol{\alpha} = \alpha_1 \cdots \alpha_M$ for which $\varphi(\boldsymbol{\alpha}) = \varphi(\alpha_1) \cdots \varphi(\alpha_M)$ equals $\boldsymbol{b}$. So $\Sigma_{v,\boldsymbol{b}}$ consists of all possible encoding paths starting in v that encode the word $\boldsymbol{b}$. We will refer to these collection of

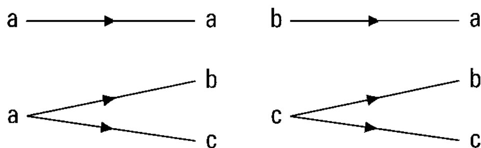

Figure 18: *The collections* $\Sigma_{v,\boldsymbol{b}}$.

walks $\Sigma_{v,\boldsymbol{b}}$ as *state trees*. (In [9], they are called *independent path-sets* or IP's.)

We now construct an FSTD $\mathcal{G}$ as follows. The states of $\mathcal{G}$ are the state trees $\Sigma_{v,\boldsymbol{b}}$, with v a state of $\mathcal{M}$ and $\boldsymbol{b} = b_1 \cdots b_M \in \mathcal{W}_v$. Furthermore, for each such state $\Sigma_{v,\boldsymbol{b}}$ and for each $b \in \boldsymbol{B}$, let $\mathcal{G}$ have a labeled transition

$$\Sigma_{v,\boldsymbol{b}} \xrightarrow{b/\alpha} \Sigma_{\mathrm{end}(\alpha),\boldsymbol{b}'}, \tag{8}$$

where α is a transition in $\mathcal{M}$ leaving v with $\varphi(\alpha) = b_1$ for which $\boldsymbol{b}' = b_2 \cdots b_M b$ is contained in $\mathcal{W}_{\mathrm{end}(\alpha)}$. Then the earlier discussion shows that the following holds.

Theorem 5: The FSTD $\mathcal{G}$ is an encoder graph for the code with look-ahead encoder $[\mathcal{M}, \varphi, \mathcal{W}]$.

Note that an encoder graph for the constrained system $\mathcal{L}$ generated by $\mathcal{M}$ can be obtained by replacing each label b/α in (8) by the source/code-word label $b/L(\alpha)$. We will refer to the encoder graph thus obtained as $\mathcal{G}_L$. Note also that the above construction replaces look-ahead by encoding delay; the encoding walks produced by the two encoders will be equal up to a shift over M symbols.

We now offer a different interpretation of the above construction. In fact, what happens in Theorem 5 is that each state v of $\mathcal{M}$ is subdivided into states $v^{\boldsymbol{b}}$, $\boldsymbol{b} \in \mathcal{W}_v$, according to different futures after state v (represented by the sets $\Sigma_{v,\boldsymbol{b}}$). Moreover, since M future transitions are taken into consideration, this subdivision would require M rounds of state splitting. (Further details can be found in Appendix B.)

As one of the referees pointed out, Theorem 5 is related to the recoding of right closing maps to right resolving maps [5].

Example 10: Let $[\mathcal{M}, \varphi, \mathcal{W}]$ be the one-symbol look-ahead encoder described in Example 1, Figure 1. The four collections $\Sigma_{v,\boldsymbol{b}}$ are depicted in Figure 18. When we construct the encoder graph $\mathcal{G}$ as in Theorem 5, we obtain the four-state encoder graph in Figure 19. (In this figure, the four states a_0, a_1, b, and c correspond to the four state trees $\Sigma_{a,0}$, $\Sigma_{a,1}$, $\Sigma_{b,0}$, and $\Sigma_{c,1}$.)

This encoder can also be constructed as follows. We first subdivide state a into states a_0 and a_1, where a_1 is used to encode when the upcoming source symbol

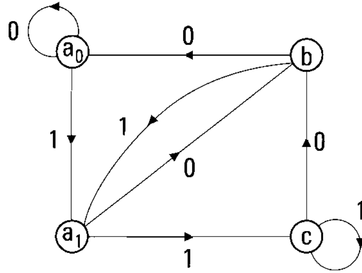

Figure 19: *A finite-state encoder.*

equals 1 (so a_1 will have successors b and c), and a_0 is used otherwise (so a_0 will have successors a_0 and a_1). As a result each of the states a_0, a_1, b, and c now encodes precisely one source symbol, which appears on all transitions leaving the state. Next, we shift back all source labels by one transition: the (common) source label on the transitions *leaving* a state is now assigned to all transitions *entering* that state. The result is the encoder graph presented in Figure 19. Note that this construction is in fact the same as the construction in Theorem 5, and produces the same encoder graph.

The codes produced by the encoders in Figure 1 and Figure 19 are indeed equal up to a shift over one symbol. For example, to encode the source sequence 0110010 from state a the encoder in Figure 1 produces the encoding path $aacbaaba$, and the encoder in Figure 19, starting in state a_0, encodes the source sequence 110010 (the previous sequence with the first symbol dropped) into the corresponding path $a_0a_1cba_0a_1b$.

To fully appreciate our final remark, the reader should probably be somewhat familiar with state splitting. Observe that the encoder in Figure 19, being obtained from the FSTD in Figure 1 by one round of state splitting, has a path-decoding window of size *two*; beside the present transition in Figure 1, the decoder needs to know (at least when this transition ends in a) one future transition to determine the corresponding transition in Figure 1. However, the reader can easily verify that knowledge of the *future transition only* is sufficient for decoding! Consequently, the *"effective"* path-decoding window size is only *one* instead of two. (This is of course evident in view of the equivalence of the two codes.) Part of this paper may be seen as an effort to profit from such "reduced window" phenomena in a systematic way. □

Both encoder graphs $\mathcal{G}$ and $\mathcal{G}_L$ will have $\sum_{v \in V} |\mathcal{W}_v|$ states, which can be large. Fortunately, the number of states of $\mathcal{G}_L$ can usually be reduced by state merging (cf., Section 4.2). This operation applies for example when the outgoing transitions in two states can be paired off in such a way that the two transitions

Table 1: *The encoder.*

present state	source symbol input 0: output	0: next state	1: output	1: next state
A	10	E	10	F
B	01	E	01	F
C	00	A	10	B
D	00	C	00	B
E	00	A	00	B
F	00	C	00	D

in each pair have the same source/code-word label and end in the same state. If this is the case, then each source sequence will be encoded from both states by the *same* code word sequence. Consequently, these states can be identified, that is, *merged* into one state. This operation is applied repeatedly until no more states can be identified. (In fact, this approach ensures that all states that can be merged will be identified eventually; since $\mathcal{G}_L$ is deterministic w.r.t. the source/code-word labeling, it is reduced in this way to its Shannon cover, see e.g. [2] or [19].)

Example 11: In this example, we consider the construction of an encoder graph for the $(2, 7)$-code obtained in Example 9. Only the main points are discussed here, the details of the construction are left to the reader.

Since there are 21 distinct pairs $(v, \boldsymbol{b})$ of states v and words $\boldsymbol{b}$ in $\mathcal{W}_v$, application of Theorem 5 leads to an encoder graph with 21 states. However, due to the regularity of the code, the number of states can be reduced to six by merging. Indeed, the reader may verify that
i) States $\Sigma_{2,00}$, $\Sigma_{3,00}$, $\Sigma_{4,00}$, $\Sigma_{5,00}$, $\Sigma_{6,00}$, and $\Sigma_{7,00}$ merge into one state A,
ii) states $\Sigma_{1_1,01}$, $\Sigma_{2,01}$, $\Sigma_{3,01}$, $\Sigma_{4,11}$, and $\Sigma_{5,11}$ can be merged into one state B,
iii) states $\Sigma_{2,10}$, $\Sigma_{3,10}$, $\Sigma_{4,10}$, and $\Sigma_{5,10}$ can be merged into one state C,
iv) states $\Sigma_{2,11}$ and $\Sigma_{3,11}$ can be merged into one state D,
v) states $\Sigma_{0,10}$ and $\Sigma_{1_2,00}$ can be merged into one state E, and
vi) states $\Sigma_{0,11}$ and $\Sigma_{1_2,01}$ can be merged into one state F.
These merging operations lead to the six-state encoder with input/output relations as in Table 1. □

In the remainder of this section we discuss a (partial) converse to Theorem 5. Our results will establish a further link with the work in [9], and will be used in Appendix B to investigate codes produced by the ACH state splitting algorithm. Suppose that, in each state v of an FSTD $\mathcal{M}$, the set of walks of length M starting in v is partitioned into mutually disjoint collections $\Sigma_{v,i}$, $i = 1, \ldots, n_v$, again

referred to as state trees. We write

$$\Sigma_{v,i} \xrightarrow{\alpha} \Sigma_{w,j} \tag{9}$$

if α is a transition in $\mathcal{M}$ from v to w such that for each walk $\alpha_1 \cdots \alpha_M$ in $\Sigma_{w,j}$, the walk $\alpha\alpha_1 \cdots \alpha_{M-1}$ is contained in $\Sigma_{v,i}$. We construct a graph $\mathcal{G}$ with as "states" the state trees $\Sigma_{v,i}$, and with "transitions" of the form (9). Let $\boldsymbol{B}$ be a set of size N and suppose that each state $\Sigma_{v,i}$ of $\mathcal{G}$ has at least N successors. Then, in each state, we can label N transitions leaving that state with the N distinct symbols from $\boldsymbol{B}$. So $\mathcal{G}$ now has labeled transitions

$$\Sigma_{v,i} \xrightarrow{b/\alpha} \Sigma_{w,j}, \tag{10}$$

and looks like an encoder graph.

Observe that walks in $\mathcal{G}$ generate walks in $\mathcal{M}$, and, because the state trees are disjoint, distinct right-infinite walks in $\mathcal{G}$ generate distinct right-infinite walks in $\mathcal{M}$. So we conclude that $\mathcal{G}$ is the encoder graph of a code for $\mathcal{M}$. Note that we may obtain such a graph $\mathcal{G}$ from a BD code $[\mathcal{M}, \varphi, \mathcal{W}]$ with path memory 0, as in Theorem 5.

What kind of code is generated by the encoder graph $\mathcal{G}$? We will show that in fact such a code is a BD code, although not necessarily one with path memory 0. To state the precise result, we need some preparation. We define a (partial) labeling φ of the transitions in the Mth history graph $\mathcal{M}_M$ of $\mathcal{M}$ (that is, of the walks of length $M+1$ in $\mathcal{M}$) as follows. Let $\varphi(\alpha_0 \cdots \alpha_M) = b$ whenever $\alpha_0 \cdots \alpha_{M-1} \in \Sigma_{v,i}$, $\alpha_1 \cdots \alpha_M \in \Sigma_{w,j}$, and $\Sigma_{v,i} \xrightarrow{b/\alpha_0} \Sigma_{w,j}$ holds for some i and j, where $v = \text{beg}(\alpha_0)$ and $w = \text{end}(\alpha_0)$. (Since the state trees are disjoint, i and j are determined uniquely.) Note that then $\varphi(\alpha_0\beta_1 \cdots \beta_M) = b$ holds for *all* walks $\beta_1 \cdots \beta_M$ in $\Sigma_{w,j}$. Now, for each walk $\boldsymbol{\alpha} = \alpha_0 \cdots \alpha_{M-1}$ in $\mathcal{M}$ (that is, for each state $\boldsymbol{\alpha}$ of $\mathcal{M}_M$), let the set $\mathcal{W}_{\boldsymbol{\alpha}}$ consist of all words $b_0 \cdots b_{M-1}$ for which there is a walk in $\mathcal{G}$ of the form

$$\Sigma_{v_0,i_0} \xrightarrow{b_0/\alpha_0} \Sigma_{v_1,i_1} \xrightarrow{b_1/\alpha_1} \cdots \xrightarrow{b_{M-1}/\alpha_{M-1}} \Sigma_{v_M,i_M}. \tag{11}$$

With these definitions of φ and $\mathcal{W}$, we have the following result.

Theorem 6: The triple $[\mathcal{M}_M, \varphi, \mathcal{W}]$ constitutes an M-symbol look-ahead BD code for $\mathcal{M}$ with path memory (at most) M. This BD code is equal to the code generated by $\mathcal{G}$ up to a shift over M symbols.

Proof: Let $\boldsymbol{\alpha} = \alpha_0 \cdots \alpha_{M-1}$ be a state in $\mathcal{M}_M$, let $b_0 \cdots b_{M-1}$ be a word in $\mathcal{W}_{\boldsymbol{\alpha}}$, and let $b \in \boldsymbol{B}$. To prove the first part of our theorem, we need to show that there is a transition in $\mathcal{M}_M$, with source label b_0, from $\boldsymbol{\alpha}$ to some other state $\boldsymbol{\alpha}'$,

Table 2: *The admissible words of length 5.*

label	word	term. state	parity init. state
a	00000	(+5)	-
b	00001	0	e
c	00010	1	o
d	00100	2	e
e	01000	3	o
f	01001	0	o
g	10000	4	e
h	10010	1	e

Table 3: *The approximate eigenvector.*

v	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
ϕ_v	7	9	12	9	12	9	11	8	11	8	11	4	7	4	7	4	0	0	0

say, such that the word $b_1 \cdots b_{M-1}b$ is contained in $\mathcal{W}_{\alpha'}$. This can be seen as follows. By the definition of $\mathcal{W}_\alpha$, there is a walk as in (11) in $\mathcal{G}$. Since each state in $\mathcal{G}$ has a leaving transition with source label b, there is a transition in $\mathcal{G}$ of the form $\Sigma_{v_M,i_M} \xrightarrow{b/\alpha} \Sigma_{w,j}$. Let $\alpha' = \alpha_1 \cdots \alpha_{M-1}\alpha$. By definition of $\mathcal{W}_{\alpha'}$, this set contains the word $b_1 \cdots b_{M-1}b$. Since necessarily $\alpha \in \Sigma_{v_0,i_0}$ and $\alpha' \in \Sigma_{v_1,i_1}$, it follows from the definition of φ that the transition $\alpha_0 \cdots \alpha_{M-1}\alpha$ in $\mathcal{M}_M$ leaving state α has source label $\varphi(\alpha_0 \cdots \alpha_{M-1}\alpha) = b_0$ as required.

To prove the second part of our theorem, we reason as follows. Suppose that $\mathcal{G}$ encodes a sequence $\{b_n\}$ of source symbols into a walk $\{\alpha_n\}$ in $\mathcal{M}$. From the definition of φ we then conclude that $\varphi(\alpha_n \cdots \alpha_{n+M}) = b_n$, that is, in the BD code the source label b_n is assigned to transition α_{n+M} (provided its history is $\alpha_n \cdots \alpha_{n+M-1}$). □

4.7 Further examples

In this section we use our construction method to derive three new codes, all with a smaller decoding window than the best codes known for the constraints involved.

4.7.1 A new rate-$2/5$ $(2, 18, 2)$-constrained code

Our first example concerns the construction of a rate-$2/5$ $(2, 18, 2)$-constrained code with a decoding window size of 3 code words or 15 bits, and error propagation of 5 bits.

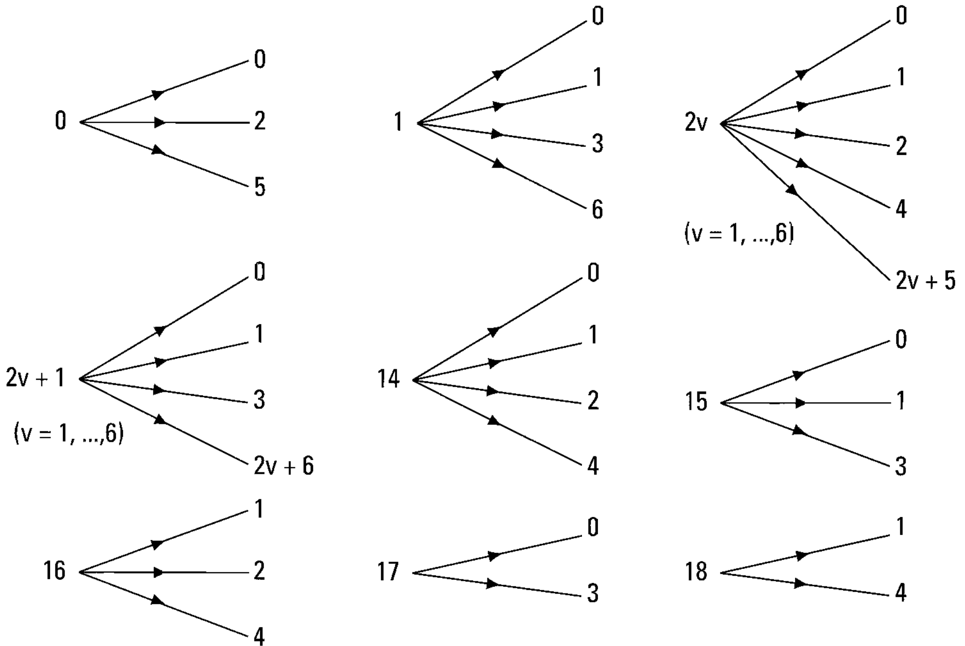

Figure 20: *The local structure of the FSTD.*

A binary sequence satisfies the $(2, 18, 2)$ constraint if any two consecutive ones are separated by an *even* number of at least 2 and at most 18 zeros. (So $0 \leftrightarrow 00$ sets up a one-to-one correspondence between $(1, 9)$-constrained sequences and $(2, 18, 2)$-constrained sequences.)

The FSTD commonly used to represent this constraint has 19 states, which are numbered $0, 1, \ldots, 18$. Here, the numbering is such that walks ending in state v generate sequences ending in precisely v zeroes. Since we consider rate-$2/5$ codes, we take $\boldsymbol{B} = \{0, 1, 2, 3\}$ (so $N = 4$) and we label the transitions with admissible words of length 5. There are eight such words, shown in Table 2. From this table we see that the label on a given transition is determined by its terminal state only (if its state number is 2 or more), or by its terminal state together with the parity (even or odd) of its initial state (if its state number is 0 or 1). For this reason we do not always indicate the code word labels on transitions in what follows. The local structure of this FSTD is described in Figure 20.

In Table 3 we list a collection of state-weights that constitute an approximate eigenvector. As suggested by the values of these weights, we work with a look-ahead $M = 2$. Since states 16, 17, and 18 have weight 0, they may be dropped. The resulting FSTD $\mathcal{M}$, shown in Figure 21, generates a subsystem of the original $(2, 18, 2)$-constrained system. (The labels in this figure refer to the 5-bit words

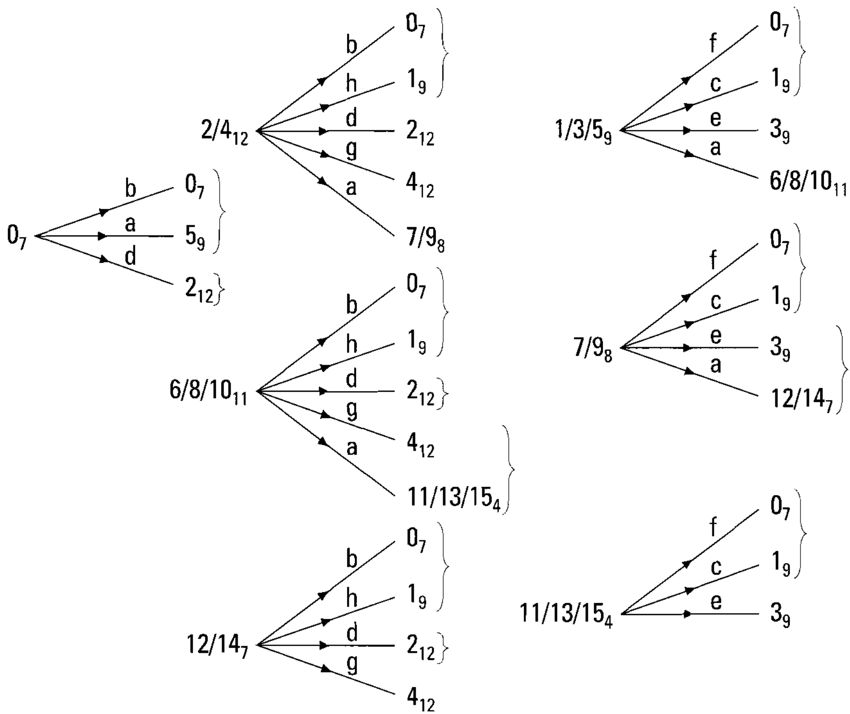

Figure 21: *The FSTD $\mathcal{M}$.*

in Table 2.) As usual, the weights of the states are represented as indices to the state-number. We have also indicated a BD partition for those states that have one. (Unique only for state 0.)

We see that all states except 2, 4 and 1, 3, 5 have a BD partition. Some state splitting on the successors of these states is needed, but as we will see, one round will do. So we will choose $K = 1$. A lot of freedom remains, though, both in the choice of state splitting operations and in choosing the BD partitions. One possibility is to split all relevant successors of states 2, 4 and 1, 3, 5 (that is, states 2, 3, 4, states 7, 9, and states 6, 8, 10), and then to use a computer to optimize in stage three over all possible BD partitions. Although this would be feasible, we choose a different approach.

Let us be ambitious and aim for $D = 1$, i.e., for a decoding window of three code words only, an improvement of one code word compared to the best code known (see [19]). Once the present transition in $\mathcal{M}$ is known, we require one code word look-ahead to determine the corresponding transition after state splitting (which is also sufficient since $\mathcal{M}$ is deterministic). The present code word, when not equal to $a = 00000$, determines both the parity of the initial state of the present

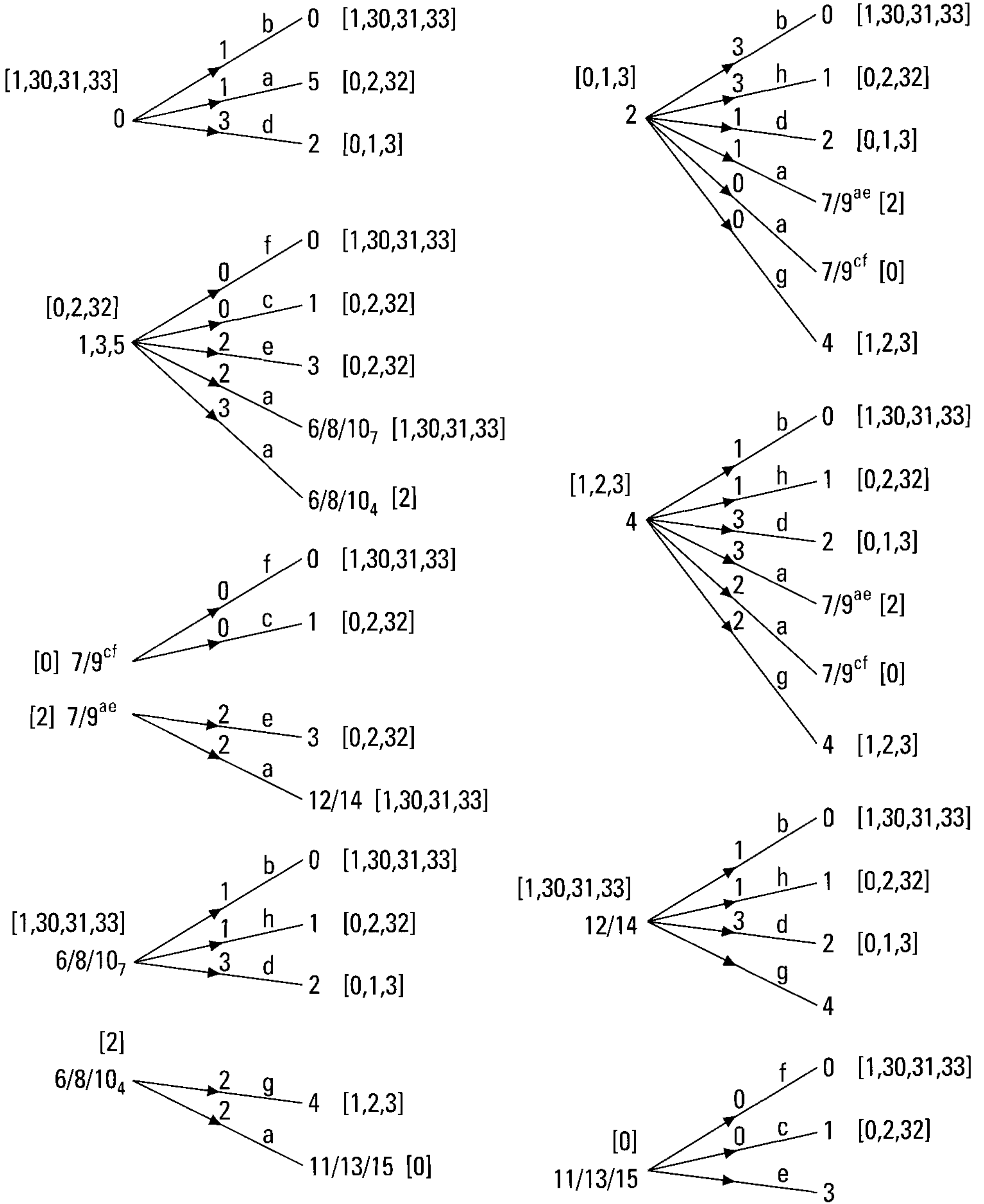

Figure 22: *The resulting code.*

Table 4: *Decoding table.*

word	decoding
$(a, c, e, g, h)a(a, e, g)$	3
$(a, c, e, g, h)a(b, c, d, f, h)$	2
$(b, f)a$	1
$(d)a(a, e)$	1
$(d)a(c, f)$	0
$(\neq d)b, h$	1
$(d)b, h$	3
c	0
$(\neq d)d$	3
$(d)d$	1
e	2
f	0
$(\neq d)g$	2
$(d)g$	0

transition and its terminal state. Finally, we dispose of only one previous symbol to distinguish between the possible initial states of the present transition.

From the above we first conclude that, after state splitting, the labeling of transitions with source symbols has to be independent on how the initial state is entered, except possibly for states 0 and 1, where the labeling may depend on the parity of the previous state. A similar statement then holds for the lists of admissible prefixes associated with the states. Moreover, states $v \geq 5$ cannot be distinguished by looking back one code word only (all of these states are entered by a transition labeled with code word $a = 00000$), hence "consistent labeling" is required for these states.

We can now conclude the following. Without loss of generality we may merge states 11,13, and 15 into one state 11/13/15, and similarly, we may merge states 12 and 14 into one state 12/14, states 7 and 9 into one state 7/9, and finally states 6,8, and 10 into one state 6/8/10. Next, suppose that in states 11/13/15 and 12/14 we choose the BD partitions as indicated in Figure 21. (Other choices will lead to essentially the same code.) Then states 7/9 and 6/8/10 each split in a natural way into a part resembling states 11/13/15 and 12/14, respectively, and one other part (see Figure 22). (Indeed, this state split is indicated by the consistent labeling requirement, that is, due to our aim for $D = 1$.) After this splitting operation, there exists a BD partition for the remaining states, and we can now construct and optimize our code. One of the possibilities is shown in Figure 22.

The freedom left in assigning source labels is used in two ways. In the first

Table 5: *The admissible words of length 3.*

label	word	term. state
a	000	(+3)
b	001	0
c	010	1
d	100	2
e	101	0

Table 6: *The approximate eigenvector.*

v	0	1	2	3	4	5	6
ϕ_v	12	19	18	17	15	11	7

place, it is possible to have consistent labeling for states 1, 3, and 5, so that these states can also be merged. (This will simplify the encoder.) Secondly, we have chosen the indicated labeling of the transitions from states 2 and 4 to have one bit less of error propagation: normally 6 bits, here only 5 bits since the future code word influences only the second bit of the present source symbol. (In fact, this code has "almost" error propagation of 4 bits only.) This code indeed has a decoding window size of three symbols. For convenience, we have presented a decoding table for this code in Table 4. An encoder can be constructed using the methods described in Section 6. It turns out that, after state merging, this encoder has only 22 states, which is also less than the code mentioned in [19].

4.7.2 A new rate-$2/3$ $(1, 6)$-constrained code

We now construct a rate-$2/3$ $(1, 6)$-constrained code with decoding window size of 5 code words or 15 bits, and error propagation of 9 bits only.

Certain highly efficient (d, k)-constrained codes are notoriously difficult to construct. Examples include the rate-$2/5$ $(2, 4)$ code, the rate-$3/5$ $(1, 4)$ code, and the rate-$2/3$ $(1, 6)$ code discussed here. Good codes of these types are easily constructed with our method, though. As an example we consider the $(1, 6)$ code.

The construction starts with the FSTD commonly used to represent this constraint. This FSTD has seven states, numbered $0, 1, \ldots 6$, where the numbering is such that a walk ending in state v generates a word ending in precisely v zeroes. The possible code word labels are the five admissible words of length 3 in Table 5. For ease of description, we separate state 0 into two states 001 and 101 according to how state 0 is entered. To achieve the desired rate, our source alphabet $\boldsymbol{B}$ must have size $N = 4$, so we let $\boldsymbol{B} = \{0, 1, 2, 3\}$.

A local description of the resulting FSTD $\mathcal{M}$ is given in Figure 23. Note

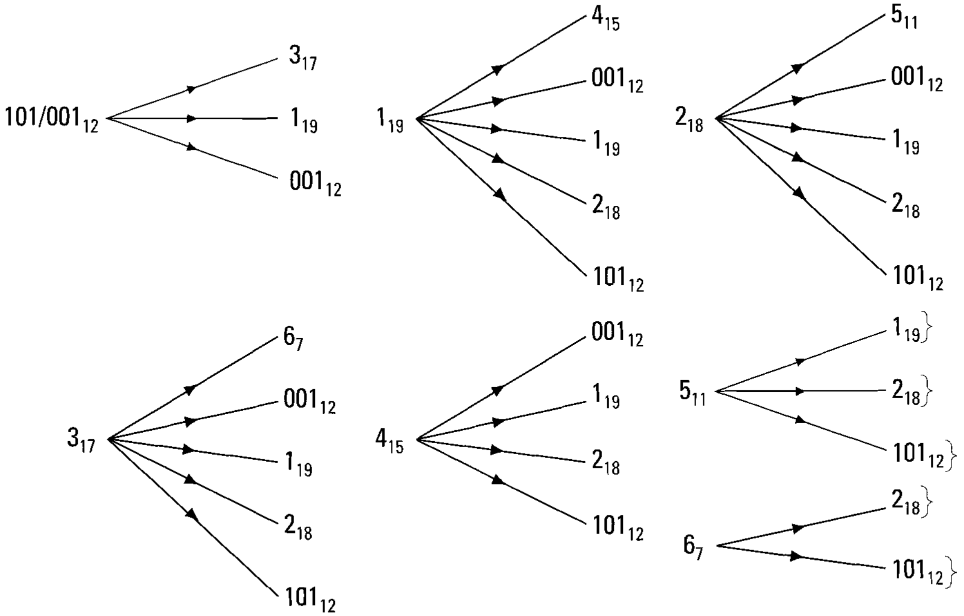

Figure 23: *The local structure of $\mathcal{M}$.*

that the code word label carried by a transition is determined by its end state. The smallest $[\mathcal{M}, 4]$-approximate eigenvector ϕ is listed in Table 6. In Figure 23 these weights are indicated as usual as a subscript to the state number. We choose a look-ahead $M = 2$. As a consequence, the states 1, 2, and 3 (having a weight larger than 16) have to be split into parts of weight at most 16. We choose this value of M since a lot of state splitting is required anyway.

It is not difficult to verify from Table 6 and Figure 23 that in order to have a BD partition in each of the states, at least *two* rounds of state splitting are required. Therefore, we choose $K = 2$. So our decoding window will contain two code words of look-ahead, and since $\mathcal{M}$ is deterministic, this amount of look-ahead is also sufficient to determine the terminal state of a transition after two rounds of state splitting once the initial state in $\mathcal{M}$ is known. For states 5 and 6 we choose the BD partition indicated in Figure 23. Other results (see [13]) now suggest splitting off from both states 1 and 2 a part of weight 16 (or parts with total weight 16). Moreover we have seen in our other examples that it is good strategy to split states in such a way that resulting offsprings can later be merged with other states. (This same heuristic is used in ordinary ACH to minimize the number of encoder states.) Observe that this aim is often automatically achieved when the choice of state splits is guided by a consistent labeling requirement.

The above discussion strongly suggests splitting states 4, 5, and 6 as indicated

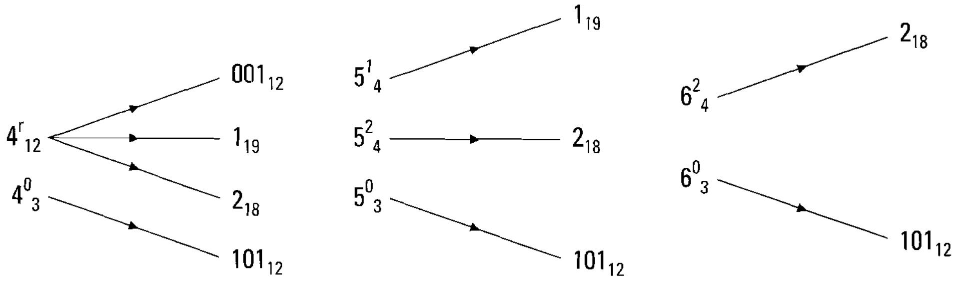

Figure 24: *First round of state-splitting.*

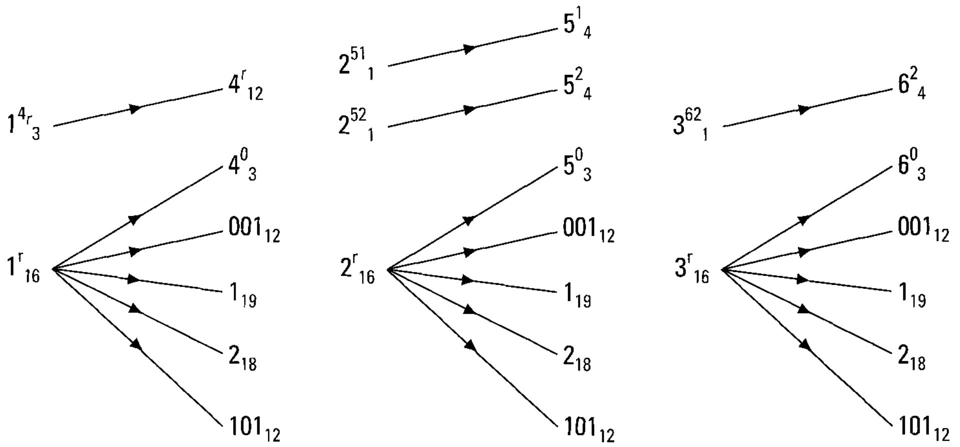

Figure 25: *Second round of state-splitting.*

in Figure 24 and then combining the offsprings of weight 3 from these states with the remaining successor states of states 1, 2, and 3 to form a part of weight 16, which can then be split off from these states in a second round of state splitting. (See Figure 25.) After these state splitting operations, we do indeed have a BD partition in each state, as shown in Figure 26.

The BD partitions shown in Figure 26 are not unique: In the BD partition for state 4^r, we may interchange states 2^{51} and 2^{52}, and for states $1/2/3^r$, we may also interchange states 1^{4r} and $4/5/6^0$. However there is obviously no advantage in not choosing consistent labeling between states 4^r and $1/2/3^r$, so essentially the only choice left is the above choice in state 4^r. It turns out that the best code is indeed obtained by the choice indicated in Figure 26.

Now we have reached stage 3 of our method, where we optimize the code for the smallest decoding window. Although our choice of state splitting operations has been consistent with an aim for $D = 1$, that is, with an aim for a look-back of

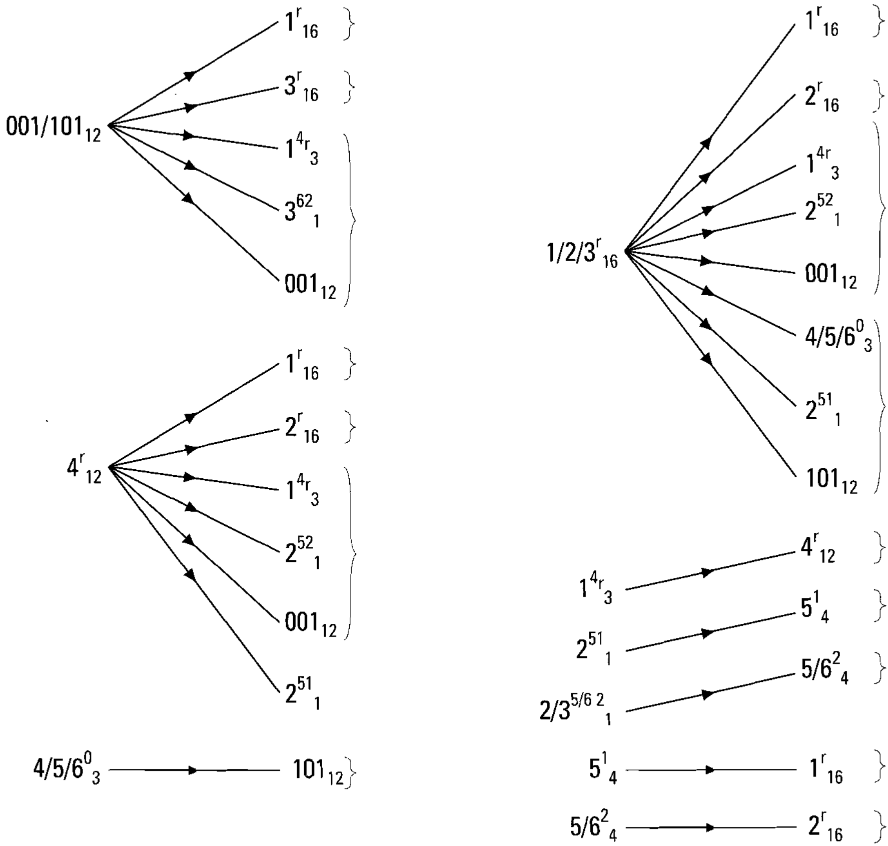

Figure 26: *The resulting FSTD with BD partitions.*

only one code word, at this stage it turns out to be impossible to construct a BD code achieving this aim. (This is easily verified as follows. Assign source symbol variables to transitions, consistent between states originating from states $v \geq 3$. Then states 1^{4r} and 2^{52} necessarily have prefix lists that cannot be combined in the BD partition for state $1/2/3^r$, as would be required.) So we need a decoding window of at least five code words; two code words of look-back, the present code word, and two code words of look-ahead. A BD code with such a decoding window is shown in Figure 27. (The reader will have no difficulty to verify that two previous code words, the present code word, and two future code words are indeed sufficient to determine the actual transition followed in Figure 27.)

Note that the labeling is consistent between states $v \geq 3$ except for the labels of transitions ending in state 2^r, which are either 3 (from states $1/2/3^r$ and 4^r) or 1 (from state $5/6^2$), and for the labels of transitions ending in state 101, which

Figure 27: *The resulting code.*

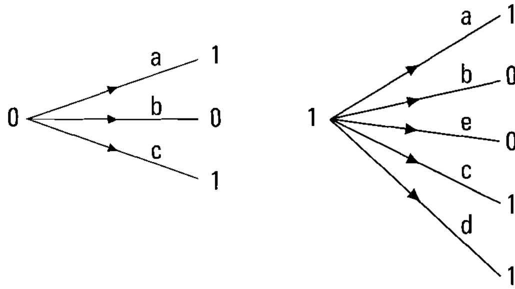

Figure 28: *The local structure of $\mathcal{M}$.*

are either 1 (from state $1/2/3^r$) or 0 (from state $4/5/6^0$. Therefore an error in the first code word in the decoding window can affect at most one bit of the original source symbol. So this code has an error propagation of 9 bits only (instead of the expected 10 bits). Note however that the "span" of an error pattern can be 10 bits, so according to some definitions the error propagation would still be 10 bits.

Finally we note that, using the methods described in Section 6, we can find a finite-state encoder for this code having only 21 states.

4.7.3 A new rate-$2/3$ $(1, 9)$-constrained code

As our final example, we construct a rate-$2/3$ $(1, 9)$-constrained code with decoding window size of 2 code words or 6 bits.

This example—which does not involve state splitting— is included to illustrate the usefulness of the more general local encoder structures in Section 4 as a code-construction tool. Moreover, in this example we initially work without an approximate eigenvector.

A classic method for constructing (d, k) codes is to forbid certain sequences of words in a suitable (d, ∞) code, possibly in combination with certain substitution rules. Our construction of a $(1, 9)$ code is based on this idea; we will first construct a $(1, \infty)$ BD code, and then eliminate all encoding walks that generate the words 000.000.000 or 000.000.001. (This is of course sufficient to obtain the desired k-constraint, as the reader may easily verify.)

Our starting point to construct a rate $2/3$ $(1, \infty)$ code is the FSTD $\mathcal{M}$ with local structure as in Figure 28. The code word labels are the five admissible words of length 3 given earlier in Table 5. Since we desire a rate of $2/3$, we take $N = 4$, and we let $\boldsymbol{B} = \{0, 1, 2, 3\}$.

Our aim is to find a local encoder structure in each state of $\mathcal{M}$, with an additional property. So we will assign prefix lists $\mathcal{W}_v$ and $\mathcal{W}_\alpha$ and labels $\varphi(\alpha)$ to states v and transitions α of $\mathcal{M}$, where each $\mathcal{W}_\alpha$ will be a relabeling of $\mathcal{W}_{\text{end}(\alpha)}$

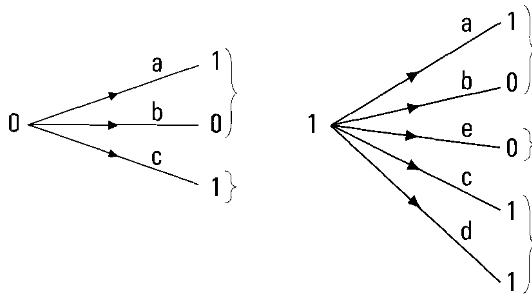

Figure 29: *The partitions.*

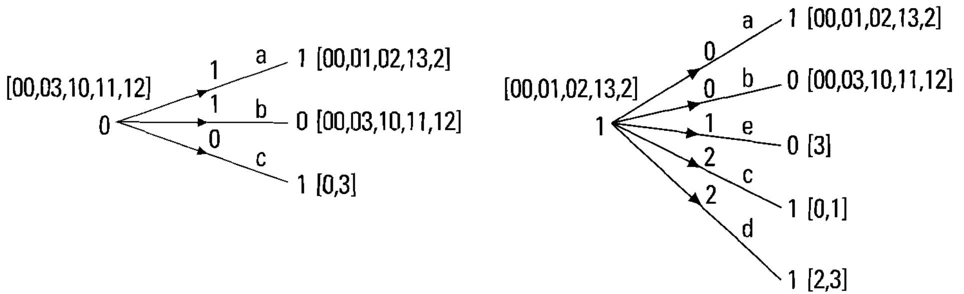

Figure 30: *The local encoder structures.*

(up to depth 1), In such a way that "local encoding" is possible, that is, such that (5) in Section 4 holds. (However, contrary to what we have seen in our examples up to now, we do not require that the prefix lists $\mathcal{W}_v$ have a particular structure.) Note that if we succeed in finding such a local encoder structure, then Theorem 3 guarantees that we can obtain a BD code, with (path) decoding window size of two transitions. The single most important step in such a construction is to find the partitions of outgoing transitions in the two states induced by the labeling, and we will concentrate on that step first. To help us choose these partitions, we note that if two prefix lists $\mathcal{W}_\alpha$ and $\mathcal{W}_\beta$, with $v = \text{beg}(\alpha) = \text{beg}(\beta)$, assigned to transitions α and β that belong to the same part of the partition in state v, have a word in common, then removing this word from one of these prefix lists serves to eliminate certain encoding paths! We want to do so at state 1, if entered by a transition labeled $a = 000$, in order to avoid the generation of the words 000.000 and 000.001. This suggests combining transitions labeled with code words $a = 000$ and $b = 001$ into one part, and if this part is induced by source label x, say, to eliminate the word $xx\cdots$ from the prefix list assigned to (000)1. This observation also suggests choosing a look-ahead of $M = 2$ source symbols. The fact that we combine transitions labeled $a = 000$ and $b = 001$

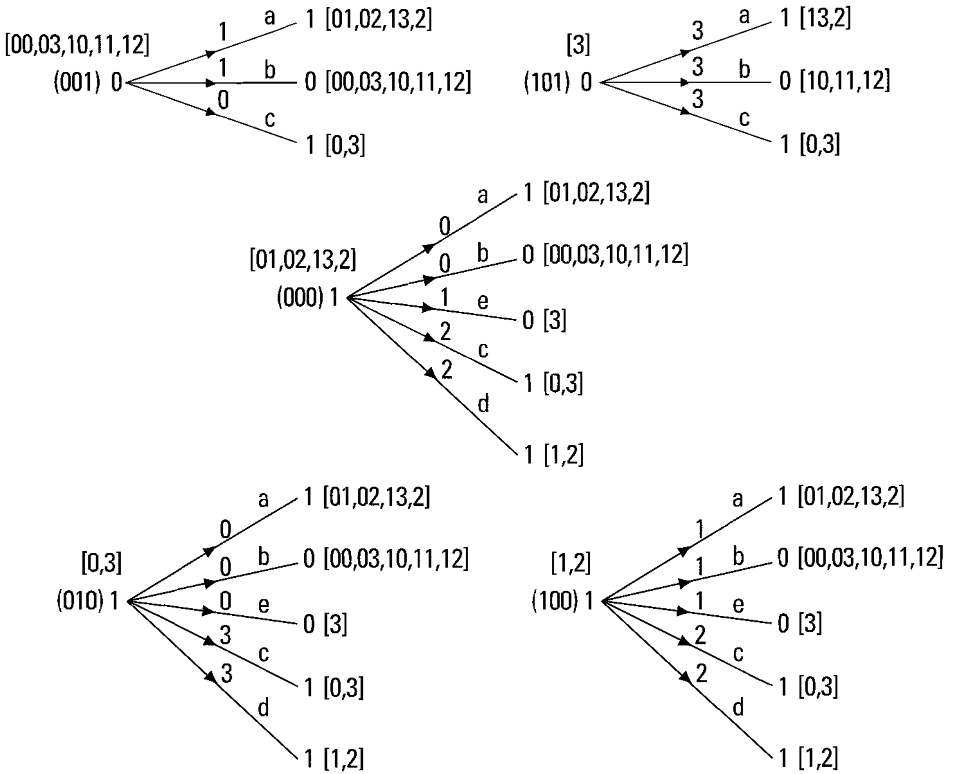

Figure 31: *The resulting code.*

imposes a constraint on the structure of the sets $\mathcal{W}_0$ and $\mathcal{W}_1$, which in turn eliminates certain partitions. In fact, there is only one way to choose the partitions now, which is the one shown in Figure 29. These partitions suggest that we take weights $\phi_0 = 5$ and $\phi_1 = 8$. A local encoder structure for $\mathcal{M}$ with all desired properties is shown in Figure 30.

From these local encoder structures we can construct the BD encoder in a straightforward manner. One of the possibilities is shown in Figure 31. Note that certain words are eliminated from prefix lists at the end of transitions in order to make the encoder deterministic, and of course the word 00 is eliminated from the prefix lists associated with state (000)1 in order to meet our k-constraint. Obviously, our code has a decoding window size of two words: one word look-back to determine the state in Figure 31, and then the present word to determine the actual transition followed from that state. The code also satisfies the $k = 9$ constraint: if the word $a = 000$ is generated, then the encoder is in state (000)1 and thus the next two upcoming source symbols are not both zero, whence at most four

consecutive zero bits are generated next. It turns out that a finite-state encoder for this code constructed as described in Section 6 has only 8 states.

We have some final remarks concerning this example. Of course, this code could also have been constructed in a way similar to that used in other examples, starting from the third power of the FSTD commonly used to represent the $(1, 9)$-constraint. However, the code was actually found in the way described above (with the aim of meeting a k-constraint with smallest possible k), and since we believe the method to be both interesting and potentially useful in similar situations, we decided to present this construction method here. We conjecture that the value $k = 9$ is optimal, in the sense that no rate-$2/3$ $(1, 8)$-constrained code can have a decoding window of only two code words.

4.8 Conclusions

We have presented a new technique to construct sliding-block decodable codes with a small decoding window for constrained systems of finite type. Our methods works very well when the maximum number of transitions from a state in the FSTD-representation of the constraint is small. The power of this method is demonstrated in several examples; these include the construction (by hand!) of a rate-$2/3$ $(d, k) = (1, 6)$ code with a decoding window of only five code words or 15 bits.

The codes obtained by our method can also be constructed by means of the ACH state splitting algorithm, provided that suitable choices are made for the state splitting steps and the assignment of source symbols in the resulting encoder graph. When constructed by state splitting, these codes will exhibit the *reduced-window property*: the *effective* decoding window will be (sometimes much) smaller than predicted from the number of rounds of state splitting used in their construction. Indeed, our method could be incorporated into the ACH algorithm to try and guide the choice of the state splitting operations.

Our approach is very suitable to construct codes by hand, possibly assisted by computer. Further research is needed to determine the best implementation by computer. Other topics for further research include the generalization of this method to the case where the constraints are of almost-finite type.

Acknowledgments

I would like to thank my colleagues J.H. van Lint, Sr. and K.A.S. Immink for their comments on earlier versions of this paper, and L.M.G. Tolhuizen and C.P.M. Baggen for their help while preparing the final version.

4.9 Appendix A: additions to Section 4.4

In this appendix we collect some material connected to Section 4.

4.9.1 Proof of Theorem 3

The proof essentially formalizes the idea of encoder construction as in Figure 6 of Example 3. In the proof, we will express the fact that a set of words is a relabeling of another set of words with the aid of a *relabeling map*. Here, a relabeling map is a map $\tau : \boldsymbol{B}^M \mapsto \boldsymbol{B}^M$ with the property that if, for some number k, two words $x = x_1 \cdots x_M$ and $y = y_1 \cdots y_M$ from U agree in the first k symbols, then their images $\tau(x)$ and $\tau(y)$ also agree in the first k symbols.

It is easily seen that for $U, W \subseteq \boldsymbol{B}^M$, W is a relabeling of U if and only if W is the image of U under some relabeling map. (Indeed, if two words x and y in U are generated by paths in the parsing tree of U that agree in the first k arcs, then after relabeling the parsing tree, the same is true for their images $\tau(x)$ and $\tau(y)$.)

In the proof, we construct for each walk π in $\mathcal{M}$ a label $\varphi(\pi)$ in $\boldsymbol{B}$, a set $\mathcal{W}_\pi$ of words of length M, and a relabeling map τ_π such that the following holds.
i) if two distinct walks π and ω agree in the last s transitions (or if they both end in the same state, if $s = 0$), then the words $\tau_\pi(x)$ and $\tau_\omega(x)$ agree in the last $M - T + s$ symbols (or are equal if $s \geq T$), for all words $x = x_1 \cdots x_M$.
ii) $\mathcal{W}_\pi = \tau_\pi(\mathcal{W}_{\text{end}(\pi)})$,
iii) $\mathcal{W}_\pi \boldsymbol{B} \subseteq \bigcup_{\alpha \in A_{\text{end}(\pi)}} \varphi(\pi\alpha)\mathcal{W}_{\pi\alpha}$,
We construct these objects *inductively*, for walks π of length k, for $k = 2, 3, \ldots$. Remark that i)–ii) above hold for $k = 1$, and iii) for $k = 0$, according to the assumptions in the theorem. (By convention, a state is a walk of length 0.)

In what follows, we will need a few simple properties of relabeling maps or, more briefly, R-maps. If $\tau : \boldsymbol{B}^M \mapsto \boldsymbol{B}^M$ is an R-map, then we denote by $\tilde{\tau}$ the map defined by $\tilde{\tau}(x_0 \cdots x_M) = \tau(x_0 \cdots x_{M-1})x_M$. Also, if $b, c \in \boldsymbol{B}$ are such that τ maps all words beginning with the symbol b into words beginning with symbol c, then we write $\tau^{(b)}$ to denote the map defined by $\tilde{\tau}(bx_1 \cdots x_M) = c\tau^{(b)}(x_1 \cdots x_M)$. The reader will have no difficulty verifying that both $\tilde{\tau}$ and $\tau^{(b)}$ are again R-maps. Also, we observe that the composition $\tau\sigma$ of two R-maps τ and σ is again an R-map.

Now suppose that we have constructed objects $\varphi(\pi)$, $\mathcal{W}_\pi$, and τ_π as above for all walks of length less than k, and suppose furthermore that i)–iii) hold whenever all objects involved in the statement are defined. Now let π be a walk of length $k - 1$. From the defining properties of a local encoder structure, we have

$$\mathcal{W}_{\text{end}(\pi)}\boldsymbol{B} \subseteq \bigcup_{\alpha \in A_{\text{end}(\pi)}} \varphi(\alpha)\mathcal{W}_\alpha,$$

hence by ii),

$$\mathcal{W}_\pi \boldsymbol{B} = \tilde{\tau}_\pi(\mathcal{W}_{\mathrm{end}(\pi)}\boldsymbol{B}) \subseteq \bigcup_{\alpha\in A_{\mathrm{end}(\pi)}} \tilde{\tau}_\pi\Big(\varphi(\alpha)\mathcal{W}_\alpha\Big). \tag{12}$$

For each transition α leaving end(π), let us define the symbol $\varphi(\pi\alpha)$ by

$$\tilde{\tau}_\pi(\varphi(\alpha)x_1\cdots x_M) = \varphi(\pi\alpha)\tau_\pi^{(\varphi(\alpha))}(x_1\cdots x_M), \tag{13}$$

where $\tau_\pi^{(\varphi(\alpha))}$ is the R-map defined from τ_π as explained earlier. Next we note that $\mathcal{W}_\alpha = \tau_\alpha(\mathcal{W}_{\mathrm{end}(\alpha)})$, so if we define the R-map $\tau_{\pi\alpha}$ by $\tau_{\pi\alpha} = \tau_\pi^{(\varphi(\alpha))}\tau_\alpha$, then with the aid of (13), we may express (12) as

$$\mathcal{W}_\pi \boldsymbol{B} \subseteq \bigcup_{\alpha\in A_{\mathrm{end}(\pi)}} \varphi(\pi\alpha)\tau_{\pi\alpha}\Big(\mathcal{W}_{\mathrm{end}(\alpha)}\Big). \tag{14}$$

Hence if we put

$$\mathcal{W}_{\pi\alpha} = \tau_{\pi\alpha}\Big(\mathcal{W}_{\mathrm{end}(\alpha)}\Big), \tag{15}$$

then (14) shows that ii) and iii) also hold for the new objects. Let us now verify that i) also holds. So suppose that the two distinct walks π and ω, each of length less than k, agree in the last s transitions, and let α be a transition such that $\mathrm{end}(\pi) = \mathrm{end}(\omega) = \mathrm{beg}(\alpha)$. Let $x = x_1\cdots x_M$ be a word of length M over $\boldsymbol{B}$. By i), the words $\tau_\pi(x)$ and $\tau_\omega(x)$ agree in the last $M - T + s$ symbols. By definition of $\tau_{\pi\alpha}$, we have that

$$\tilde{\tau}_\pi(\varphi(\alpha)\tau_\alpha(x)) = \varphi(\pi\alpha)\tau_{\pi\alpha}(x), \tag{16}$$

and similarly,

$$\tilde{\tau}_\omega(\varphi(\alpha)\tau_\alpha(x)) = \varphi(\omega\alpha)\tau_{\omega\alpha}(x). \tag{17}$$

From the above expressions and from the definitions of $\tilde{\tau}_\pi$ and $\tilde{\tau}_\omega$, it follows immediately that $\tau_{\pi\alpha}(x)$ and $\tau_{\omega\alpha}(x)$ agree in the last $M - T + s + 1$ symbols, as required.

Now we shall use the objects constructed above to obtain the required BD code. By i), the R-maps τ_π (and hence also the sets of words $\mathcal{W}_\pi$) constructed above depend only on the last T transitions of π. Consequently, if $\pi = \pi_1\cdots\pi_T$ has length T and if $\pi' = \pi_2\cdots\pi_T$, then by iii) we have that

$$\begin{aligned}\mathcal{W}_\pi \boldsymbol{B} &\subseteq \textstyle\bigcup_{\alpha\in A_{\mathrm{end}(\pi)}} \varphi(\pi\alpha)\mathcal{W}_{\pi\alpha}\\ &= \textstyle\bigcup_{\alpha\in A_{\mathrm{end}(\pi)}} \varphi(\pi\alpha)\mathcal{W}_{\pi'\alpha}.\end{aligned}$$

Therefore, the map φ, restricted to walks of length $T+1$ in $\mathcal{M}$ (that is, to transitions in $\mathcal{M}_T$), together with the sets of words $\mathcal{W}_\pi$ for walks π of length T (that is, for states of $\mathcal{M}_T$), constitute an M-symbol look-ahead encoder for $\mathcal{M}_T$, that is, a BD code for $\mathcal{M}$ with path memory T.

4.9.2 Proof of Theorem 4

Let us first observe that since $\mathcal{W}_\alpha$ is a relabeling of $\mathcal{W}_{\mathrm{end}(\alpha)}$, we have

$$|\mathcal{W}_\alpha| \leq |\mathcal{W}_{\mathrm{end}(\alpha)}| = \phi_{\mathrm{end}(\alpha)}. \tag{18}$$

The fact that ϕ is an $[\mathcal{M}, |\boldsymbol{B}|]$-approximate eigenvector can now be shown in the same way as in the proof of Theorem 1. But we can say more: if $\mathcal{W}_v^b$ denotes the collection of all words in $\mathcal{W}_v$ with prefix b, then we have

$$\mathcal{W}_v^b \boldsymbol{B} \subseteq b \bigcup_{\alpha \in A_v^b} \mathcal{W}_{\mathrm{end}(\alpha)}. \tag{19}$$

So by combining (18) and (19), we obtain that

$$|\mathcal{W}_v^b||\boldsymbol{B}| \leq \sum_{\alpha \in A_v^b} \phi_{\mathrm{end}(\alpha)}. \tag{20}$$

Moreover, we obviously have

$$|\mathcal{W}_v^b| \leq |\boldsymbol{B}|^{M-1}, \tag{21}$$

hence we conclude that

$$|\mathcal{W}_v^b| \leq \min\left(\lfloor N^{-1} \sum_{\alpha \in A_v^b} \phi_{\mathrm{end}(\alpha)} \rfloor, N^{M-1} \right). \tag{22}$$

Finally, since

$$\phi_v = |\mathcal{W}_v| = \sum_b |\mathcal{W}_v^b|, \tag{23}$$

the theorem follows.

4.9.3 Some comments on Theorem 4

Although the necessary conditions for BD partitions in Theorem 4 are rather strong, they are not sufficient in general. However in the case where $M = 1$, a given partition is a BD partition *if and only if* these conditions hold, as the reader can easily verify. This observation has a nice consequence. A collection of partitions $A_v = \cup_b A_v^b$ for each state of $\mathcal{M}$ induces in a natural way a similar collection of partitions for the Mth power $\mathcal{M}^M$ of $\mathcal{M}$. Moreover, it is not hard to see that if the original partitions for $\mathcal{M}$ satisfy the conditions in Theorem 4 for some N and for look-ahead M, then the induced partitions for $\mathcal{M}^M$ also satisfy these conditions, but now for alphabet size N^M and look-ahead 1. From these observations we may conclude that if for a given collection of partitions the conditions stated in Theorem 4 hold, then there exists a BD code for the Mth power $\mathcal{M}^M$ of $\mathcal{M}$ with source alphabet $\boldsymbol{B}^M$, that is, a "periodic" BD code, a result first obtained by Franaszek [9].

4.9.4 The complexity problem for BD partitions

Let $\boldsymbol{B} = \{0, \ldots, N-1\}$ be a source alphabet and let M denote the word-length. For each integer n, $0 \leq n \leq N^M$, define $\mathcal{W}(M, n)$ to be the set of the n smallest words of length M over $\boldsymbol{B}$. We investigate the complexity of the *BD problem* $\mathrm{BD}(n, (n_i \mid i \in I), M)$. Here n and n_i, $i \in I$, are integers with $0 \leq n, n_i \leq N^M$, $i \in I$, and the problem is to find a partition of I into parts I_b, $b \in \boldsymbol{B}$, such that

$$\mathcal{W}(M, n)\boldsymbol{B} \subseteq \bigcup_{b \in \boldsymbol{B}} b\Big[\bigcup_{i \in I_b} \mathcal{W}_i\Big] \tag{24}$$

holds for suitable relabelings $\mathcal{W}_i$ of $\mathcal{W}(M, n_i)$ (or to verify that no such partition exists). Note that by taking $n = \phi_v$, $I = A_v$, and $n_\alpha = \phi_{\mathrm{end}(\alpha)}$ for $\alpha \in A_v$ we obtain the problem of finding a BD partition in state v.

Let us say that a subset J of I *covers* a set of words $\mathcal{W} \subseteq \boldsymbol{B}^M$ if $\mathcal{W} \subseteq \cup_{i \in J} \mathcal{W}_i$ holds for suitable relabelings $\mathcal{W}_i$ of $\mathcal{W}(M, n_i)$. Write $n = qN^{M-1} + r$, $0 \leq r < N^{M-1}$. From (24) it is easily seen that we need to find $q + 1$ mutually disjoint subsets J of I, q of which cover $\boldsymbol{B}^M$ and one which covers $\mathcal{W}(M-1, r)\boldsymbol{B}$.

How can we verify whether a given $J \subseteq I$ covers $\mathcal{W}(M-1, r)\boldsymbol{B}$? (To treat both cases simultaneously, we also allow $r = N^{M-1}$ here.) Obviously, we should have

$$r \leq \lfloor N^{-1} \sum_{i \in J} n_i \rfloor. \tag{25}$$

(Cf. Theorem 4.) However, except when $M = 1$ this easily verified necessary condition is not in general sufficient. We will show that, for $M \geq 2$, the verification amounts to solving a BD problem, but now with word-length $M - 1$ instead of M. To see this, write $n_i = q_i N^{M-1} + r_i$, $0 \leq r_i < N^{M-1}$, and note that the set $\mathcal{W}(M, n_i)$ is the union of sets $b\boldsymbol{B}^{M-1}$, $b = 0, \ldots, q_i - 1$, and a further set $q_i \mathcal{W}(M-1, r_i)$. Therefore, after a suitable relabeling the set $\mathcal{W}_i$ can be made to cover any q_i sets of the form $b\boldsymbol{B}^{M-1}$, together with a set of the form $b_i \mathcal{W}_i'$, with $\mathcal{W}_i'$ a relabeling of $\mathcal{W}(M-1, r_i)$. Here b_i can still be chosen arbitrarily in $\boldsymbol{B}$. So together, the sets $\mathcal{W}_i$, $i \in J$, can be made to cover the sets $b\boldsymbol{B}^{M-1}$ for $0 \leq b < \sum_{i \in J} q_i$, and it is easily seen that to verify whether the remaining part of $\mathcal{W}(M-1, r)\boldsymbol{B}$ can also be covered we must solve $\mathrm{BD}(r - N^{M-2} \sum_{i \in J} q_i, (r_i \mid i \in J), M-1)$, a BD problem for word-length $M - 1$. (Note that possibly $r \leq N^{M-2} \sum_{i \in J} q_i$, in which case this last verification can be skipped.)

The above suggest the following recursive algorithm to solve the BD problem.

i) Find the minimal subsets of I that cover $\boldsymbol{B}^M$ or $\mathcal{W}(M-1, r)\boldsymbol{B}$. (In this stage we may need to solve BD problems for smaller word-lengths.)

ii) Find disjoint subsets $J_0, \ldots, J_q$ of I such that J_0 covers $\mathcal{W}(M-1, r)\boldsymbol{B}$ and $J_1, \ldots, J_q$ each cover $\boldsymbol{B}^M$.

Even a partial solution in step ii) of this algorithm can be useful when the BD problem has no solution, cf. Section 5.

Now what does this mean for the complexity problem for BD partitions? Note that the worst-case complexity is (at least) exponential in $d_{\max} = \max_{v \in V} |A_v|$, the maximum number of transitions leaving a state. Moreover, the recursion depth of the algorithm is determined by $M = \max_{v \in V} \lceil \log_N \phi_v \rceil$. The conclusion is that our construction method for BD codes should be applied only when $d_{\max}$ and M are not too big. Practical experience shows that the method can be applied easily for values of $d_{\max}$ at least up to 10 and $M = 2$ or 3. (Cf. Example 2 in Section 8.)

4.10 Appendix B: connection with the ACH algorithm

Here we will establish a connection between the ACH state splitting algorithm and our method, and thereby explain some of the reasons why our method is so effective. We begin with a review of the ACH algorithm and its properties, as far as relevant here. For proofs of our claims and for further details we refer to [2] or the excellent overview [19]. After this introduction to ACH, we will show that the BD method and the ACH algorithm are related by a time shift over M symbols, where M is the number of rounds of state splitting (see also [10]). Finally, we show that our method is related to ACH by a time shift over K symbols, for some K between 0 and M. Along the way we obtain a lower bound on M in terms of the smallest possible maximal component of an approximate eigenvector, which may be of independent interest.

The ACH algorithm is a method to transform a given FSTD into an encoder graph. It is based on a certain type of transformation referred to as a *round of state splitting*. Such a transformation produces from a given FSTD $\mathcal{M}$ a new FSTD $\mathcal{M}'$ in the following way. For each state v of $\mathcal{M}$, let the set A_v of transitions leaving v be partitioned into sets A_v^i, $i = 1, \ldots, n_v$. We construct the FSTD $\mathcal{M}'$ as follows. Each state v of $\mathcal{M}$ produces states v^i, $i = 1, \ldots, n_v$, in $\mathcal{M}'$, and each transition α from state v to state w contained in A_v^i produces transitions

$$\alpha^j : v^i \mapsto w^j,$$

$j = 1, \ldots, n_w$, in $\mathcal{M}'$, all of which inherit their label from α. This construction makes precise the intuitive idea that each state v can be split into new states v^i according to different futures (represented by their possible successors). We refer to states v^i and transitions α^j as to *children* of state v and transition α in $\mathcal{M}'$, and

to v and α as their *parents* in $\mathcal{M}$, respectively. This notion of children and parents extends to walks in $\mathcal{M}$ and $\mathcal{M}'$ in the obvious way.

The single most important property of this transformation is that all children $\alpha'\beta'$ in $\mathcal{M}'$ of a given walk $\alpha\beta$ in $\mathcal{M}$ agree in transition α'. (Intuitively, we think of this property as a transition in $\mathcal{M}$ with *given future* determining a *unique* transition in $\mathcal{M}'$.) Consequently, the parent/child relation in fact provides a one-to-one correspondence between bi-infinite walks in $\mathcal{M}$ and $\mathcal{M}'$ and, in particular, $\mathcal{M}$ and $\mathcal{M}'$ generate the same constrained sequences. In the terminology of Section 2, the labeling map L' that assigns to each transition of $\mathcal{M}'$ its parent in $\mathcal{M}$ (the *ancestor map*) is block-decodable, with window (0, 1).

We say that an FSTD $\mathcal{M}^*$ is produced from the FSTD $\mathcal{M}$ by *M rounds of state splitting* if $\mathcal{M}^*$ is the result of a sequence of transformations

$$\mathcal{M} = \mathcal{M}^{(0)} \mapsto \mathcal{M}^{(1)} \cdots \mapsto \mathcal{M}^{(M)} = \mathcal{M}^*,$$

where each intermediate transformation consists of a round of state splitting. We may extend the parent/child relation by transitivity to states and transitions in $\mathcal{M}$ and $\mathcal{M}^*$. An easy induction argument then suffices to provide proof of the following claims. Each state (transition) in $\mathcal{M}^*$ has a unique parent state (transition) in $\mathcal{M}$. Moreover, the parent/child relation provides a one-to-one correspondence between bi-infinite walks in $\mathcal{M}$ and $\mathcal{M}^*$ and, in particular, $\mathcal{M}$ and $\mathcal{M}^*$ generate the same constrained sequences. Indeed, the ancestor map L^* is block-decodable, with window $(0, M)$: all children $\alpha_0^\star \dots \alpha_M^\star$ in $\mathcal{M}^*$ of a given walk $\alpha_0 \dots \alpha_M$ in $\mathcal{M}$ agree in transition $\alpha_0^\star$. We denote this transition by $\Phi(\alpha_0 \dots \alpha_M)$ and refer to it as the transition in $\mathcal{M}^*$ determined by the transition α_0 in $\mathcal{M}$ with *future* $\alpha_1 \dots \alpha_M$.

The ACH state splitting algorithm in fact transforms a *pair* $(\mathcal{M}, \phi)$, where ϕ is an $[\mathcal{M}, N]$-approximate eigenvector, and does so by what is called *ae-consistent* rounds of state splitting. (Here N is a positive integer representing the desired code rate.) We say that a round of state splitting that transforms $\mathcal{M}$ into $\mathcal{M}'$ as explained earlier is *ae-consistent* (with respect to the approximate eigenvector ϕ) if, for each state v of $\mathcal{M}$, the weight ϕ_v associated with v can be distributed over its children v^i in $\mathcal{M}'$, in such a way that the resulting assignments of weights to the states of $\mathcal{M}'$ now constitutes an $[\mathcal{M}', N]$-approximate eigenvector. Moreover, to exclude various degenerate cases we require that *at least one* state v' of $\mathcal{M}'$ has been assigned a non-zero weight smaller than the weight of its parent state in $\mathcal{M}$. As a consequence of this last requirement there will be more non-zero weights in $\mathcal{M}'$ than in $\mathcal{M}$. It is not hard to see that a collection of partitions A_v^i provides a ae-consistent round of state splitting precisely when

$$\phi_v \leq \sum_i \lfloor N^{-1} \sum_{\alpha \in A_v^i} \phi_{\mathrm{end}(\alpha)} \rfloor .$$

(Compare this requirement with the necessary condition for the existence of a local encoder structure in Theorem 4.)

The main result in [2] applies to an irreducible FSTD $\mathcal{M}$ and states that if the non-zero weights in ϕ are not constant, then some ae-consistent split with respect to ϕ can always be found. Consequently, after a finite number of transformations, the pair $\mathcal{M}, \phi$ gets transformed into a pair $\mathcal{M}^*, \phi^\star$ for which $\phi^\star$ is an $[\mathcal{M}^*, N]$-approximate eigenvector with constant non-zero weights. Eliminating zero-weight states from $\mathcal{M}^*$ if necessary, we may assume that $\phi^\star$ is constant, and then the definition of an approximate eigenvector reveals that, necessarily, at least N transitions leave each state in $\mathcal{M}^*$. Consequently, by eliminating (if necessary) some states and transitions from $\mathcal{M}^*$, we may obtain an FSTD $\mathcal{N}$ that is an *encoder graph* for $\mathcal{M}$ (that is, precisely N transitions leave each state of $\mathcal{N}$). In what follows, we assume that the transitions of $\mathcal{N}$ are labeled with source symbols from a source alphabet $\boldsymbol{B}$ of size N, in such a way that for each state of $\mathcal{N}$ the N transitions leaving this state all have distinct labels.

Let us now consider the decoder for the code produced by the encoder $\mathcal{N}$. The crucial observation is that the code has a *path-decoding window* of size $M+1$. Indeed, let the *decoding function* φ be defined as follows. For a walk $\alpha_0 \cdots \alpha_M$ in $\mathcal{M}$, let $\varphi(\alpha_0 \cdots \alpha_M)$ denote the source label of the (unique) transition $\alpha^\star = \Phi(\alpha_0 \ldots \alpha_M)$ in $\mathcal{M}^*$ determined by transition α_0 with future $\alpha_1 \cdots \alpha_M$, provided that $\alpha^\star$ is also present in $\mathcal{N}$ (ϕ is left undefined otherwise). We will suppose that the labeling of $\mathcal{M}$ is of *finite type* (see Section 2), with memory m and anticipation a, say. Consequently, from an encoded sequence $f = \{f_n\}$ we may recover each transition in the walk $\boldsymbol{\alpha} = \{\alpha_n\}$ in $\mathcal{M}$ generating this sequence by observing $m+1+a$ consecutive symbols from f. Finally, we can recover the original source sequence used to encode $f = \{f_n\}$ from $\boldsymbol{\alpha} = \{\alpha_n\}$ through the decoding function φ, which essentially "reads off" source labels of successive transitions $\alpha_n^\star = \Phi(\alpha_n \ldots \alpha_{n+M})$ contained in the walk in $\mathcal{N}$ that generated $f = \{f_n\}$.

We now investigate the connection between the ACH algorithm and our method. We will first use the encoder graph $\mathcal{N}$ to construct a collection of state trees in $\mathcal{M}$, and then we will apply Theorem 6 to construct a BD code equivalent to the ACH code up to a time shift over M symbols. For a state v^* in $\mathcal{N}$ with parent state v in $\mathcal{M}$ let Σ_{v^*} be the collection of parents in $\mathcal{M}$ of walks of length M in $\mathcal{N}$ with initial state v^*. Observe that each Σ_{v^*} is non-empty, all walks in Σ_{v^*} have initial state v, and two sets Σ_{v^*} and Σ_{w^*} are disjoint for distinct states v^* and w^* in $\mathcal{N}$. (Indeed, if $\alpha_0 \cdots \alpha_M$ is a walk in $\mathcal{M}$ with $\mathrm{beg}(\alpha_1) = v$, then all children of $\alpha_1 \cdots \alpha_M$ start from the terminal state v^* of $\Phi(\alpha_0 \ldots \alpha_M)$ in $\mathcal{M}^*$, and this state v^* necessarily is a child of v.) Next, we write

$$\Sigma_{v^*} \xrightarrow{b/\alpha} \Sigma_{w^*} \tag{26}$$

if there is a transition α^* from v^* to w^* in $\mathcal{N}$ with parent α and source label b in $\boldsymbol{B}$.

Let the graph $\mathcal{G}$ have as states the sets Σ_{v^*} and transitions as in (26). For each walk $\boldsymbol{\alpha}$ of length M in $\mathcal{M}$, let $\mathcal{W}_{\boldsymbol{\alpha}}$ consist of all words $b_0 \cdots b_{M-1}$ that are a source labeling of some walk $\boldsymbol{\alpha}^\star$ in $\mathcal{N}$ with parent $\boldsymbol{\alpha}$.

Theorem 7: The triple $[\mathcal{M}_M, \varphi, \mathcal{W}]$, where $\mathcal{W}$ is as defined above and where φ is the decoder function, constitutes an M-symbol look-ahead BD code with path memory (at most) M. This code is equivalent to the original ACH code by a time shift over M symbols.

Proof: Since $\mathcal{N}$ is an encoder graph, each state Σ_{v^*} in $\mathcal{G}$ has N successors. Moreover, it is fairly obvious that $\mathcal{G}$ and $\mathcal{N}$ generate the same code for $\mathcal{M}$. Now the theorem is a direct consequence of Theorem 6. (It is easily verified that φ and $\mathcal{W}$ as defined here coincide with those used in Theorem 6.) □

The shifting mechanism referred to in the above theorem functions in the following way. If $\varphi(\alpha_0 \ldots \alpha_M) = b$, then in the ACH code the label b is assigned to transition α_0 with future $\alpha_1 \ldots \alpha_M$, while in the BD code the label b is assigned to transition α_M with history $\alpha_0 \ldots \alpha_{M-1}$.

Corollary 8: [10] A code constructed by the ACH algorithm (an ACH code) by M rounds of state splitting is equal to an M-symbol look-ahead BD code, up to a shift over M symbols.

Conversely, each BD code can be produced by the ACH algorithm, with an appropriate choice of approximate eigenvector and state splitting steps, and for an appropriate labeling with source symbols of the resulting encoder graph. Moreover, given a local encoder structure with look-ahead M, an ordinary finite-state encoder can be constructed directly by only M rounds of state-splitting.

Detailed proof of these statements, involving induction on the amount of look-ahead M in combination with a relabeling technique, is given in Appendix C. For completeness' sake, we sketch a proof of these statements here, using the notation from Theorem 4. The idea is to split each state v into states v^b, for $b \in \boldsymbol{B}$, according to the partitions A_v^b. (Theorem 4 shows that this splitting is ae-consistent with respect to ϕ.) Assign to state v^b the set of words $w_2 \cdots w_M$ for which $bw_2 \cdots w_M \in \mathcal{W}_v$, and define the list of words associated with transitions in a similar way. Also, a transition ending in state w^c is assigned label c. In this way, we obtain a local encoder structure with look-ahead $M - 1$, by one round of state splitting. Continue this procedure until the look-ahead equals zero.

From the above correspondence between ACH codes and BD codes we may draw an interesting conclusion. Let θ_v denote the maximum value of $|\mathcal{W}_{\boldsymbol{\alpha}}|$ over all walks $\boldsymbol{\alpha}$ of length M ending in v. It is an easy exercise to show from Theorem 1

that θ is an $[\mathcal{M}, N]$-approximate eigenvector. Since obviously all components of θ have a value of at most N^M, we obtain the following result. (This result also follows from Theorem 5 in [18].)

Theorem 9: If the ACH algorithm produces an encoder graph for $\mathcal{M}$ by M rounds of state splitting, then $N^M \geq L$, where L denotes the smallest integer for which we can find an $[\mathcal{M}, N]$-approximate eigenvector with all its components of size less than or equal to L.

We now come to the main observation of this section. It turns out that, frequently, the splitting steps in the ACH algorithm and the labeling with source symbols of the resulting encoder graph $\mathcal{N}$ can be chosen in such a way that $\mathcal{N}$ has what we will call the *reduced window property*: There is a number $T < M$ such that the function $\varphi(\alpha_0 \cdots \alpha_M)$ depends only on $\alpha_{M-T} \cdots \alpha_M$, or in other words, only the $T + 1$ rightmost transitions in the (path-) decoding window are required for decoding. Consequently, the corresponding BD code has in fact path-memory T instead of M!

The reason *why* this can happen so frequently is more difficult to explain. In fact, elsewhere we will show that, given an FSTD $\mathcal{M}$ of finite type, the ACH algorithm is capable (with a suitable choice of approximate eigenvector) to construct, up to equivalence by a time shift, *all* sliding-block decodable codes that possess a finite-state encoder with finite encoding delay and a finite-type code word labeling. (The same result has recently be obtained in quite a different way by Ashley and Marcus [4].) If the required shift referred to above is larger than the "look-back part" of the decoding window of the given code, then the corresponding ACH code will have the reduced window property.

Although there may exist a sequence of splitting steps such that the resulting encoder graph $\mathcal{N}$, when suitably labeled with source symbols, has the reduced window property, it is not at all evident how to recognize suitable splitting steps so as to work towards this property. (Some information concerning this problem can be found in [13].) At first it would seem that code construction through the BD method only replaces this problem by another one of comparable difficulty. Fortunately, we may (at least partly) overcome these problems by mixing the BD method and the state splitting approach. Indeed, let us consider what happens if we stop state splitting after K rounds, for some number K between 0 and T. Let $\mathcal{M}'$ denote the resulting FSTD after these K rounds. (Observe that now a transition in $\mathcal{M}$ as well as K future transitions are needed to determine the corresponding transition in $\mathcal{M}'$.)

We can then construct (essentially) the same code as in Theorem 7, but now as an $(M - K)$-symbol look-ahead BD code *for* $\mathcal{M}'$. The decoding function φ'

for $\mathcal{M}_{M-K}$ is now defined as follows. If $\alpha'_0 \cdots \alpha'_{M-K}$ is a walk in $\mathcal{M}'$, then

$$\varphi'(\alpha'_0 \cdots \alpha'_{M-K}) = b,$$

where b is the source label of the transition $\alpha^\star_0$ in $\mathcal{N}$ determined by this walk (and undefined otherwise). Consequently, whenever $\alpha_0 \cdots \alpha_M$ is a walk in $\mathcal{M}$ for which $\alpha'_0 \cdots \alpha'_{M-K}$ is a child of $\alpha_0 \cdots \alpha_{M-K}$ in $\mathcal{M}'$, we have

$$\varphi(\alpha_0 \cdots \alpha_M) = \varphi'(\alpha'_0 \cdots \alpha'_{M-K}).$$

Using this relation between φ and φ', it is easily shown that $\varphi'(\alpha'_0 \cdots \alpha'_{M-K})$ depends in fact only on the last $T-K+1$ transitions $\alpha'_{M-T} \cdots \alpha'_{M-K}$. (Indeed, suppose that the two walks $\alpha'_0 \cdots \alpha'_{M-K}$ and $\beta'_0 \cdots \beta'_{M-K}$ in $\mathcal{M}'$ agree in the last $T-K+1$ transitions. Now if $\alpha'_{M_K+1} \cdots \alpha'_M$ is a walk that extends both these walks, then the parent walks $\alpha_0 \cdots \alpha_M$ and $\beta_0 \cdots \beta_M$ of these extensions agree in the last $T+1$ transitions, whence

$$\varphi'(\alpha'_0 \cdots \alpha'_{M-K}) = \varphi(\alpha_0 \cdots \alpha_M) = \varphi(\beta_0 \cdots \beta_M) = \varphi'(\beta'_0 \cdots \beta'_{M-K}),$$

which proves our claim.)

So we may conclude that the original ACH code is identical (up to a time shift over $M-K$ symbols) to a BD code for $\mathcal{M}'$ with look-ahead $M-K$ and path-memory $T-K$. In particular, for $K=T$ we obtain a BD code with path-memory 0, that is, an $(M-T)$-symbol look-ahead assignment for $\mathcal{M}'$. If we have some luck, then the structure of the look-ahead assignments $\mathcal{W}_{v'}$ associated with the states v' of $\mathcal{M}'$ is such that the methods in Section 4 apply. (Otherwise more splitting steps would be required.) If we have still more luck, then even fewer than T rounds of state splitting will do; this occurs if all sets $\mathcal{W}_{\alpha'_{M-T+1} \cdots \alpha'_{M-K}}$ that correspond to walks $\alpha'_{M-T+1} \cdots \alpha'_{M-K}$ ending in the same state have the *same structure*. (See Section 4.) Our examples provide ample evidence that this approach is indeed a profitable one, requiring often only a few rounds of state splitting to obtain good codes.

4.11 Appendix C: BD codes from ACH

Here, we begin by showing that each M-symbol look-ahead encoder or local encoder structure $[\mathcal{M}, \varphi, \mathcal{W}]$ can in fact be obtained from the pair $(\mathcal{M}, \phi)$ by ae-consistent state-splitting, where $\phi_v = |\mathcal{W}_v|$, for all states v. (Later we discuss the case of general BD codes.) We first do this for a look-ahead encoder, then we indicate how to adapt the proof in the case of a local encoder structure.

In both cases, we transform the pair $(\mathcal{M}, \varphi)$ into a pair $(\mathcal{M}', \varphi')$ by ae-consistent state-splitting and the encoder $[\mathcal{M}, \varphi, \mathcal{W}]$ into an encoder $[\mathcal{M}', \varphi', \mathcal{W}']$ for a code, equivalent to the original code up to a shift, in the following way. To obtain $\mathcal{M}'$, we partition each set A_v of transitions leaving state v into sets A_v^b, for $b \in \boldsymbol{B}$, where A_v^b consists of all transitions α leaving v for which the source label $\varphi(\alpha) = b$. So each state v splits into states v^b, for all $b \in \boldsymbol{B}$, and each transition α, from v to w, say, produces transitions α^c, for all $c \in \boldsymbol{B}$, where α^c is a transition from v^b to w^c, with $b = \varphi(\alpha)$, which inherits its label $L'(\alpha^c) = L(\alpha)$ from α.

Also, in both cases we define the sets $\mathcal{W}'_{v^b}$ as the collection of all words $b_2 \cdots b_M$ for which the word $bb_2 \cdots b_M$ is contained in $\mathcal{W}_v$, and we let $\phi'_{v^b} = |\mathcal{W}'_{v^b}|$. It is not difficult to check that ϕ' is an approximate eigenvector for $\mathcal{M}'$. (Alternatively, we can invoke Theorem 4.)

In the case of a look-ahead encoder, we define $\varphi'(\alpha^c) = c$. We now claim that $[\mathcal{M}', \varphi', \mathcal{W}']$ constitutes an $(M-1)$-symbol look-ahead encoder. Indeed, this follows immediately from the definition of a look-ahead encoder, together with the observation that $\mathcal{W}_v = \cup_{b \in \boldsymbol{B}} \mathcal{W}'_{v^b}$ holds for each state v. Moreover, it is an easy matter to check that encodings produced by this encoder are the same as those produced by the original encoder, up to a shift by one symbol.

In the case of a local encoder structure, we have to adapt the definition of φ', and to define the sets $\mathcal{W}'_{\alpha'}$, for transitions α' in $\mathcal{M}'$. We proceed as follows. Recall that we require the existence of a relabeling map (or R-map) $\tau_\alpha : \mathcal{W}_{\mathrm{end}(\alpha)} \mapsto \mathcal{W}_\alpha$ for each transition α of $\mathcal{M}$ (see Appendix A, proof of Theorem 3). By the definition of an R-map, all words in $\mathcal{W}_{\mathrm{end}(\alpha)}$ that begin with a fixed source symbol $c \in \boldsymbol{B}$ are mapped to words in $\mathcal{W}_\alpha$ that all begin with the same source symbol c', say. We slightly abuse notation and denote this source symbol c' by $\tau_\alpha(c)$.

Now let $\mathcal{W}'_{\alpha^c}$ denote the collection of all words $c_2 \cdots c_M$ for which the word $\tau_\alpha(c)c_2 \cdots c_M$ is the image under τ_α of a word in $\mathcal{W}_{\mathrm{end}(\alpha)}$ that begins with the symbol c. Moreover, we let $\tau'_{\alpha^c} : \mathcal{W}'_{\mathrm{end}(\alpha)^c} \mapsto \mathcal{W}'_{\alpha^c}$ denote the map on $\mathcal{W}'_{\mathrm{end}(\alpha)^c}$ "induced" by τ_α. It is easily checked that the maps τ'_{α^c} are again R-maps.

Finally, we define the source symbol labeling φ' of transitions in $\mathcal{M}'$ by letting $\varphi'(\alpha^c) = \tau_\alpha(c)$. Now, using the definition of a local look-ahead encoder and the additional observation that $\mathcal{W}_\alpha = \cup_{c \in \boldsymbol{B}} \mathcal{W}'_{\alpha^c}$ holds for each α, it is a simple matter to check that $[\mathcal{M}', \varphi', \mathcal{W}']$ constitutes an $(M-1)$-symbol look-ahead local encoder structure. As before, there occurs a shift by one symbol.

The proof of our claims now follows by induction on M. Indeed, the above procedure is repeated until the amount of look-ahead M is reduced to zero, in which case the resulting look-ahead encoder or local encoder structure in fact represents an encoder for the original code, up to a shift by M symbols.

Now, we consider the case of an M-symbol look-ahead BD code $[\mathcal{M}_T, \varphi, \mathcal{W}]$

based on the Tth order history graph $\mathcal{M}_T$ of $\mathcal{M}$. As shown above, an encoder for an equivalent code up to a shift over M symbols can be obtained from $(\mathcal{M}_T, \phi)$, where $\phi_{\alpha_0\cdots\alpha_{T-1}} = |\mathcal{W}_{\alpha_0\cdots\alpha_{T-1}}|$ for all paths $\alpha_0\cdots\alpha_{T-1}$ in $\mathcal{M}$, by M rounds of ae-consistent state-splitting. Therefore, in order to show that an equivalent encoder can be obtained from $\mathcal{M}$ by ae-consistent state-splitting, it is sufficient to show that the pair $(\mathcal{M}_T, \phi)$ can be obtained through ae-consistent state-splitting from a pair $(\mathcal{M}, \phi^{(0)})$, for some approximate eigenvector $\phi^{(0)}$ for $\mathcal{M}$. This can always be achieved as follows. First, we consider the pair $(\mathcal{M}^{[T]}, \phi)$, where $\mathcal{M}^{[T]}$ denotes the Tth order edge graph of $\mathcal{M}$, which is the same graph as $\mathcal{M}_T$, except that a "transition" $\alpha_0\cdots\alpha_T$ now carries label $L(\alpha_0)$ instead of label $L(\alpha_T$. (A "state" $\alpha_0\cdots\alpha_{T-1}$ is now best considered as the state $v = \text{beg}(\alpha_0)$ with "future" $\alpha_0\cdots\alpha_{T-1}$.) Note that $\mathcal{M}_T$ and $\mathcal{M}^{[T]}$ generate the same sequences, but shifted over T symbolds. It is now fairly evident that the pair $(\mathcal{M}^{[T]}, \phi)$ can be obtained from T rounds of ae-consistent state-splitting from the pair $(\mathcal{M}, \phi^{(0)})$. The intermediate pairs $(\mathcal{M}^{[k]}, \phi^{(k)})$ occuring in the process are composed from the kth order edge graph $\mathcal{M}^{[k]}$ and an approximate eigenvector $\phi^{(k)}$ for this edge graph. Here, $\phi^{(k)}_{\alpha_0\cdots\alpha_{k-1}} = \sum^* \phi_{\text{end}(\pi)}$, where the summation ranges over all paths $\pi = \alpha_0\cdots\alpha_{k-1}\pi_k\cdots\pi_{T-1}$ starting in state v.

We note that under certain conditions ae-consistent state-splitting starting with $(\mathcal{M}, \phi)$ is also possible. This occurs for example when for each path $\alpha_0\cdots\alpha_{T-1}$, the size of the set of words $\mathcal{W}_{\alpha_0\cdots\alpha_{T-1}}$ only depends on the terminal state of the path. In that case, the ae-consistent state-splitting process to obtain an encoder for $[\mathcal{M}_T, \varphi, \mathcal{W}]$ from the pair $(\mathcal{M}_T, \phi)$ can be made to correspond to a similar process starting with $(\mathcal{M}, \phi)$. We leave further details concerning this special case to the reader.

If we combine the above results, we indeed find an ae-consistent state-splitting procedure to obtain an encoder for our BD code $[\mathcal{M}_T, \varphi, \mathcal{W}]$.

References

[1] R.L. Adler, R.K. Brayton, M. Hassner, and B.P. Kitchens, "A rate-2/3 (1, 6) RLL code", *IBM Techn. Discl. Bul.*, vol. 27, no. 8, Jan. 1985.

[2] R.L. Adler, D. Coppersmith, and M. Hassner, "Algorithms for sliding block codes. An application of symbolic dynamics to information theory", *IEEE Trans. Inform. Theory*, vol. IT-29, no. 1, pp. 5–22, Jan. 1983.

[3] R.L. Adler, J. Friedman, B. Kitchens, and B.H. Marcus, "State splitting for variable-length graphs", *IEEE Trans. Inform. Theory*, vol. IT-32, no. 1, pp. 108–113, Jan. 1986.

[4] J. Ashley and B.H. Marcus, "Canonical encoders for sliding block decoders", *Siam J. Disc. Math.*, vol. 8, no. 4, pp. 555–605, Nov. 1995.

[5] M. Boyle, B. Kitchens, and B.H. Marcus, "A note on minimal covers for sofic systems", *Proc. AMS*, vol. 95, no. 3, pp. 403–411, Nov. 1985.

[6] P.A. Franaszek, "On future-dependent block coding for input-restricted channels", *IBM J. Res. Develop.*, vol. 23, pp. 75–81, Jan. 1979.

[7] P.A. Franaszek, "Synchronous bounded delay coding for input restricted channels", *IBM J. Res. Develop.*, vol. 24, pp. 43–48, Jan. 1980.

[8] P.A. Franaszek, "A general method for channel coding", *IBM J. Res. Develop.*, vol. 24, pp. 638–641, Sept. 1980.

[9] P.A. Franaszek, "Construction of bounded delay codes for discrete noiseless channels", *IBM J. Res. Develop.*, vol. 26, pp. 506–514, July 1982.

[10] P.A. Franaszek, "Coding for constrained channels: A comparison of two approaches", *IBM J. Res. Develop.*, vol. 33, pp. 602–608, Nov. 1989.

[11] C.D. Heegard, B.H Marcus, and P.H. Siegel, "Variable-length state splitting with applications to average runlength constrained (ARC) codes", *IEEE Trans. Inform. Theory*, vol. IT-37, no. 3, pp. 759–777, May 1991.

[12] H.D.L. Hollmann, "A block-decodable $(d,k) = (1,8)$ run-length-limited rate 8/12 code", *IEEE Trans. Inform. Theory*, vol. 40, no. 4, pp. 1292–1296, July 1994.

[13] H.D.L. Hollmann, "Bounded-delay encodable codes for constrained systems from state combination and state splitting", *Benelux Symp. Inform. Theory*, Veldhoven, May 1993.

[14] K.A. Schouhamer Immink, "Block-decodable run-length-limited codes via look-ahead technique", *Philips J. Res.*, vol. 46, no. 6, pp. 293–310, 1992.

[15] K.A. Schouhamer Immink, *Coding techniques for digital recorders*, Englewood Cliffs, NY: Prentice-Hall, 1991.

[16] R. Karabed and B.H. Marcus, "Sliding-block coding for input-restricted channels", *IEEE Trans. Inform. Theory*, vol. IT-34, no. 1, pp. 2–26, Jan. 1988.

[17] A. Lempel and M. Cohn, "Lookahead coding for input-restricted channels", *IEEE Trans. Inform. Theory*, vol. IT-28, pp. 933–937, Nov. 1982.

[18] B.H. Marcus and R. Roth, "Bounds on the number of states in encoder graphs for input-constrained channels", *IEEE Trans. Inform. Theory*, vol. IT-37, no. 3, pp. 742–758, May 1991.

[19] B.H Marcus, P.H. Siegel, and J.K. Wolf, "Finite-state modulation codes for data storage", *IEEE J. Sel. Areas Commun.*, vol. 10, no. 1, pp. 5–37, Jan. 1992.

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

[21] P.H. Siegel, "Recording codes for digital magnetic storage", *IEEE Trans. Magnet.*, vol. MAG-21, no. 5, pp. 1344–1349, Sep. 1985.

Chapter 5

Bounded delay encodable, block-decodable codes for constrained systems

Abstract— **We introduce and investigate the class of bounded delay encodable block-decodable (BDB) codes. Several characterizations for this class of codes are given, and some construction methods, especially for one-symbol look-ahead BDB codes, are described. In another direction, we use our results to show the existence of a decision procedure for some basic coding problems.**

Index Terms — **RLL code, BDB code, block-decodable code, principal state-set, look-ahead coding .**

5.1 Introduction

Digital data transmission or storage systems commonly apply *modulation codes* to translate or *encode* sequences of source data into sequences that obey certain constraints. Typical "binary" examples include the family of (d, k)-constraints, where the number of zeroes between consecutive ones must be at least d and at most k [18], [14], and the related family of (d, k) RLL constraints, where each run of zeroes or ones must have length at least $d + 1$ and at most $k + 1$ [12]. Both families of constraints have important applications in storage systems such as magnetic and optical disks and tapes.

Commonly, to achieve the desired encoding the source data is grouped into words of length m and a *code* is used to translate these words into a sequence of

© 1996, IEEE. Reprinted, with permission, from IEEE Transactions on Information Theory vol. 42, no. 6, November 1996.

words of length n (called *codewords*) that obeys the specified constraints. The quantity R= m/n is called the *rate* of the code.

Many code construction methods actually deliver *sliding block (decodable)* codes, where after encoding, a given block can be retrieved or *decoded* by examining only its corresponding codeword, the last a codewords preceding it, and the next m codewords succeeding it, for some fixed integers a and m. The collection of these codewords is called the *decoding window* for the given block, and the quantity $a + 1 + m$ is called the *window size* of the decoder.

An important subclass of sliding-block codes for practical applications are the *block-decodable* codes. These codes have window size one, consequently codewords can be decoded without the knowledge of preceding or succeeding codewords. Block-decodable codes can be advantageously used in combination with error correcting codes having a fixed block-size n such as Reed-Solomon codes over the finite field GF(2^n), since error propagation is limited to one codeword. Several construction methods for such codes have been proposed, but unfortunately, efficient (high-rate) codes obtained by these methods require in general a fairly large block-length.

Recent work by Immink [12] (see also [11] and Hollmann [10] has changed this picture. In these papers, two new construction methods for block-decodable codes, namely state-combination [12] and the principal state-sets method [10], have been introduced, and several new "record" block-decodable codes (for example, a rate $m/n = 8/16$ (2, 8) RLL code and many others in [12], and a rate $m/n = 8/12$ (1, 8) RLL code in [10]) have been obtained.

All block-decodable codes referred to above belong to the class of *Bounded delay (encodable), Block-decodable codes* or *BDB codes* ([10], see also [4]). Here we investigate these BDB codes in detail. After Section 5.2, where we establish our notation, we show in Section 5.3 that all BDB codes for constraints of finite type can be obtained in a well-defined way from the minimal deterministic presentation (the Shannon cover) of the constraint. These results are used later in Section 5.7 to obtain decision procedures for some basic coding problems. Some proofs concerning this material, and a detailed discussion concerning the universality of the BD method and the ACH algorithm, can be found in Appendix A and B.

Then, in Sections 5.4 and 5.5, we offer a description of M-symbol look-ahead BDB codes in terms of principal state-sets (Section 5.4) for $M \leq 1$ and state-trees (Section 5.5) for general M. This method, which generalizes the principal states method of Franaszek, offers a practical construction method for one-symbol look-ahead BDB codes, see e.g. [10].

A generalization of the state combination method as introduced in [12] (see also [11]) is investigated in Section 5.6. We discuss our results in Section 5.8.

5.2 Preliminaries

We consider the design of a (block-decodable) code for a constrained system $\mathcal{L}$, with a given *source alphabet* $\boldsymbol{B}$, at a given rate $R = p/q$, where $p = \log_2 N$ and $N = |\boldsymbol{B}|$ denotes the size of $\boldsymbol{B}$. (In most practical applications, $N = 2^p$ and each *source symbol* from $\boldsymbol{B}$ consists of p user bits.) Typically, $\mathcal{L}$ is presented by a *finite-state transition-diagram* or FSTD. An FSTD, which is essentially a labeled digraph, can be formally described as a four-tuple $\mathcal{M} = (V, A, L, \boldsymbol{F})$, consisting of a (finite) collection V of vertices or *states*, a (finite) collection A of arcs or *transitions*, and a map $L : A \mapsto \boldsymbol{F}$, the *labeling map*, that assigns to each transition a symbol from the *labeling alphabet* $\boldsymbol{F}$. Each transition α of $\mathcal{M}$ has a unique *initial state* $\text{beg}(\alpha)$ and a unique *terminal state* $\text{end}(\alpha)$, and is referred to as a *transition from* $\text{beg}(\alpha)$ *to* $\text{end}(\alpha)$. We denote by A_v the collection of transitions leaving state v (the outgoing transitions in v), that is, all transitions α with $\text{beg}(\alpha) = v$.

The constrained system $\mathcal{L}$ is *generated* (or *presented*) by $\mathcal{M}$ if $\mathcal{L}$ consists of all words $L(\omega) = L(\omega_1) \cdots L(\omega_n)$ obtained from sequences $\omega = \omega_1 \cdots \omega_n$ of transitions in $\mathcal{M}$ for which $\text{end}(\omega_i) = \text{beg}(\omega_{i+1})$, $i = 1, \ldots, n-1$. Such a sequence ω is called a *walk* in $\mathcal{M}$, of *length n*, and $L(\omega)$ is called the sequence *generated* by this walk.

The said coding problem requires that we translate or *encode* arbitrary sequences of source symbols into sequences of q-bit *codewords* that are contained in $\mathcal{L}$, that is, generated by some walk in $\mathcal{M}$. Usually the code word size q is a multiple of the size q_1 of the labeling symbols in $\boldsymbol{F}$, say $q = nq_1$. (Typically, $\boldsymbol{F} = \{0, 1\}$ and $q_1 = 1$.) Therefore it is desirable to have a presentation for $\mathcal{L}$ where the labeling symbols are actually codewords. Such a presentation can be obtained from the *nth order power graph* $\mathcal{M}^n$ of $\mathcal{M}$. This FSTD has the same states as $\mathcal{M}$, but the transitions are now walks of length n in $\mathcal{M}$; a walk $\omega = \omega_1 \cdots \omega_n$ is considered as a transition in $\mathcal{M}^n$ from state $\text{beg}(\omega) = \text{beg}(\omega_1)$ to state $\text{end}(\omega) = \text{end}(\omega_n)$, and is labeled with the codeword $L(\omega)$ generated by ω. So by replacing $\mathcal{M}$ by an appropriate power graph, we may assume that the labels of $\mathcal{M}$ are actually q-bit codewords, and now the encoding problem is to transform an arbitrary source sequence $\{b_n\}$ into a sequence $\{f_n\}$ of codewords or *blocks* that can be generated by some walk in $\mathcal{M}$.

We will be interested in encoders that employ a finite, *bounded*, amount of look-ahead to transform source sequences directly into *walks* in $\mathcal{M}$. (The actual encoding of the source sequence is then obtained by reading off the codeword labels from the corresponding walk.) Look-ahead techniques are investigated for example in [5], [16], [12], and are further developed in [9]. The following discussion of look-ahead encoders closely follows [9], where further details and proofs

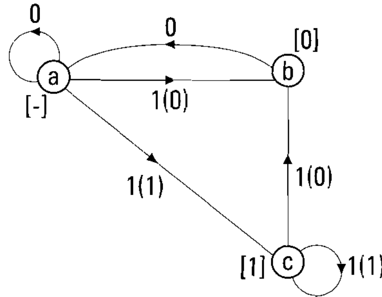

Figure 1: *A look-ahead encoder.*

can be found.

An *M-symbol look-ahead encoder for $\mathcal{M}$* involves a map $\varphi : A \mapsto \boldsymbol{B}$ that assigns to each transition α of $\mathcal{M}$ a source symbol $\varphi(\alpha)$ from $\boldsymbol{B}$. The task of the encoder is to translate each sequence $b_1, b_2, \cdots$ of source symbols into a walk $\alpha_1, \alpha_2, \ldots$ in $\mathcal{M}$ such that $\varphi(\alpha_n) = b_n$ holds for all n.

In the simplest case, $M = 0$ (no look-ahead is required) and the pair $[\mathcal{M}, \varphi]$ constitutes an *encoder graph*, that is, for each state v and each source symbol b, there is a transition α leaving v with source label $\varphi(\alpha) = b$. This transition can then be used to encode b, by the codeword $L(\alpha)$, if the encoder is presently in state v.

Observe that an encoder graph based on $\mathcal{M}$ exists if and only if at least N transitions leave each state of $\mathcal{M}$. Even if this is not the case, encoding may still be possible by employing look-ahead to avoid entering a state from which the upcoming source symbol(s) cannot be encoded. The encoder is then allowed to base its choice of the next transition to be included in the encoding path on the present source symbol along with the next M upcoming source symbols, for some fixed number M termed the *look-ahead* of the encoder. Since a state is entered with knowledge of the next M upcoming source symbols only (one symbol being encoded by the entering transition), encoding from this state must be possible for *all* source sequences starting with these M symbols. Consequently, with each state v of the encoder there will be associated a collection of words $\mathcal{W}_v$, called the *list of admissible prefixes* (each member being a source word of length at most M) specifying the prefixes of sequences of source symbols that can be encoded from this state. Consider the following simple example, taken from [9].

Example 1: A typical look-ahead encoder, with source alphabet $\{0, 1\}$, is presented in Figure 1. Note that we have not indicated the codeword labels of the transitions. Throughout this paper, we will adopt the convention that unlabeled transitions are assumed to be labeled with their terminal state. (The reason for

this seemingly strange convention is that often the FSTD $\mathcal{M}$ representing the constraint will be in Moore-machine form, with as states the codewords, and with the transitions from a state indicating which codewords are possible successors to the codeword associated with this state.) So here, we have $\boldsymbol{F} = \{a, b, c\}$.

The encoder has three states, marked a, b, and c. In state a, no requirements on upcoming source symbols are imposed (indicated in Figure 1 by the prefix list $\mathcal{W}_a = [-]$ containing the *empty* word only.) State b may be entered only with an upcoming source symbol equal to 0 (indicated by the prefix list $\mathcal{W}_b = [0]$), and similarly, state c has prefix list $\mathcal{W}_c = [1]$. Each transition in Figure 1 carries a source symbol, sometimes followed by a parenthesized symbol indicating a constraint on the upcoming source symbol. For example, to encode source symbol 1 from state a, the encoder looks ahead at the upcoming source symbol, and chooses the transition from a to b, labeled 1(0), if this symbol equals 0, or the transition from a to c, labeled 1(1), otherwise. (A similar notation is used e.g. in [16].) It is easily verified that Figure 1 indeed presents an encoder; encoding of a source sequence from an encoder state is possible if the sequence has a prefix contained in the list of admissible prefixes associated with that state. This encoder never employs more than *one* symbol look-ahead, and is therefore called a *one-symbol look-ahead encoder*.

Finally, we observe that the code is *block-decodable*: the codeword a always encodes source symbol 0, and both codewords b and c always encode source symbol 1. □

Which conditions must be satisfied by the lists of admissible prefixes $\mathcal{W}_v$ in order that M-symbol look-ahead encoding is indeed possible? To answer that question, we first observe that we may assume without loss of generality that each list $\mathcal{W}_v$ consists entirely of words of length M over $\boldsymbol{B}$. (Indeed, replacing $\mathcal{W}_v$ by the set of words of length M with a prefix in $\mathcal{W}_v$ does not affect whether or not a given source sequence has a prefix in $\mathcal{W}_v$.) Then, the required condition states that for each state v of $\mathcal{M}$, for each word $b_1 \cdots b_M \in \mathcal{W}_v$, and for each source symbol $b = b_{M+1}$ from $\boldsymbol{B}$, there must be a transition α leaving state v such that $\varphi(\alpha) = b_1$ (that is, α encodes b_1), and $b_2 \cdots b_{M+1} \in \mathcal{W}_{\mathrm{end}(\alpha)}$. To understand the second condition, note that since the encoder bases its choice of transition α on $b_1, \cdots, b_M, b_{M+1}$ only, all source sequences with prefix $b_2 \cdots b_{M+1}$ must be encodable from state end(α).

Conversely, if the lists $\mathcal{W}_v$ satisfy these conditions and if at least one of the lists is non-empty, then M-symbol look-ahead encoding is indeed possible, as the reader can easily verify. (Observe that if v is a state for which $\mathcal{W}_v$ is non-empty, then each source sequence, when prefixed with a fixed word in $\mathcal{W}_v$, can be encoded starting from this state v.)

It will be convenient to introduce the following notation. We denote by $\mathcal{W}_v\boldsymbol{B}$ the set of all words $b_1\cdots b_{M+1}$ with $b_1\cdots b_M \in \mathcal{W}_v$ and $b_{M+1} \in \boldsymbol{B}$. If $\varphi(\alpha)$ denotes the source label of transition α, then we write $\varphi(\alpha)\mathcal{W}_{\mathrm{end}(\alpha)}$ to denote the collection of all words $\varphi(\alpha)b_2\cdots b_{M+1}$ with $b_2\cdots b_{M+1} \in \mathcal{W}_{\mathrm{end}(\alpha)}$. The collection of all subsets of a set X will be denoted by $\mathcal{P}(X)$. We can now state the following formal definition of a look-ahead encoder (see [9]).

Definition 1: An *M-symbol look-ahead encoder* with *source alphabet* $\boldsymbol{B}$ for a FSTD $\mathcal{M}$ consists of a triple $[\mathcal{M}, \varphi, \mathcal{W}]$, where $\varphi : A \mapsto \boldsymbol{B}$ denotes a labeling of the transitions with symbols from $\boldsymbol{B}$, and $\mathcal{W} : V \mapsto \mathcal{P}(\boldsymbol{B}^M)$ is a map which assigns to each state v of $\mathcal{M}$ a collection $\mathcal{W}_v$ of words of length M over $\boldsymbol{B}$, *list of admissible prefixes at* v, such that
(a) at least one $\mathcal{W}_v$ is non-empty, and
(b) for all states v, we have

$$\mathcal{W}_v\boldsymbol{B} \subseteq \bigcup_{\alpha\in A_v} \varphi(\alpha)\mathcal{W}_{\mathrm{end}(\alpha)}, \tag{1}$$

where A_v denotes the set of transitions leaving v.
We will refer to such a map $\mathcal{W}$ as an M-symbol *look-ahead assignment* for $\mathcal{M}$ (with respect to φ, or with respect to the partition of the labeling alphabet of $\mathcal{M}$ induced by φ).

In practical applications it is desirable that the decoding decision on a codeword depends on the *local context* of that codeword only in order to limit error propagation. (See e.g. [18].) Codes with that property are termed *sliding-block decodable*. We point out that codes that possess a look-ahead encoder $[\mathcal{M}, \varphi, \mathcal{W}]$ as in Definition 1 are certainly sliding-block decodable if (the labeling map L of) $\mathcal{M}$ is *of finite type*, that is, if there are numbers m and a, $m, a \geq 0$, such that walks $\omega_{-m}\cdots\omega_0\cdots\omega_a$ that generate a given word $f_{-m}\cdots f_0\cdots f_a$ all agree in transition ω_0. The numbers m and a are termed the *memory* and *anticipation* of L and $\mathcal{M}$, respectively, We can then reconstruct the walk $\omega_{-m}\cdots\omega_0\cdots\omega_{n+a}$ used to generate a given sequence of codewords $f_{-m}\cdots f_0\cdots f_{n+a}$ in $\mathcal{L}$, except for the first m and last a transitions. Indeed, each ω_k, $0 \leq k \leq n$, is determined uniquely by $f_{k-m}\cdots f_k\cdots f_{k+a}$, the *decoding window* for the codeword f_k. As a result, f_k is decoded as b_k, where $b_k = \varphi(\omega_k)$ is the source symbol associated with transition ω_k. The number $m+1+a$ is termed the *decoding-window size* of the code. It is well-known that a constrained system $\mathcal{L}$ is of finite type (that is, $\mathcal{L}$ can be generated by an FSTD of finite type) precisely when $\mathcal{L}$ can be specified in terms of a finite list of *forbidden blocks* [1].

We say that a code is *block-decodable* if the code is sliding-block decodable with a decoding window of size one codeword, that is, if the decoding decision

on a codeword is independent of preceding and succeeding codewords. Observe that the set of codewords $\boldsymbol{F}$ of a block-decodable code can be partitioned into *codeword classes* $\boldsymbol{F}_b$, $b \in \boldsymbol{B}$, such that a codeword $f \in \boldsymbol{F}$ is decoded as source symbol b precisely when $f \in \boldsymbol{F}_b$.

Many code construction methods employ an *approximate eigenvector*. (See e.g. [3], [16], [1],[9].) Here, a collection of non-negative integer *weights* ϕ_v, not all zero, associated with the states v of $\mathcal{M}$ constitute an $[\mathcal{M}, N]$*-approximate eigenvector* if, for each state v, the sum of the weights of terminal states of the outgoing transitions is not less than N times the weight of state v, that is, if

$$N\phi_v \leq \sum_{\alpha \in A_v} \phi_{\mathrm{end}(\alpha)} \tag{2}$$

holds for each state v. Usually, we think of $\phi = (\phi_v)_{v \in V}$ as a *vector*, and the above condition is stated as

$$N\phi \leq D_{\mathcal{M}}\phi, \tag{3}$$

where the inequality is to be interpreted component-wise. Here $D_{\mathcal{M}}$ denotes the $|V| \times |V|$ *adjacency matrix* of $\mathcal{M}$; $D_{\mathcal{M}}(v, w)$ equals the number of transitions from state v to state w. We quote the following theorem from [9].

Theorem 1: Let $[\mathcal{M}, \varphi, \mathcal{W}]$ be an M-symbol look-ahead encoder with source alphabet $\boldsymbol{B}$. Then the weights ϕ_v defined by $\phi_v = |\mathcal{W}_v|$, the size of the list of admissible prefixes associated with state v, satisfy $0 \leq \phi_v \leq |\boldsymbol{B}|^M$ and constitute an $[\mathcal{M}, |\boldsymbol{B}|]$-approximate eigenvector.

5.3 BDB codes

Informally, a *bounded-delay encodable, block-decodable code* or *BDB code* with source alphabet $\boldsymbol{B}$ for a constrained system $\mathcal{L}$ is a collection of disjoint codeword classes $\boldsymbol{F}_b$, $b \in \boldsymbol{B}$, with the property that each source sequence $b_1 b_2 \cdots$ can be encoded into a constrained sequence $f_1 f_2 \cdots$ in $\mathcal{L}$ with $f_i \in \boldsymbol{F}_{b_i}$ by a finite-state machine, with a *bounded* encoding delay, or, equivalently, using a bounded amount of look-ahead.

In a sense, the use of the word "code" is somewhat misleading, since it suggest that a BDB code is characterized by its code sequences, while in reality it is characterized by its *decoder map* (and the fact that bounded-delay encoding is possible). This point is further discussed in the appendix.

Evidently, an M-symbol look-ahead encoder for an FSTD $\mathcal{M}$ with $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}$ which is block-decodable in fact provides a BDB code for $\mathcal{L}$. We will refer to such an encoder as a *BDB encoder for* $\mathcal{M}$. In the appendix, we present a formal definition of BDB codes, together with a proof of the following theorem.

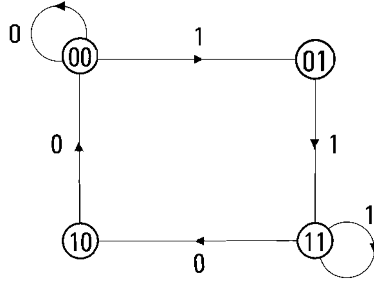

Figure 2: *A FSTD for* $(1, \infty)$ *RLL sequences.*

Theorem 2: Each BDB code for a constrained system $\mathcal{L}$ can be encoded by a BDB encoder for some FSTD $\mathcal{M}$ with $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}$.

We illustrate the above by a few examples.

Example 2: (See [12].) Let $\mathcal{L}$ denote the collection of $(1, \infty)$ RLL sequences, that is, binary sequences not containing the words 101 and 010. ("Runs have length at least two".) $\mathcal{L}$ can be represented by the FSTD $\mathcal{G}$ in Figure 2. We wish to find a rate $m/n = \log_2(6)/4$, one-symbol look-ahead BDB code for $\mathcal{L}$. So let $\boldsymbol{B} = \{1, \ldots, 6\}$, and let $\mathcal{M} = \mathcal{G}^4$ be the fourth power of $\mathcal{G}$. Recall that the transitions of $\mathcal{M}$ are labeled with symbols from the set $\boldsymbol{F}$ consisting of the words of length four in $\mathcal{L}$. We define codeword classes $\boldsymbol{F}_b$, $b \in \boldsymbol{B}$, as follows.

$$
\begin{aligned}
\boldsymbol{F}_1 &= \{0000\}, & \boldsymbol{F}_4 &= \{1111\}, \\
\boldsymbol{F}_2 &= \{0011\}, & \boldsymbol{F}_5 &= \{0111, 1001, 1110\}, \\
\boldsymbol{F}_3 &= \{1100\}, & \boldsymbol{F}_6 &= \{1000, 0110, 0001\}.
\end{aligned}
$$

We claim that these constitute a BDB code. Indeed, the reader may verify that the map $\mathcal{W} : \{00, 01, 10, 11\} \mapsto \mathcal{P}(\boldsymbol{B})$ defined by

$$\mathcal{W}_{00} = \mathcal{W}_{11} = \boldsymbol{B}, \qquad \mathcal{W}_{01} = \{3, 4, 6\}, \qquad \mathcal{W}_{10} = \{1, 2, 5\},$$

is a (one-symbol) look-ahead assignment for $\mathcal{M}$ (with respect to the $\boldsymbol{F}_i$.) □

Example 3: Consider the FSTD $\mathcal{M}$ in Figure 3. According to the convention introduced in Example 1, the label ling alphabet $\boldsymbol{F}$ of this FSTD is $\boldsymbol{F} = \{a, b, c, d\}$. Let $\boldsymbol{F}_0 = \{a\}$, $\boldsymbol{F}_1 = \{b, c, d\}$, and put $\boldsymbol{B} = \{0, 1\}$. The reader can easily verify that the sets

$$\mathcal{W}_a = \boldsymbol{B}^2, \qquad \mathcal{W}_b = \{00, 01\}, \qquad \mathcal{W}_c = \{10\}, \qquad \mathcal{W}_d = \{11\},$$

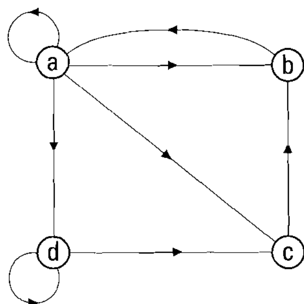

Figure 3: *The FSTD $\mathcal{M}$ for Example 3.*

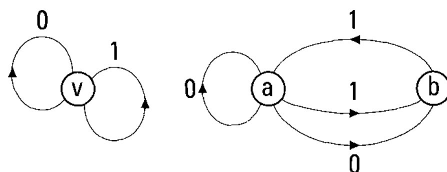

Figure 4: *The FSTD's $\mathcal{G}$ (left) and $\mathcal{M}$ (right) for Example 4.*

constitute a two-symbol look-ahead assignment for $\mathcal{M}$, hence the $\boldsymbol{F}_i$ constitute a two-symbol look-ahead BDB code. □

According to Theorem 2, we might as well have defined a BDB code for a constrained system $\mathcal{L}$ as the collection of codeword classes of a block-decodable look-ahead encoder for some FSTD $\mathcal{M}$ that generates a constrained system included in $\mathcal{L}$. At first, it would seem that such a definition is highly dependent on the particular FSTD $\mathcal{M}$. Fortunately, this is not the case if the constrained system is of finite type. Indeed, we have the following theorem.

Theorem 3: Let $\mathcal{L}$ be a constrained system generated by a FSTD $\mathcal{M}$ of finite type. Then, each M-symbol look-ahead BDB code for $\mathcal{L}$ is the collection of codeword classes of an $(M + a)$-symbol block-decodable look-ahead encoder for $\mathcal{M}$, where a denotes the anticipation of $\mathcal{M}$.

The proof of this result can also be found in Appendix A. As the next example shows, this result need not hold if the FSTD is not of finite type. (A more subtle example that illustrates another limitation of this result is quoted in Appendix B.)

Example 4: Consider the FSTD's $\mathcal{G}$ and $\mathcal{M}$ in Figure 4. Both these FSTD's have labeling alphabet $\boldsymbol{F} = \{0, 1\}$, and generate the constrained system $\mathcal{L}$ consisting of unconstrained binary sequences. Let $\boldsymbol{B} = \{0, 1\}$, and let $\boldsymbol{F}_0 = \{0\}$ and $\boldsymbol{F}_1 = \{1\}$. Obviously, this partition constitute a (trivial) BDB code for $\mathcal{L}$; if we let $\mathcal{W}_v = \Lambda$, where Λ denotes the *empty word* over $\boldsymbol{B}$, then $\mathcal{W}$ defines a

0-symbol look-ahead assignment for $\mathcal{G}$. However, the reader may verify that there does *not* exist a look-ahead assignment for the FSTD $\mathcal{M}$ with respect to this partition of $\boldsymbol{F}$. Indeed, an even run of ones followed by a zero can only be encoded by starting with a transition from state a, while an odd run of ones followed by a zero can only be encoded by starting with a transition from state b. □

It is well-known (see e. g. [1]) that $\mathcal{L}$ is of finite type precisely then when its *(right) Shannon cover* (the "smallest" deterministic FSTD that generates $\mathcal{L}$) has finite memory. (Since it is deterministic, it has anticipation $a = 0$.) Also, constrained systems of finite type are precisely those constrained systems $\mathcal{L}$ that can be characterized by a finite list of *forbidden words* that are not allowed to occur as sub-words in words of $\mathcal{L}$. The above theorem then has the following consequence.

Theorem 4: Each M-symbol look-ahead BDB code for a constrained system of finite type is the collection of codeword classes of a block-decodable M-symbol look-ahead encoder for its (unique) right Shannon cover.

The above results have some interesting theoretical consequences. Let us agree to call a code construction method *universal* for a class of constrained systems and for codes of a certain type for such systems if each such code for each such system can in principle be obtained by use of the method. The *code system* of a code is the collection of all possible encoded sequences produced by the encoder for the code. It is now possible to show the following.

Theorem 5: Both the ACH state-splitting method and the BD-method in [9] are universal for sliding-block decodable codes of constrained systems of finite type that have a finite-state encoder with a finite-type code word labeling. Moreover, both methods can construct any given sliding block-decoder of codes for constraints of finite type. Here, the constructed encoder may have a code system different from the code system of the given code in the case that this code system is not of finite type.

Since both the BD method and the ACH algorithm transform graphs of finite type into graphs that are again of finite type, the above theorem can be interpreted as stating that both methods are capable to construct everything that is not a-priory impossible.

In what follows we give a sketch of the proof of this claim. (Details concerning the more subtle points of this theorem are given in Appendix B.) The key observation is that a sliding-block decodable code with window size T for a constrained system $\mathcal{L}$ generated by a FSTD $\mathcal{M}$ is equivalent to a block-decodable code for the Tth higher block system $\mathcal{L}_T$ generated by the Tth order edge-graph $\mathcal{M}^{[T]}$ of $\mathcal{M}$(see e.g. [1]). Now the claim for the BD method follows from the

above results together with the observations in [9, Section 2]. The corresponding claim for the state-splitting method follows from the equivalence of the two methods as shown in [9, appendices] (see also Chapter 4, Appendix C). Most of the above claims for the state-splitting method was also proved in [2] by other means.

5.4 Principal state-sets for one-symbol look-ahead BDB codes

In this section and the next one, we shall describe our first construction method for BDB codes. The method is universal, in the sense that *each* BDB code can thus be constructed. This section covers the case of one-symbol look-ahead BDB codes, and serves as an introduction to the more general case.

Let $\mathcal{M} = (V, A, L, \boldsymbol{F})$ be a FSTD, generating the constrained system $\mathcal{L}$, say. Let $\{\boldsymbol{F}_b\}_{b \in \boldsymbol{B}}$ be a partition of the labeling alphabet $\boldsymbol{F}$ of $\mathcal{M}$. Suppose that somebody claims that the $\boldsymbol{F}_b$, $b \in \boldsymbol{B}$, constitute the codeword classes of a (one-symbol look-ahead, say) BDB code for $\mathcal{L}$. How could we verify this? Of course, one method would be to find lists of admissible prefixes for all states of $\mathcal{M}$, but this may be difficult. We will now describe an alternative method.

Let $\mathcal{E}$ denote a collection of non-empty subsets of the state-set V of $\mathcal{M}$. We will commonly refer to members of $\mathcal{E}$ as *principal state-sets*. As will become clear later on, this notion generalizes the concept of *principal states* as introduced by Franaszek [3]. A subset $C \subseteq \boldsymbol{F}$ will be called a *codeword class with respect to* $\mathcal{E}$ if for each principal state-set $E \in \mathcal{E}$ we can find a state e in E such that the collection of C-*successors* of e in $\mathcal{M}$ contains a principal state-set E' in $\mathcal{E}$. Here, a state v of $\mathcal{M}$ is a C-*successor* of state e if there is a transition in $\mathcal{M}$ from e to v with label contained in C. Now we have the following result.

Theorem 6: Let $\mathcal{M} = (V, A, L, \boldsymbol{F})$ be a FSTD that generates a constrained system $\mathcal{L}$. Let $C_1, \ldots, C_N$ be mutually disjoint subsets of $\boldsymbol{F}$, and suppose that each C_i is a codeword class with respect to some given collection $\mathcal{E}$ of non-empty subsets of the states of $\mathcal{M}$. Then the C_i are the codeword classes of a one-symbol look-ahead BDB code for $\mathcal{L}$ with source alphabet $\boldsymbol{B} = \{1, \ldots, N\}$.

Proof: Think of a (principal state-set) E from $\mathcal{E}$ as representing states end(α) of a collection of transitions α from some (earlier determined) state v in $\mathcal{M}$ for which $L(\alpha) \subseteq C_i$ is a feasible coding alternative for a given source symbol i in $\boldsymbol{B}$. Let $b \in \boldsymbol{B}$ be the upcoming symbol to encode. Since C_b is a codeword class with respect to $\mathcal{E}$, there is a state e in E such that the collection of C_b-successors of e contain a set E' from $\mathcal{E}$. We now encode i as $L(\beta)$, where β is a transition from v to e with label contained in C_i. Now continue the encoding with e and E'. □

Example 5: Consider again the collection $\boldsymbol{F}_b$, $b = 1, \ldots, 6$, from Example 2. Let $\mathcal{E} = \{E_1, E_2, E_3\}$, where

$$E_1 = \{00\}, E_2 = \{11\}, E_3 = \{01, 10\}.$$

It is not difficult to verify that the $\boldsymbol{F}_b$ are codeword classes with respect to this collection $\mathcal{E}$ of principal state-sets. Since the $\boldsymbol{F}_b$ are obviously disjoint, they constitute a one-symbol look-ahead BDB code according to Theorem 6. Moreover, it is easily verified that the sets

$$\boldsymbol{F}_7 = \{0111, 1000\}, \qquad \boldsymbol{F}_8 = \{0110, 0001, 1110, 1001\}$$

are also codeword classes with respect to $\mathcal{E}$. Consequently, $\boldsymbol{F}_1, \ldots, \boldsymbol{F}_4, \boldsymbol{F}_7, \boldsymbol{F}_8$ constitute a second BDB code. □

Example 5 (see also Example 7) illustrates a nice property of the description of a BDB code in terms of principal state-sets: If, from a good hunch, or by any other means, we "guess" to start with some "good" collection $\mathcal{E}$ of principal state-sets, then we may mechanically list *all* codeword classes with respect to $\mathcal{E}$, and just pick out the largest possible (or a very large) collection of mutually disjoint ones. This latter task, **SET PARTITIONING**, is computationally difficult ("NP-hard") in general, but can nevertheless be solved in many cases, even for rather large source alphabets. (This last point is exemplified by [10], see also Example 7.)

We will discuss methods to obtain "good" collections of principal state-sets when we relate this method to the method of *state combination* as described by Immink [12].

Our next result shows that each one-symbol look-ahead BDB code can be obtained from a suitable collection of principal state-sets. In order to state this result, we introduce some new notation. Let $\mathcal{M} = (V, A, L, \boldsymbol{F})$ be a FSTD, and let $\mathcal{W}_v$, $v \in V$, denote a collection of admissible prefix lists for a one-symbol look-ahead BDB encoder for $\mathcal{M}$ with respect to a partition $\{\boldsymbol{F}_b\}_{b \in \boldsymbol{B}}$ of $\boldsymbol{F}$. For each state v of $\mathcal{M}$ and for each symbol b from $\mathcal{W}_v$, let $E(v, b)$ denote the collection of all states of the form end(α) with α a transition of $\mathcal{M}$ with beg$(\alpha) = v$ and $L(\alpha) \in \boldsymbol{F}_b$. We denote the collection of all these sets $E(v, b)$ by $\mathcal{E}(\mathcal{W})$.

Theorem 7: With the above notation, each $\boldsymbol{F}_b$, $b \in \boldsymbol{B}$, is a codeword class with respect to the collection $\mathcal{E}(\mathcal{W})$.

Proof: Let $c \in \boldsymbol{B}$. We will show that $\boldsymbol{F}_c$ is a codeword class with respect to $\mathcal{E}(\mathcal{W})$. So let $E(v, b) \in \mathcal{E}(\mathcal{W})$, that is, $b \in \mathcal{W}_v$. By the properties of $\mathcal{W}$, there is a transition α from v to a state $w = $ end(α) with $L(\alpha) \in \boldsymbol{F}_c$ for which $c \in \mathcal{W}_w$. Hence $w \in E(v, b)$ and the set $E(w, c) \in \mathcal{E}(\mathcal{W})$ is defined and is contained in (in fact, equal to) the collection of $\boldsymbol{F}_c$-successors of w. □

The construction method suggested by Theorem 6, which we will call the *principal state-set method*, may be considered as a generalization of the method to construct block codes through *principal states*. In that case, all principal state-sets consist of a *single* state only, and the resulting BDB codes are block codes, that is, encoding requires no look-ahead at all. Further relations with the work of Franaszek on Bounded Delay codes ([3]–[6]) will be discussed in a later section.

The principal state-set method offers further advantages when one is looking for (large) one-symbol look-ahead BDB codes for a given constrained system, for various block-lengths n, that is, for BDB encoders for $\mathcal{G}^n$ for various values of n, where $\mathcal{G}$ is a FSTD that describes the constraint. We illustrate this for the case of $(1, \infty)$ RLL sequences in our next example. (See [13] for further examples of this kind.)

Example 6: Consider again the FSTD $\mathcal{G}$ in Figure 2 that describes the $(1, \infty)$ RLL constraint. For $n \geq 4$, let $\mathcal{M} = \mathcal{G}^n$ be the nth power of $\mathcal{G}$. (So the transitions in $\mathcal{M}$ are labeled with admissible words of length n.) Let $\mathcal{E}$ denote the collection of principal state-sets as in Example 5. In order to decide if a given set C of words of length n is a codeword class with respect to $\mathcal{E}$, it is sufficient to know the *type* of each word in C, that is, the first two and the last two symbols of each word, since these four symbols determine the successor relations and terminal states of transitions labeled with this word.

We say that an admissible word $x = x_1 \cdots x_n$ has *begin type* x_1x_2, *end type* $x_{n-1}x_n$ and *type* x_1x_2—$x_{n-1}x_n$. Admissible words of length at least four are of sixteen different possible types. (When dealing with an arbitrary constrained system $\mathcal{L}$ of finite type, the begin- and end-types of long enough words are states in the left- and right-Shannon cover for $\mathcal{L}$, respectively.) The *class type* of a set of admissible words C of length n is the collection of types of words in C, and the class type of C is a *codeword class type with respect to* $\mathcal{E}$ if C (and hence also any other collection C' of words of the same class type as C) is a codeword class with respect to $\mathcal{E}$.

It turns out that there are precisely 47 distinct inclusion-minimal codeword class types with respect to $\mathcal{E}$. Moreover, it is possible to derive expressions that enumerate admissible words of length n of a given type. Using these results, it can be shown that for each value of n with $n \geq 4$ there is a one-symbol look-ahead BDB encoder for $\mathcal{M} = \mathcal{G}^n$ with $N(n)$ codeword classes, where

$$N(n) = \lfloor F_{n+3}/2 \rfloor = F_{n+1} + \lfloor F_n/2 \rfloor. \tag{4}$$

In the above expressions, F_n denotes the nth Fibonacci number, defined by $F_1 = F_2 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 3$. Further details can be found in a future

paper [13]. In that paper it is also shown that no BDB code for this constraint and with block length n can have more than $N(n)$ codeword classes.

Observe that the case $n = 4$, $N(n) = 6$, is the subject of Examples 2 and 5. We also observe that, since $N(6) = 17$, there is a $(1, \infty)$ RLL BDB code with rate $m/n = 4/6$. We will return to this case in the next section. □

Example 7: This example concerns $(1, 8)$ RLL BDB codes. The common FSTD for this constraint (which is in fact the Shannon cover) has states 01^i and 10^i, $1 \leq i \leq 9$. As principal state-sets, we choose the following collections.

$$\{10^i\}, \quad 2 \leq i \leq 4, \qquad \{10, 01\},$$
$$\{01^i\}, \quad 2 \leq i \leq 4, \qquad \{01^i, 10^j\}, \quad 5 \leq i, j \leq 9, (i, j) \neq (9, 9).$$

In [10] we describe a one-symbol look-ahead rate $m/n = 8/12$ BDB code for this constraint with as principal state-sets the above collection. In fact, this paper lists a number of codeword class types, and then some counting shows that sufficiently many disjoint codeword classes of these class types can be formed. The same principal state-sets and codeword class types could be used for other values of the block-length n as well, as long as $n \geq 9$. (This last condition ensures that a forbidden word is confined to two consecutive codewords.) The results in that paper also show that some further simplifications in the search for codeword class types are possible. □

We now discuss the problem of how to find a "good" collection of principal state-sets for a given FSTD $\mathcal{M}$. Our main method is based on (approximate) eigenvectors of the adjacency matrix $D_{\mathcal{M}}$ corresponding to $\mathcal{M}$. Intuitively, this is appealing since a component of such a vector reflects in some way the amount of information that can be encoded from the corresponding state.

Unfortunately, these methods can only serve as a guideline, and some creativity is still required. An interesting and related method is the following. Suppose we try to find a code with N codeword classes. First, we construct an $[\mathcal{M}, N]$-approximate eigenvector ϕ with all components equal to zero, one, or two. Then we run the *state combination method* as described in [12] with $\mathcal{M}$ and ϕ to obtain "good" state combinations. Finally, we use the state combinations thus obtained as our collection of principal state-sets. In a later section we relate the state combination method to our work and give some more details on the above method.

5.5 State-trees for M-symbol look-ahead BDB codes

In this section we generalize our previous results for the case where $M = 1$ to all values of M. As before, we let $\mathcal{M} = (V, A, L, F)$ denote a FSTD representing a

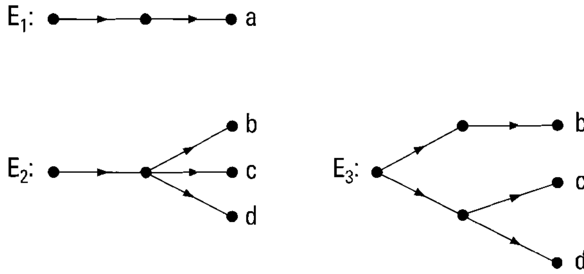

Figure 5: *Three state-trees.*

constrained system $\mathcal{L}$, we let $\{F_b\}_{b \in B}$ denote a partition of the labeling alphabet F of $\mathcal{M}$, and $\mathcal{W} : V \mapsto \mathcal{P}(B^M)$ denotes an M-symbol look-ahead assignment for $\mathcal{M}$ with respect to this partition. So the F_b are the codeword classes of an M-symbol look-ahead BDB encoder for $\mathcal{M}$ (or for $\mathcal{L}$) with source alphabet B.

In the previous section we defined the sets $E(v, b)$, $v \in V$, $b \in \mathcal{W}_v$, for the case where $M = 1$. We think of such a set as a *rooted directed tree* (with root corresponding to the state v) of *depth* one (all walks from the root to a *sink*, that is, a vertex in the tree without successors, have length one), with sinks labeled with states of $\mathcal{M}$. It is this structure that we intend to generalize.

We define a *state-tree* for $\mathcal{M}$ to be a rooted directed tree in which all walks from the root to a sink have the same length D, for some number $D \geq 0$, with sinks labeled with states from $\mathcal{M}$. The number D is the *depth* of the state-tree. (In the case where $D = 0$, the state-tree consists of a single vertex, labeled with a state of $\mathcal{M}$, with no arcs.) A vertex s of a state-tree E has *depth* K in E if the (unique) walk in E from the root to s has length K.

Example 8: Consider again the FSTD $\mathcal{M}$ in Figure 3. In Figure 5 we have three state-trees for $\mathcal{M}$, each of depth two. □

In what follows, we will use collections of state-trees to construct BDB encoders for $\mathcal{M}$, just as we did in the previous section with collections of principal state-sets. As before, the idea is to use the state-trees as "states" of the encoder, and to move from encoder state to encoder state according to upcoming codeword classes, in such a way that the resulting encoder state sequence defines a walk in $\mathcal{M}$ with a "correct" labeling. We think of a state-tree as describing in some way the collection of walks in $\mathcal{M}$ that are feasible encoding alternatives at a given encoding moment.

First we explain how to move from one state-tree to another. The idea is very simple, but a precise description is a bit complicated. Therefore, we first present a suggestive example.

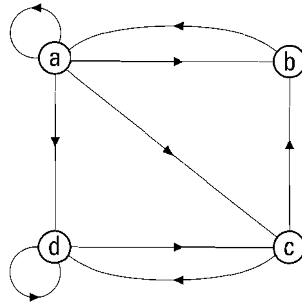

Figure 6: *The FSTD $\mathcal{M}$ for Example 9.*

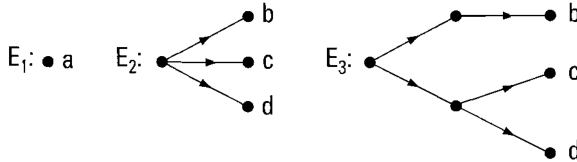

Figure 7: *Three state-trees for Example 9.*

Example 9: Consider the FSTD $\mathcal{M}$ in Figure 6. In Figure 7 we have three state-trees E_1, E_2, and E_3 for $\mathcal{M}$. Let $C = \{b, c, d\}$. We wish to suggest that C moves E_1 to E_2, E_2 to E_3, and E_3 to E_3. We have sketched the process in Figure 8. In this figure, we have first extended the E_i with leaves which represent C-successors in $\mathcal{M}$ of the labels of the sinks of the E_i. Then, in each of these extended trees, we see some E_j reappear as a sub-state-tree. (The appropriate root is indicated by a circle.) □

We now give the precise definition promised earlier. Let E and F be state-trees for $\mathcal{M}$, and let C be a subset of $\boldsymbol{F}$. Recall that a state w of $\mathcal{M}$ is a C-successor of a state v if there is a transition α in $\mathcal{M}$ with $L(\alpha) \in C$, $\text{beg}(\alpha) = v$, and $\text{end}(\alpha) = w$. We say that C *moves* E *to* F, written as $E \xrightarrow{C} F$, if either
(i) F has depth 0, so F consists of a single vertex labeled with some state w of $\mathcal{M}$, and this state w is C-successor of the label of some sink of E, or
(ii) F has depth $K \geq 1$, and it is possible to label the vertices of F at depth $K - 1$ with states of $\mathcal{M}$ such that the following holds. The label on the terminal vertex of a leaf of F is C-successor to the label on the initial vertex of this leaf. Moreover, the state-tree obtained by dropping all leaves from this relabelled F is a sub-state-tree of E.

Now let $\mathcal{E}$ denote a collection of state-trees (not necessarily all of the same depth) for $\mathcal{M}$. A subset C of the labeling alphabet $\boldsymbol{F}$ of $\mathcal{M}$ is a *codeword class with respect to* $\mathcal{E}$ if for each state-tree E in $\mathcal{E}$ there is a state-tree F in $\mathcal{E}$ such that

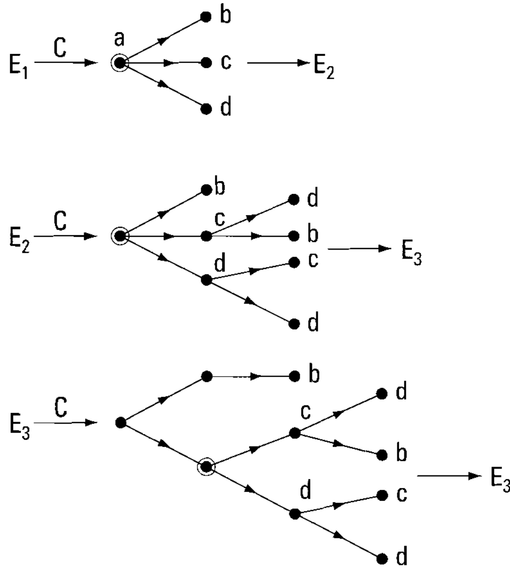

Figure 8: *How C moves the state-trees.*

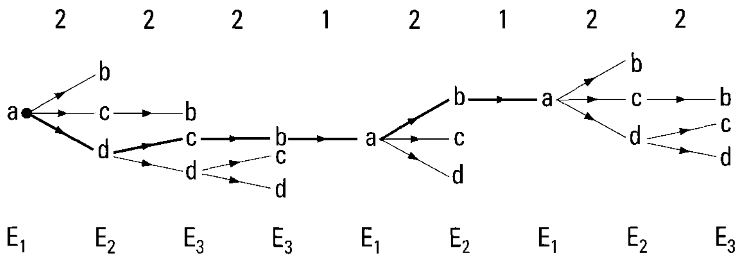

Figure 9: *Encoding of* 222121(22).

$E \xrightarrow{C} F$, that is, such that C moves E to F.

Theorem 8: Let $\mathcal{M} = (V, A, L, \boldsymbol{F})$ be a FSTD that generates a constrained system $\mathcal{L}$. Let $C_1, \ldots, C_N$ be mutually disjoint subsets of $\boldsymbol{F}$, and suppose that each C_i is a codeword class with respect to some given collection $\mathcal{E}$ of state-trees of $\mathcal{M}$. Then the C_i are the codeword classes of an M-symbol look-ahead BDB code for $\mathcal{L}$, where M is the maximal depth of the state-trees in $\mathcal{E}$.

A proof of the above theorem can be given along similar lines as the proof of Theorem 6. Instead of giving this proof, we present an example which illustrates the encoding process.

Example 10: Consider the FSTD $\mathcal{M}$ in Figure 3. In Figure 7 we have three state-trees E_1, E_2, and E_3 for $\mathcal{M}$. Put $\mathcal{E} = \{E_1, E_2, E_3\}$. Let $C_1 = \{a\}$ and $C_2 =$

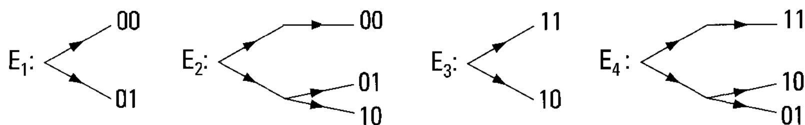

Figure 10: *Four state-trees for Example 11.*

$\{b, c, d\}$. It is easily verified that the C_i are codeword classes with respect to $\mathcal{E}$. (In Example 9 this is verified for C_2, since the transition from c to d in the FSTD in Figure 6 is never required.) According to Theorem 8, C_1 and C_2 constitute a two-symbol look-ahead BDB code for the constrained system generated by $\mathcal{M}$, with source alphabet $\boldsymbol{B} = \{1, 2\}$. In Figure 9 we show the process of encoding the source sequence 222121(22), starting from state-tree E_1. The heavy line indicates the walk in $\mathcal{M}$ taken by the encoder. Consequently, the sequence 222121(22) encodes as *dcbaba*. □

We will now show that, as expected, each BDB code arises from a collection of state-trees. Indeed, given an M-symbol look-ahead assignment $\mathcal{W}$ for $\mathcal{M}$, we may construct for each state v of $\mathcal{M}$ and for each $\boldsymbol{b} = b_1 \cdots b_M$ in $\mathcal{W}_v$ a state-tree $E(v, \boldsymbol{b})$ as follows. The vertices of $E(v, \boldsymbol{b})$ may be identified with those walks $\omega_1 \cdots \omega_k$ in $\mathcal{M}$ which generate the word $b_1 \cdots b_k$, for $k = 0, \ldots, M$, with an arc from $\omega_1 \cdots \omega_k$ to $\omega_1 \cdots \omega_k \omega_{k+1}$ if both walks correspond to vertices of $E(v, \boldsymbol{b})$. (So the root corresponds to the *empty walk* in $\mathcal{M}$.) Finally, a vertex at depth M corresponds to a walk $\omega_1 \cdots \omega_M$ in $\mathcal{M}$ that generates $b_1 \cdots b_M$ and is labeled with the state $\mathrm{end}(\omega_M)$ in $\mathcal{M}$. We denote the collection of all state-trees $E(v, \boldsymbol{b})$ with v a state of $\mathcal{M}$ and $\boldsymbol{b} \in \mathcal{W}_v$ by $\mathcal{E}(\mathcal{W})$. We now have the following.

Theorem 9: Let $\mathcal{M} = (V, A, L, \boldsymbol{F})$ be a FSTD, and let $\mathcal{W}$ denote an M-symbol look-ahead assignment for $\mathcal{M}$ with respect to a partition $\{\boldsymbol{F}_b\}_{b \in \boldsymbol{B}}$ of $\boldsymbol{F}$. Then each $\boldsymbol{F}_b$, $b \in \boldsymbol{B}$, is a codeword class with respect to the collection of state-trees $\mathcal{E}(\mathcal{W})$.

The construction method suggested by Theorem 8 will be referred to as the *state-tree method.* Everything that was said about the principal state-set method can be repeated here for the state-tree method.

In the remainder of this section, we will illustrate the state-tree method for the construction of BDB codes by some examples concerning BDB codes for (d, k) RLL sequences. As remarked earlier, "optimal" BDB codes for the $(1, \infty)$ RLL constraint require at most one-symbol look-ahead. We wonder if the same is true for other values of d and k as well. In fact, at first it is not at all evident that there are BDB codes for these constraints that require more than one-symbol look-ahead at all. Our next example presents such a BDB code.

Example 11: In this example we show that for $(1, \infty)$ RLL constrained sequences, there can be found "true" two-symbol look-ahead BDB codes. (Recall that the forbidden words for this constraint are 010 and 101.) We will use the notation of Example 6 for the types of codewords and codeword classes.

So let $\mathcal{M}$ be the nth power of the FSTD $\mathcal{G}$ in Figure 2, where n denotes some integer with $n \geq 6$. (This requirement ensures that all possible codeword types are realised by some codeword.) Consider the collection $\mathcal{E} = \{E_1, \ldots, E_4\}$ of state-trees in Figure 10. It is not difficult to verify that

$$C_1 = \{00\text{–}00,\ 11\text{–}01\},$$
$$C_2 = \{10\text{–}00,\ 01\text{–}01,\ 01\text{–}10\},$$
$$C_3 = \{10\text{–}00,\ 10\text{–}01\}$$

are types of codeword classes with respect to $\mathcal{E}$ (and, by the way, also with respect to $\{E_1, E_2\}$). Hence $\{C_1, C_2, C_3\}$ is a two-symbol look-ahead BDB code.

We claim that the above collection is *not* a one-symbol look-ahead BDB code. (The following is a typical proof of such a statement.) Indeed, consider the encoding of 2(2). We distinguish two cases.

(i) We encode as 01–01. Then the upcoming symbol 2 can only be encoded as 10–00. Consequently, it would not be possible to encode a sequence 223, since this last type has no successor in C_3.

(ii) We encode either as 10–00 or as 01–10. In both cases, the upcoming symbol 2 can be encoded only as 01–10 or as 01–01. Now, consider the encoding of 22(1). Again, we distinguish two cases.

(a) We choose 01–10 to encode the second symbol 2. Then we can encode the upcoming symbol 1 only as 00–00. Consequently, encoding of the sequence 2213 would not be possible, since the type 00–00 has no successor in C_3.

(b) We choose 01–10 to encode the second symbol 2. Then we can encode the upcoming symbol 1 only as 11–01. Consequently, encoding of the sequence 22123 would not be possible, as the reader may easily verify.

As a consequence of the above, if $\mathcal{C}$ is a collection of disjoint codeword classes with respect to $\mathcal{E}$, and if $\mathcal{C}$ contains a codeword class of each of the types C_1, C_2, and C_3, then $\mathcal{C}$ is an M-symbol look-ahead BDB code for $M = 2$, but *not* for $M = 1$. For example, let the codeword size $n = 7$. Then the collection

$$C_1 = \{0000000,\ 1100001\},$$
$$C_2 = \{1000000,\ 0110001,\ 0111110\},$$
$$C_3 = \{1001100,\ 1000001\},$$

constitutes a two-symbol look-ahead BDB code. (To these codeword classes, we may add the sets C_i' obtained from the C_i by interchanging 0 and 1.)

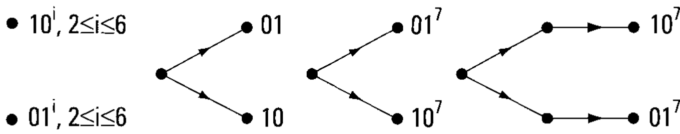

Figure 11: *The state-trees for Example 12.*

Due to the symmetry $0 \longleftrightarrow 1$ of the collection $\mathcal{E}$ of state-trees, there are evidently many other codeword class types. It is therefore possible to obtain reasonably large codes, although of course not larger than the codes in Example 6, since these are optimal. □

Example 12: Here we consider rate 2/3 $(1, k)$ RLL codes with codeword size $n = 6$. The following 17 codeword classes partition the 26 words of length six that do not contain 010 or 101.

$C_1 = \{000000\}$,	$C_{10} = \{110000\}$,
$C_2 = \{111111\}$,	$C_{11} = \{000001, 000110\}$,
$C_3 = \{001100\}$,	$C_{12} = \{111001, 111110\}$,
$C_4 = \{110011\}$,	$C_{13} = \{011100, 100000\}$,
$C_5 = \{111100\}$,	$C_{14} = \{011111, 100011\}$,
$C_6 = \{000011\}$,	$C_{15} = \{100111, 011000\}$,
$C_7 = \{000111\}$,	$C_{16} = \{110001, 011110, 100110\}$,
$C_8 = \{111000\}$,	$C_{17} = \{001110, 100001, 011001\}$.
$C_9 = \{001111\}$,	

It is not difficult to verify that these 17 codeword classes constitute a one-symbol look-ahead BDB code, with respect to the collection of principal state-sets $\{00\}$, $\{11\}$, and $\{01, 10\}$.

Next, we combine C_1 and C_2 to form a new codeword class

$$C' = \{000000, 111111\},$$

and we consider the collection of 16 codeword classes $\mathcal{C} = \{C', C_3, \ldots, C_{17}\}$. We claim that $\mathcal{C}$ is a one-symbol look-ahead BDB $(1, k)$ RLL code for $k = 11$, but not for $k = 10$. (Additional forbidden words are 0^{13} and 1^{13}.) Indeed, the reader may verify that each member of $\mathcal{C}$ is a codeword class with respect to the collection of principal state-sets

$$\{10, 01\}, \qquad \{10^i\}, \quad 2 \le i \le 7, \qquad \{01^i\}, \quad 2 \le i \le 7.$$

Moreover, we claim that $\mathcal{C}$ is a two-symbol look-ahead BDB (1, 10) RLL code. Indeed, the reader may verify that each member of $\mathcal{C}$ is codeword class with respect to the collection of state-trees in Figure 11.

Finally, we show that we can do equally well with a one-symbol look-ahead BDB code. So, let

$$C'_1 = \{000000, 011000\}, C'_2 = \{111111, 100111\},$$

and let $C' = \{C'_1, C'_2, C_3, \ldots, C_{14}, C_{16}, C_{17}\}$. The members of C' are mutually disjoint and are codeword classes with respect to the collection of principal state-sets

$$\{10, 01\}, \qquad \{10^i\}, \quad 2 \leq i \leq 6, \qquad \{01^i\}, \quad 2 \leq i \leq 6.$$

Consequently, C is a one-symbol look-ahead BDB $(1, 10)$ RLL code. □

Based on the above examples and many others, we offer the following conjecture. Let $M_n(d, k)$ denote the maximum size of a BDB code with codeword size n for the (d, k) RLL constraint.

Conjecture 1: For all n, d, and k, $M_n(d, k)$ is realized by a BDB code with look-ahead at most one.

(In a later section we describe an example showing that there may be larger block-decodable codes. But encoding that code requires an *unbounded* amount of look-ahead.) A similar conjecture may be stated for the case of (d, k)-constrained sequences, that is, binary sequences in which each run of zeroes has a length between d and k. For $d = 1$, this has been proved in [8]. Indeed, we do not know an example of a BDB code for this constraint with a larger size than the (known) optimal size of a block code (that is, a zero-symbol look-ahead BDB code). These two types of constraints are of course strongly related, and the same may be true for these two conjectures.

5.6 The state combination method

In this section we introduce an extension of the state combination method to construct BDB codes described in [12] (see also [11]). We link this method to our work, and we show that *each* BDB code may be constructed by state combination.

Both state combination and its well-known counterpart state splitting ([1]) are methods to transform a FSTD representing a given constrained system, with as ultimate goal to obtain a representation suitable for encoding purposes. However, the two methods differ in one important aspect: while the new FSTD obtained from state splitting still represents the *same* constrained system, this is not quite true for state combination. What is retained is a one-to-one correspondence between walks in the old and the new FSTD, and thus, also between words generated

by these FSTD's. Moreover, this correspondence acts as a kind of "built-in" encoding delay, as we will see.

So let $\mathcal{M} = (V, A, L, \boldsymbol{F})$ denote a FSTD, representing the constrained system $\mathcal{L} = \mathcal{L}(\mathcal{M})$, and let $U = \{u_1, \ldots, u_p\}$ be a collection of states of $\mathcal{M}$. A *state combination with U in $\mathcal{M}$* transforms $\mathcal{M}$ into a new FSTD $\mathcal{M}'$ by adding one new state, also denoted by U, to $\mathcal{M}$. The transitions involving this new state, and their corresponding labels, are the following.
(i) Each transition in $\mathcal{M}$ from some state in U to some state v not in U produces a transition in $\mathcal{M}'$ from state U to v carrying the same label.
(ii) Each collection of p transitions in $\mathcal{M}$ from a state v to each of the states u_i in U produces a transition in $\mathcal{M}'$ from v to state U with as label the set $\{f_1, \ldots, f_p\}$, where f_i denotes the label of the transition in $\mathcal{M}$ from v to u_i. Moreover, if v is contained in U, then $\mathcal{M}'$ has also a transition from U to U with label $\{f_1, \ldots, f_p\}$.
We observe that if the set U consists of a single state, then state combination with U essentially duplicates this state and all transitions involving it.

Now let $U^1, \ldots, U^k$, $U^i \subseteq V$, be sets of states of $\mathcal{M}$. A *round of state combinations with $U^1, \ldots, U^k$ on $\mathcal{M}$* consists of transforming $\mathcal{M}$ by successive state combinations with $U^1, U^2, \ldots, U^k$ (it does not matter in which order these state combinations are executed, as the reader may easily verify), followed by elimination of all original states of $\mathcal{M}$. Let $\hat{\mathcal{M}}$ be obtained from $\mathcal{M}$ by a round of state combinations with $U^1, \ldots, U^k$. The following proposition summarizes the main properties of $\hat{\mathcal{M}}$.

Proposition 10: (i) The FSTD $\hat{\mathcal{M}}$ has a transition from U^i to U^j with label E if and only if there is a state u in U^i and for each state w in U^j a transition ω_w from u to w such that $E = \{L(\omega_w) \mid w \in U^j\}$.
(ii) Each walk $\hat{\omega}_1 \cdots \hat{\omega}_{k+1}$ of length $k+1$ in $\hat{\mathcal{M}}$ uniquely determines a corresponding walk $\omega_1 \cdots \omega_k$ of length k in $\mathcal{M}$ with the property that $\mathrm{beg}(\omega_i) \in \mathrm{beg}(\hat{\omega}_i)$, $\mathrm{end}(\omega_i) \in \mathrm{end}(\hat{\omega}_i)$, and $L(\omega_i) \in \hat{L}(\hat{\omega}_i)$, $i = 1, \ldots, k$, where $\hat{L}(\hat{\omega}_i)$ denotes the label of the transition $\hat{\omega}_i$ in $\hat{\mathcal{M}}$.

Example 13: Consider again the FSTD $\mathcal{M}$ in Figure 3. One round of state combinations with $\{a\}$, $\{b\}$, and $\{c, d\}$ on $\mathcal{M}$ transforms $\mathcal{M}$ into the FSTD $\mathcal{M}'$ in Figure 12. Then a second round of state combinations, now with $\{\{a\}\}$ and $\{\{b\}, \{c, d\}\}$, transforms $\mathcal{M}'$ into the FSTD $\mathcal{M}''$ in Figure 13. □

The built-in encoding delay referred to earlier is illustrated by part (ii) of the above proposition. In fact, we can say more. With $\hat{\mathcal{M}}$ as in Proposition 10, the result of one round of state combinations applied to $\mathcal{M}$, let $\mathcal{M}^*$ be a FSTD obtained from $\hat{\mathcal{M}}$ by discarding possibly some of the transitions. A partition

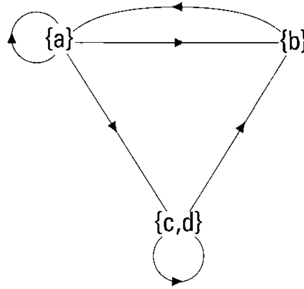

Figure 12: *The FSTD $\mathcal{M}'$ obtained after one round of state combinations.*

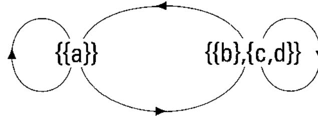

Figure 13: *The FSTD $\mathcal{M}''$ obtained after two rounds of state combinations.*

$\{\boldsymbol{F}_b\}_{b\in \boldsymbol{B}}$ of the labeling alphabet $\boldsymbol{F}$ of $\mathcal{M}$ is *consistent with* $\mathcal{M}^*$ if the label $\hat{L}(\hat{\alpha})$ of each transition $\hat{\alpha}$ of $\mathcal{M}^*$ is contained in some part $\boldsymbol{F}_b$. Such a partition defines a partition $\{\boldsymbol{F}_b^*\}_{b\in \mathbf{B}}$ of the labeling alphabet $\boldsymbol{F}^*$ of $\mathcal{M}^*$, with the part $\boldsymbol{F}_b^*$ consisting of all labels of $\mathcal{M}^*$ contained in $\boldsymbol{F}_b$.

Theorem 11: With the above notation, suppose that $\{\boldsymbol{F}_b\}_{b\in \boldsymbol{B}}$ is a partition that is consistent with $\mathcal{M}^*$ such that the corresponding partition $\{\boldsymbol{F}_b^*\}_{b\in \mathbf{B}}$ of $\boldsymbol{F}^*$ constitutes the codeword classes of an $(M-1)$-symbol look-ahead BDB encoder for $\mathcal{M}^*$. Then the $\boldsymbol{F}_b$, $b \in \boldsymbol{B}$, are the codeword classes of an M-symbol look-ahead encoder for $\mathcal{M}$.

Proof: Obvious, since by looking ahead M symbols, we obtain the next two transitions of the encoding path in $\mathcal{M}^*$, hence by part (ii) of Proposition 10 the next transition of the encoding path in $\mathcal{M}$. □

By a repeated application of the above theorem we see that we may construct an M-look-ahead BDB encoder for a FSTD $\mathcal{M}$ from a block code (that is, a 0-symbol look-ahead BDB encoder) for a FSTD that is obtained after M rounds of state combination in $\mathcal{M}$.

Example 14: Consider again the FSTD $\mathcal{M}''$ in Figure 13, obtained after two rounds of state combinations in the FSTD $\mathcal{M}$ in Figure 3. Since the partition of the labeling alphabet $\{a, b, c, d\}$ of $\mathcal{M}$ into the parts $\{a\}$ and $\{b, c, d\}$ is consistent with $\mathcal{M}''$, this partition constitutes the codeword classes of a two-symbol look-ahead BDB encoder for $\mathcal{M}$. This is of course the same BDB code that we already met in Example 10. □

State combination as proposed in [12] is a special case of the above where $\mathcal{M}$ is of Moore type with all state labels distinct, and where a state of $\mathcal{M}$ is used *at most once* in a combination, that is, the sets U^i representing the states to be combined are mutually disjoint; a state that is used in a state combination is eliminated immediately afterwards. Moreover, [12] only considers *one round* of state combinations. We refer to this paper for further examples.

We will now show that, conversely, each BDB encoder for a FSTD $\mathcal{M}$ may be obtained from rounds of suitable state combinations. So let $[\mathcal{M}, \varphi, \mathcal{W}]$ denote an M-symbol look-ahead BDB encoder, with $M \geq 1$, for a FSTD $\mathcal{M} = (V, A, L, \boldsymbol{F})$ with respect to the partition $\{\boldsymbol{F}_b\}_{b \in \boldsymbol{B}}$ of the labeling alphabet $\boldsymbol{F}$ of $\mathcal{M}$ induced by φ. Let v be a state of $\mathcal{M}$ and let $\boldsymbol{b} = b_1 \cdots b_M$ be a word in $\mathcal{W}_v$. By the defining properties of $\mathcal{W}$, we can find for each $b \in \boldsymbol{B}$ a transition α_b from v to a state w_b with $L(\alpha) \in \boldsymbol{F}_{b_1}$ and $b_2 \cdots b_M b \in \mathcal{W}_{w_b}$. Let $U(v, \boldsymbol{b})$ denote the collection of all the states w_b thus obtained. We write $\mathcal{U}(\mathcal{W})$ to denote the collection of all these sets $U(v, \boldsymbol{b})$.

Theorem 12: With the above notation, let the FSTD $\hat{\mathcal{M}}$ be obtained from one round of state combination on $\mathcal{M}$ with the sets contained in $\mathcal{U}(\mathcal{W})$. Then we can find an $(M-1)$-symbol look-ahead BDB encoder for the FSTD $\mathcal{M}^*$ obtained by eliminating certain transitions in $\hat{\mathcal{M}}$.

Proof: We begin by describing the FSTD $\mathcal{M}^*$. A transition in $\hat{\mathcal{M}}$ from X to Y, $X, Y \in \mathcal{U}(\mathcal{W})$, with label E, is retained in $\mathcal{M}^*$ if there are states v, w in $\mathcal{M}$ and $b_1, \ldots, b_{M+1}$ in $\boldsymbol{B}$ such that $b_1 \cdots b_M \in \mathcal{W}_v$, $b_2 \cdots b_{M+1} \in \mathcal{W}_w$, $X = U(v, b_1 \cdots b_M)$, $Y = U(w, b_2 \cdots b_{M+1})$, $w \in X$, and $E = \{L(\alpha_b) \mid b \in \boldsymbol{B}\}$, where α_b is a transition from w to a state $u \in Y$ with $L(\alpha)$ in $\boldsymbol{F}_{b_2}$ and $b_3 \cdots b_{M+1} b \in \mathcal{W}_{\mathrm{end}(\alpha_b)}$. Observe that the labeling set E of this transition of $\mathcal{M}^*$ satisfies $E \subseteq \boldsymbol{F}_{b_2}$. As a consequence, the given partition of $\boldsymbol{F}$ is consistent with $\mathcal{M}^*$.

For X a state of $\mathcal{M}^*$, we define W^*_X to consist of all words $b_2 \cdots b_M$ for which $X = U(v, b_1 b_2 \cdots b_M)$ for some state v and some b_1 with $b_1 b_2 \cdots b_M \in \mathcal{W}_v$. We claim that these sets W^*_X constitute an $(M-1)$-symbol look-ahead BDB encoder for $\mathcal{M}^*$, with respect to the partition of the labeling alphabet of $\mathcal{M}^*$ induced by the original partition of $\boldsymbol{F}$. Indeed, let $X \in \mathcal{U}(\mathcal{W})$, $b_2 \cdots b_M \in W^*_X$, and $b \in \boldsymbol{B}$. Let v and b_1 be such that $b_1 \cdots b_M \in \mathcal{W}_v$ and $X = U(v, b_1 \cdots b_M)$. Let α be a transition in $\mathcal{M}$ with $\mathrm{beg}(\alpha) = v$, $L(\alpha) \in \boldsymbol{F}_{b_1}$, and $b_2 \cdots b_M b \in \mathcal{W}_w$, where $w = \mathrm{end}(\alpha)$ (such a transition exists by the defining properties of $\mathcal{W}$). Then, as the reader may easily verify, $\mathcal{M}^*$ has a transition from X to $Y = U(w, b_2 \cdots b_M b)$, with label contained in $\boldsymbol{F}_{b_1}$, and $b_3 \cdots b_M b \in W^*_Y$ by definition of W^*_Y. □

We remark that the statement of Theorem 12 also holds with $\mathcal{U}(\mathcal{W})$ replaced by the collection $\mathcal{U}'(\mathcal{W})$ consisting of all sets $U'(v, b)$, $v \in V$, $b \in \boldsymbol{B}$, where

$U'(v, b)$ is the set of all states end(α) with beg(α) $= v$ and $L(\alpha) \in \boldsymbol{F}_b$. However, it can be shown that $\mathcal{U}(\mathcal{W}) \subseteq \mathcal{U}'(\mathcal{W})$, and each set in $\mathcal{U}'(\mathcal{W})$ is a union of sets in $\mathcal{U}(\mathcal{W})$.

Example 15: Let $\mathcal{M}$ be the FSTD in Figure 3, and let $\mathcal{W}$ be as in Example 3. If we apply the state combination described in Theorem 12 to $\mathcal{M}$, we obtain the FSTD $\mathcal{M}'$ in Figure 12. (If we use state combination with the collection $\mathcal{U}'(\mathcal{W})$ instead, then we obtain a superfluous state $\{b, c, d\}$.) A second application of Theorem 12, using the look-ahead assignment W^*, produces the FSTD $\mathcal{M}''$ in Figure 13. The two rounds of state combinations are identical to the ones described in Example 13. □

Example 16: Let $\mathcal{M}$ denote the fourth power of the FSTD in Figure 2. We leave it to the reader to verify that a round of state combinations with $\{00\}$, $\{11\}$, and $\{01, 10\}$ on $\mathcal{M}$, followed by the elimination of some of the transitions, can produce the one-symbol look-ahead BDB codes in Example 2 and Example 5.

As an alternative, we may also obtain these codes from state combination on the FSTD with as states the admissible words of length four for the $(1, \infty)$ RLL constraint and with the obvious transitions. This is the way in which these codes are constructed in [12]. □

Although the result in Theorem 12 is satisfactory from a theoretical point of view, it is not evident how to apply this result for the purpose of code construction. The main problem is of course to know which state combinations are useful. The results from [12] show however that good results can already be obtained by using only a very restricted set of state combinations, guided by an appropriate approximate eigenvector. (On the other hand, it can be shown that the BDB code in Example 12 cannot be constructed by the methods of [12], so *some* extension of these ideas is needed.) Moreover, we are also hampered by the following problem: Given a FSTD $\mathcal{M}$ and a number N, there is no easy way to determine whether or not there exists a 0-symbol look-ahead BDB encoder for $\mathcal{M}$ of size N. (This problem has been shown to be NP-complete in [20].) Again, in the case where $\mathcal{M}$ is obtained from state combinations on some other FSTD, the results from [12] show that as long as the increase of the number of states is small, the above search problem (referred to as the source-to-state assignment problem in [12], section 3.2.1) can often be handled.

5.7 Stable state-sets

Let $\mathcal{M} = (V, A, L, \boldsymbol{F})$ be a FSTD, and let $\{\boldsymbol{F}_b\}_{b \in \boldsymbol{B}}$ denote a *fixed* partition of the labeling alphabet $\boldsymbol{F}$ of $\mathcal{M}$. A natural question is to ask whether this parti-

tion constitutes the codeword classes of a BDB encoder for $\mathcal{M}$, that is, whether there exist admissible prefix lists for the states of $\mathcal{M}$ with respect to this partition. Sometimes this problem is easy.

Example 17: In this example, we consider a code for the $(1, \infty)$ RLL constraint with word length $n = 3$. So let $\mathcal{M}$ denote the third power of the FSTD in Figure 2, and consider the following partition of the labeling alphabet of $\mathcal{M}$.

$$\boldsymbol{F}_1 = \{000\}, \qquad \boldsymbol{F}_3 = \{011, 100\},$$
$$\boldsymbol{F}_2 = \{111\}, \qquad \boldsymbol{F}_4 = \{001, 110\}.$$

It is easily verified that the $\boldsymbol{F}_i$ are the codeword classes of a code, encoding of which requires an *unbounded* amount of look-ahead. Indeed, consider the encoding of $444 \cdots 4x$, where x is either 1 or 2. Since both 001 and 110 cannot succeed themselves, a sequence of fours has precisely two encodings, namely $001.110.001. \cdots$ and $110.001.110. \cdots$, one ending in 001 (so that $x = 1$ next cannot be encoded) and the other ending in 110 so that $x = 2$ cannot be encoded). □

However the problem becomes complicated if the number of parts gets large. The question can also be handled if we specify beforehand the maximum amount of look-ahead that may be used, by a method similar to the one in Example 11. But for all that we know the required amount of look-ahead may be larger than this maximum. Fortunately, this problem can be solved. We shall show that for each BDB code there exists a unique inclusion-maximal collection of *stable state-sets* such that encoding can be described as moving from one stable state-set to the next, employing a variable but *bounded* amount of look-ahead. (In the case of one-symbol look-ahead BDB codes, the principal state-sets associated with the code are all stable state-sets.) Conversely, if we can find a collection of stable state-sets for our partition, then this partition constitutes a BDB code. As a consequence of this description, we show the existence of a number M, depending only on the number of states $|V|$ of $\mathcal{M}$, such that a BDB encoder for $\mathcal{M}$ requires a look-ahead of at most M symbols.

We introduce some additional notation. If $\omega = \omega_1 \cdots \omega_n$ is a walk in $\mathcal{M}$, then we say that the walk ω generates the word $\boldsymbol{b} = b_1 \cdots b_n$ over $\boldsymbol{B}$ if $L(\omega_i) \in \boldsymbol{F}_{b_i}$, for all i. We may in fact consider the walk ω as a walk in the FSTD $\mathcal{M}^* = (V, A, \mathcal{L}^*, \boldsymbol{B})$ obtained from $\mathcal{M}$ by replacing the label of each transition α by $\mathcal{L}^*(\alpha) = b(\alpha)$, where $b = b(\alpha)$ denotes the (unique) symbol in $\boldsymbol{B}$ for which $L(\alpha) \in \boldsymbol{F}_b$. For two subsets X, Y of the state set V of $\mathcal{M}$ and for a word $\boldsymbol{b} = b_1 \cdots b_n$ over $\boldsymbol{B}$, we write $X \xrightarrow{\boldsymbol{b}} Y$ if there is a state $x \in X$ and a walk from x to each state y in Y that generates $\boldsymbol{b}$. Moreover, we let $X\boldsymbol{b}$ denote the collection

of all states in $\mathcal{M}$ that can be reached from X by a walk in $\mathcal{M}$ that generates $\boldsymbol{b}$. Our first result is an easy consequence of the definitions. We leave the proof to the reader.

Proposition 13: Let X, Y, and Z be subsets of V, and let $\boldsymbol{b}$ and $\boldsymbol{c}$ be words over $\boldsymbol{B}$. If $X \xrightarrow{\boldsymbol{b}} Y$ and $Y \xrightarrow{\boldsymbol{c}} Z$, then $X \xrightarrow{\boldsymbol{bc}} Z$.

A collection $\mathcal{S}$ of (nonempty) subsets of the state set V of $\mathcal{M}$ is *stable* with respect to the given partition of $\boldsymbol{F}$ if for each X in $\mathcal{S}$ and for each sequence $b_1 b_2 \cdots$ over $\boldsymbol{B}$ there is a set Y in $\mathcal{S}$ and an integer $n \geq 1$ such that $X \xrightarrow{b_1 \cdots b_n} Y$ holds. If there always is such an n with $n \leq K$, then $\mathcal{S}$ is called K*-stable.* We refer to members of a stable $\mathcal{S}$ as *stable state-sets.* We will speak simply of a stable collection $\mathcal{S}$ if it is clear from the context to which partition we refer. Our next result also follows immediately from the definitions.

Proposition 14: (i) If $\mathcal{S}$ is stable and if $X \subseteq Y$, $X \in \mathcal{S}$, then the collection $\mathcal{S} \cup \{Y\}$ is again stable.
(ii) If $\mathcal{S}_1$ and $\mathcal{S}_2$ are stable, then the union $\mathcal{S}_1 \cup \mathcal{S}_2$ is again stable.

By Proposition 14, there either exists a *unique* inclusion-maximal collection of stable state-sets with respect to a given partition of $\boldsymbol{F}$, or there does not exist such a collection at all.

Our next two results illustrate the importance of the above concepts.

Theorem 15: If a collection $\mathcal{S}$ is stable, then $\mathcal{S}$ is K-stable, for some number K that is bounded from above by a function of $|V|$.

Proof: Let $X = \{x_1, \ldots, x_m\}$ be a subset of V, and let $b_1 b_2 \cdots$ be a sequence over $\boldsymbol{B}$. Let $Y_i^{(k)}$, $1 \leq i \leq m$, $k \geq 0$, denote the collection of all states y that can be reached from x_i by a walk in $\mathcal{M}$ that generates the initial part $b_1 \cdots b_k$ of the sequence. (For $k = 0$, put $Y_i^{(0)} = \{x_i\}$.) Write $Y^{(k)} = (Y_1^{(k)}, \ldots, Y_m^{(k)})$. There are only finitely many distinct sets $Y^{(k)}$; consequently there are integers a and $p \geq 1$ such that $Y^{(a)} = Y^{(a+p)}$. Now we distinguish two cases.
(i) Some collection $Y = Y^{(k)}$, $0 \leq k \leq a + p - 1$, is a stable state-set (i.e., Y is contained in $\mathcal{S}$). In that case, we have $X \xrightarrow{b_1 \cdots b_k} Y$, and k is bounded by the number K of possible distinct collections $Y^{(k)}$. A crude upper bound for this number is

$$K \leq \left(2^{|V|}\right)^{|V|-1}. \tag{5}$$

(ii) No collection $Y = Y^{(k)}$, $0 \leq k \leq a + p - 1$, is contained in $\mathcal{S}$. Define the (ultimately periodic) sequence $c_1 c_2 \cdots$ over $\boldsymbol{B}$ by letting $c_k = b_k$, $1 \leq k \leq a + p - 1$, and $c_k = c_{k-p}$, if $k \geq a + p$. Obviously, there is no $k \geq 1$ such that $X \xrightarrow{c_1 \cdots c_k} Y$ with Y in $\mathcal{S}$, so X is not a stable state-set. □

Theorem 16: The parts $\boldsymbol{F}_b$, $b \in \boldsymbol{B}$, constitute the codeword classes of a BDB encoder for $\mathcal{M}$ if and only if there exists a stable collection $\mathcal{S}$ with respect to this partition.

Proof: (i) Suppose that $\mathcal{S}$ is stable with respect to the given partition. Then, by Theorem 15, $\mathcal{S}$ is K-stable, for some $K \geq 1$. We claim that the partition constitutes the codeword classes of an M-symbol look-ahead BDB encoder for $\mathcal{M}$, with $M = 2K - 1$. It is possible to define an M-symbol look-ahead assignment for $\mathcal{M}$, but instead we shall sketch an argument that encoding is possible, never looking ahead more than M symbols. Our encoder will have encoder states corresponding to the stable state-sets in $\mathcal{S}$. If we are in state $X \in \mathcal{S}$, and we are to encode $b_1 b_2 \cdots$, then by the defining properties of $\mathcal{S}$ we can find a number $n \leq K$ such that $X \stackrel{b_1 \cdots b_n}{\longrightarrow} Y$, for some $Y \in \mathcal{S}$, that is, each state in Y can be reached from some fixed state x in X by a walk in $\mathcal{M}$ that generates $b_1 \cdots b_n$. Similarly, there is a number $m \leq K$ such that each state in Z can be reached from some fixed state y in Y by a walk in $\mathcal{M}$ that generates $b_{n+1} \cdots b_{n+m}$. Then we encode $b_1 \cdots b_n$ with the word over $\boldsymbol{F}$ generated by the walk from x to y in $\mathcal{M}$ that generates $b_1 \cdots b_n$. Consequently, the encoding of b_i, $1 \leq i \leq n$, requires a look-ahead of at most $n + m - 1$ symbols. Since $n + m - 1 \leq 2K - 1 = M$, our claim is proved.
(ii) Now suppose that the given partition constitutes the codeword classes of a BDB encoder for $\mathcal{M}$, that is, there exists admissible prefix lists $\mathcal{W}_v \subseteq \boldsymbol{B}^M$, $v \in V$, for the states of $\mathcal{M}$ with respect to this partition, for some integer M. Define the set $U(v, \boldsymbol{b})$, for all states v and all words $\boldsymbol{b} \in \mathcal{W}_v$, as the collection of all states that can be reached from v by a walk in $\mathcal{M}$ that generates the word $\boldsymbol{b}$. It is easily verified that the collection $\mathcal{W}$ of all these sets $U(v, \boldsymbol{b})$ is M-stable with respect to the given partition. □

The above ideas may be used to prove the non-existence claim in Example 3.5 from [9]. Another observation that is useful in such cases is the following. If the FSTD $\mathcal{M}$ admits a look-ahead encoder with source alphabet $\boldsymbol{B}$, then there exists a stable collection $\mathcal{S}$ and an $[\mathcal{M}, |\boldsymbol{B}|]$-approximate eigenvector $\phi \leq \mathbf{1}$ such that $\sum_{v \in X} \phi_v \geq 1$ for each $X \in \mathcal{S}$. We leave further details to the interested reader.

As a result of the above theorems, we have a decision procedure for the following.

BDB: Given a constrained system $\mathcal{L}$ of finite type, a codeword length n and a source alphabet size m, does there exists a BDB code for $\mathcal{L}$ with codeword length n and source alphabet of size m?

In view of the observations at the end of Section 5.3, this result in turn implies that we also have a decision problem for the following.

BD: Given a constrained system $\mathcal{L}$ of finite type, a codeword length n, a source alphabet size m, and a window size T, does there exists a bounded-delay encod-

able sliding-block decodable code for $\mathcal{L}$ with codeword length n, source alphabet of size m, and decoding window of size T?
Note that the collection of BD codes referred to in **BD** are precisely the collection of codes that can be constructed by the state-splitting algorithm.

5.8 Discussion

We have defined and investigated the class of bounded-delay encodable, block-decodable (BDB) codes. It is shown that M-symbol look-ahead BDB codes can always be represented with the aid of a collection of principal state-sets (if $M \leq 1$) or state-trees (for general M). This result provides a practical code construction method for one-symbol look-ahead BDB codes (see e.g. [10]). We have also generalized the state-combination method introduced in [12]. A third description in terms of stable state-sets establishes the existence of decision procedures for some basic coding problems.

Acknowledgments

I would like to thank the referees for a number of helpful suggestions, and my colleagues Stan Baggen, Jack v. Lint, Kees Schouhamer Immink, and Ludo Tolhuizen for some fruitful discussions and for comments on earlier versions of this paper.

Appendix A

In this appendix, we collect some of our more technical results concerning BDB codes. We begin with the formal definition of BDB codes. The definition is meant to capture our intuitive idea that a BDB code should have a finite-state encoder that encodes all sequences of source symbols, possibly with prescribed initial part, and does so with a *bounded* encoding delay, or, equivalently, using a *bounded* amount of look-ahead.

Let $\mathcal{L}$ be a constrained system over an alphabet $\boldsymbol{F}$, and let $\{\boldsymbol{F}_b\}_{b\in \boldsymbol{B}}$ be a partition of $\boldsymbol{F}$. An *M-symbol look-ahead bounded-delay encodable, block-decodable (BDB) code* with *source alphabet* $\boldsymbol{B}$ and *codeword classes* $\boldsymbol{F}_b$, $b \in \boldsymbol{B}$, is a finite collection of (encoder) states S together with a map ψ that associates with some (but not necessarily all) pairs $(s, b_0 \cdots b_M)$, $s \in S$, $b_0 \cdots b_M \in \boldsymbol{B}^{M+1}$, a new state t in S and a symbol f in $\boldsymbol{F}$ such that the following holds. There is a word $c_0 \cdots c_N$ over $\boldsymbol{B}$ and a *starter* u in S such that ψ, starting in u, encodes all

sequences $b_0b_1\cdots b_{n+M}$ over $\boldsymbol{B}$ with $b_i = c_i$, $i = 0, \ldots, \min(n + M, N)$, by producing a sequence of states $s_0 = u, s_1, \ldots, s_{n+1}$ and a word $f_0 \cdots f_n$ over $\boldsymbol{F}$ with $f_i \in \boldsymbol{F}_{b_i}$ for which ψ is defined on all pairs $(s_i, b_i \cdots b_{i+M})$, with image (s_{i+1}, f_i). In other words, we require that encoding as described by ψ, starting in state u, of sequences over $\boldsymbol{B}$ with prescribed initial part, *never* gets stuck.

The above formal definition of BDB codes is rather complicated. Fortunately, these codes have a much simpler equivalent description in terms of look-ahead assignments, which is the statement of Theorem 2.

Proof of Theorem 2:
Let us assume that the pair S, ψ, determines an M-symbol look-ahead BDB encoder for a constrained system $\mathcal{L}$ over an alphabet $\boldsymbol{F}$, with codeword classes $\boldsymbol{F}_b$, $b \in \boldsymbol{B}$. Let u denote the starter state in S, and suppose that all sequences over $\boldsymbol{B}$ with initial part equal to the word $c = c_0 \cdots c_N$ can be encoded by ψ, starting in state u. We define a FSTD $\mathcal{M}$, with S as its collection of states, and sets $\mathcal{W}_s$, $s \in S$, of words over $\boldsymbol{B}$ of length M, as follows. According to our assumptions, any sequence $\boldsymbol{b} = b_0b_1\cdots$ with $b_i = c_i$, $i = 0, \ldots, N$, determines a sequence $\boldsymbol{s} = \boldsymbol{s}(\boldsymbol{b}) = s_0s_1\cdots$ of states in S with $s_0 = u$, and a sequence $\boldsymbol{f} = \boldsymbol{f}(\boldsymbol{b}) = f_0f_1\cdots$ of symbols in $\boldsymbol{F}$ with $f_n \in \boldsymbol{F}_{b_n}$ for all $n \geq 0$ such that $\psi(s_n, b_n \cdots b_{n+M})$ is defined and equals (s_{n+1}, f_n). The FSTD $\mathcal{M}$ will have a transition from state s to state s' with label f from $\boldsymbol{F}$ whenever for some sequence $\boldsymbol{b}$ as above, with corresponding sequences $\boldsymbol{s}$ and $\boldsymbol{f}$, and for some n, we have $s = s_n$, $s' = s_{n+1}$, and $f = f_n$. Moreover, in this situation we also include $b_n \cdots b_{n+M-1}$ in $\mathcal{W}_s$. By letting $b = b_{n+M}$ vary over $\boldsymbol{B}$, we see that we can always obtain a transition in $\mathcal{M}$ to satisfy part (b) in Definition 1 for a look-ahead assignment on $\mathcal{M}$, as required. □

Proof of Theorem 3:
According to Theorem 2, we may assume that the given M-symbol look-ahead code for the constrained system $\mathcal{L}$ over $\boldsymbol{F}$, with codeword classes $\boldsymbol{F}_b$, $b \in \boldsymbol{B}$, say, is described by means of a BDB encoder $[\mathcal{G}, \phi, \mathcal{U}]$ for some FSTD $\mathcal{G}$ that generates a subsystem of $\mathcal{L}$. Let $\mathcal{M} = (V, A, L, \boldsymbol{F})$ have memory m, anticipation a, and let $\mathcal{M}$ generate $\mathcal{L}$. Our construction crucially depends on the following observation. By definition of m and a, each walk $\alpha_{-m}\cdots\alpha_0\cdots\alpha_a$ in $\mathcal{G}$, generating the word $\boldsymbol{f} = f_{-m}\cdots f_a$ in $\mathcal{L}$, say, determines a transition $\omega = \omega[\alpha_{-m}\cdots\alpha_a]$ in $\mathcal{M}$ such that each walk $\omega_{-m}\cdots\omega_a$ in $\mathcal{M}$ that generates $\boldsymbol{f}$ satisfies $\omega_0 = \omega$. Moreover, if α_{a+1} is a successor of α_a in $\mathcal{G}$, then $\omega' = \omega[\alpha_{-m+1}\cdots\alpha_{a+1}]$ is a successor of ω in $\mathcal{M}$.

In order to construct an $(M+a)$-symbol look-ahead BDB encoder $[\mathcal{M}, \varphi, \mathcal{W}]$ for $\mathcal{M}$, with respect to the $\boldsymbol{F}_b$, $b \in \boldsymbol{B}$, we now proceed as follows. For each state v of $\mathcal{M}$, we let $b_1\cdots b_{M+a}$ be contained in $\mathcal{W}_v$ if and only if there exists a

walk $\alpha_{-m} \cdots \alpha_a$ in $\mathcal{G}$ such that
(1) $b_1 \cdots b_M \in \mathcal{U}_{\mathrm{end}(\alpha_0)}$,
(2) $\alpha_1 \cdots \alpha_a$ is the walk from end(α_0) in $\mathcal{G}$ determined by $b_1 \cdots b_{M+a}$ and $\mathcal{U}$, and
(3) the transition $\omega = \omega[\alpha_{-m} \cdots \alpha_a]$ in $\mathcal{M}$ ends in v.
It is not difficult to see that the sets $\mathcal{W}_v$ thus obtained define a look-ahead encoder for $\mathcal{M}$. Indeed, in the above situation, a further symbol $b = b_{M+a+1}$ from $\boldsymbol{B}$ determines a further transition α_{a+1} in $\mathcal{G}$, and then $b_2 \cdots b_{M+1} \in \mathcal{U}_{\mathrm{end}(\alpha_1)}$ and the transition $\omega' = \omega[\alpha_{-m+1} \cdots \alpha_{a+1}]$ in $\mathcal{M}$ satisfies beg(ω') = end(ω) = v. So the transition ω' is the transition needed to satisfy part (b) of Definition 1 as required. □

Appendix B

Here, we add a few remarks on the interpretation of the results in Appendix A. In general, the actual encoding resulting from a given M-symbol look-ahead encoder need not be completely determined. (At certain stages, there may be more than one possible encoding transition.) In practice, this does not bother us: we can just pick one, and each choice will still lead to a sequence that gets decoded properly.

Nevertheless, in any practical application the encoder function need to be further specified. Note that such a specification influences the encoder and therefore also the resulting *code system*, the collection of sequences that can be produced by the encoder. Note furthermore that not each possible *encoder* for a given BDB code for some constrained system $\mathcal{L}$ (i.e., for a given *decoder*) can be obtained from a look-ahead encoder based on the minimal deterministic presentation (the Shannon cover or Fisher cover) $\mathcal{M}$ for $\mathcal{L}$ simply by deleting certain transitions. Indeed, the resulting code system of such an encoder would necessarily be deterministic, but not all sliding-block decodable codes for constraints of finite type need to have a code system of finite type. However, as shown by Theorems 2 and 3, any such encoder can be "simulated" within a look-ahead encoder for $\mathcal{M}$. (In fact, Theorem 2 delivers a look-ahead encoder from the given finite-state encoder that has the *same* code system; this is not true for Theorem 3.)

These remarks are well-illustrated by [2, Example 11.4]. In that example, the encoder in Figure 14 is investigated. Note that this encoder is a 0-symbol look-ahead encoder for a block-decodable code, with codeword classes $\{a\}$ and $\{b, c\}$. The authors show that the code system of this code is not of finite type, but is contained in a constraint of finite type with set of forbidden words $\{bb, cc, bab, cac\}$. The Shannon cover of this constrained system is the five-state FSTD in Figure 15.

From the results in [2] combined with our work in [9], we know that, as in the case of the state-splitting algorithm, one further operation on block-decodable

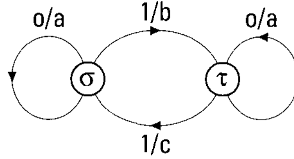

Figure 14: *An encoder for a block-decodable code.*

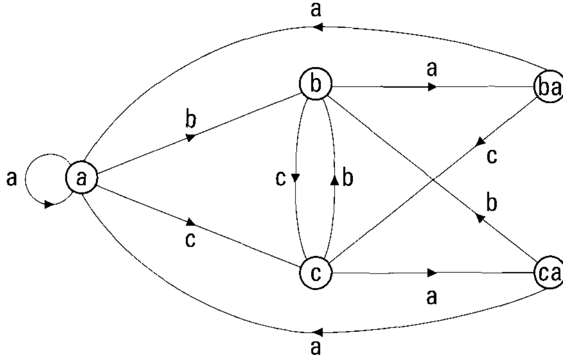

Figure 15: *The Shannon cover.*

look-ahead encoders based on Shannon covers need to be allowed if we want to be able to obtain *all* encoders for block-decodable codes. For example, in the case of [2, Example 11.4], the desired encoder is "hidden" inside the block-decodable look-ahead encoder obtained by means of Theorems 2 and 3; a further operation of "state-expansion" ("copying" of states) is needed in order to obtain the desired encoder. In this example, state a must be copied.

In general, we can obtain a fully specified look-ahead encoder by replacing in $\mathcal{M}$ each state v by states v^{ν}, where ν ranges over the states of $\mathcal{G}$, and by adapting the proof of Theorem 3 accordingly. (The superscript ν now indicates that state v is entered while the encoder $\mathcal{G}$ entered state ν.) Moreover, in the case that the code word labeling of the encoder $\mathcal{G}$ is of finite type, with memory T, say, then (note that $\mathcal{G}$, being an encoder for a block-decodable code, is deterministic) knowledge of the T previous transitions of the path in $\mathcal{M}$ by which a state v is entered is sufficient to determine the corresponding encoder state ν in $\mathcal{G}$. As a consequence, in the finite-type case we can obtain a fully specified look-ahead encoder based on the Tth history graph $\mathcal{M}_T$ of $\mathcal{M}$ (see [9] for the definition of the history graph). Further details concerning such constructions are left to the reader.

The above reasoning can easily be transformed into a proof for Theorem 5 in the case of block-decodable codes. The general case follows from the results in [9] (see also Chapter 4, Appendix C).

References

[1] R.L. Adler, D. Coppersmith, and M. Hassner, "Algorithms for sliding block codes. An application of symbolic dynamics to information theory", *IEEE Trans. Inform. Theory*, vol. IT-29, no. 1, pp. 5-22, Jan. 1983.

[2] J. Ashley and B.H. Marcus, "Canonical encoders for sliding block decoders", *Siam J. Disc. Math.*, vol. 8, no. 4, pp. 555–605, Nov. 1995.

[3] P.A. Franaszek, "On future-dependent block-coding for input-restricted channels", IBM J. Res. Develop., vol. 23, pp. 75–81, Jan. 1979.

[4] P.A. Franaszek, "Synchronous bounded delay coding for input-restricted channels", IBM J. Res. Develop., vol. 24, pp. 43–48, Jan. 1980.

[5] P.A. Franaszek, "A general method for channel coding", IBM J. Res. Develop., vol. 24, pp. 638–641, September 1980.

[6] P.A. Franaszek, "Construction of bounded delay codes for discrete noiseless channels", IBM J. Res. Develop., vol. 26, pp. 506–514, July 1982.

[7] P.A. Franaszek, "Coding for constrained channels: a comparison of two approaches", IBM J. Res. Develop., vol. 33, pp. 602–608, Nov. 1989.

[8] J. Gu and T. Fuja, " A note on look-ahead in the encoders of block-decodable (d, k) codes", preprint.

[9] H.D.L. Hollmann, "On the construction of bounded-delay encodable codes for constrained systems", *IEEE Trans. Inform. Theory*, vol. IT-41, no. 5, pp. 1354-1378, September 1995.

[10] H.D.L. Hollmann, "A block-decodable $(d, k) = (1, 8)$ runlength-limited rate $8/12$ code", *IEEE Trans. Inform. Theory*, vol. IT-40, no. 4, pp. 1292-1296, July 1994.

[11] H.D.L. Hollmann, "Bounded-delay encodable codes for constrained systems from state combination and state splitting", *Benelux Symp. Inform. Theory*, Veldhoven, May 1993.

[12] K.A.S. Immink, "Block-decodable runlength-limited codes via look-ahead technique", *Philips J. Res.*, vol. 46, no. 6, pp. 293–310, 1992.

[13] K.A.S. Immink and H.D.L. Hollmann, "Enumeration of Bounded delay encodable, Block-decodable runlength-limited codes", in preparation.

[14] K.A.S. Immink, *Coding techniques for digital recorders*, Englewood Cliffs, NY: Prentice-Hall, 1991.

[15] R. Karabed and B.H. Marcus, "Sliding-block coding for input-restricted channels", *IEEE Trans. Inform. Theory*, vol. IT-34, no. 1, pp. 2–26, Jan. 1988.

[16] A. Lempel and M. Cohn, "Look-ahead coding for input-restricted channels", *IEEE Trans. Inform. Theory*, vol. IT-28, pp. 933–937, Nov. 1982.

[17] B.H. Marcus and R. Roth, "Bounds on the number of states in encoder graphs for input-constrained channels", *IEEE Trans. Inform. Theory*, vol. IT-37, no. 3, sec. VII, pp. 742–758, May 1991.

[18] B.H Marcus, P.H. Siegel, and J.K. Wolf, "Finite-state modulation codes for data storage", IEEE J-SAC, vol. 10, no. 1, pp. 5–37, Jan. 1992.

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

[20] P.H. Siegel, "On the complexity of limiting error propagation in sliding block codes", *Proc. of the Second IBM Symp. on Coding and Error Control*, San Jose, California, Jan. 1985.

Chapter 6

A block-decodable $(d, k) = (1, 8)$ runlength-limited rate $8/12$ code

Abstract — **We describe a $(d, k) = (1, 8)$ runlength-limited (RLL) rate $8/12$ code with fixed code word length 12. The code is block-decodable; a code word can be decoded without knowledge of preceding or succeeding code words. The code belongs to the class of Bounded Delay Block-decodable (BDB) codes with one symbol (8 bits) look-ahead. Due to its format, this code is particularly attractive for use in combination with error correcting codes such as Reed-Solomon codes over the finite field GF(2^8).**

Index Terms — **RLL code, BDB code, block-decodable code, principal state-set, look-ahead coding.**

6.1 Introduction

A (d, k) RLL sequence is a binary sequence in which each run of like symbols has a length between $d + 1$ and $k + 1$. RLL sequences as well as (d, k) sequences (binary sequences describing the transitions in RLL sequences) have found many applications in various high-capacity storage media. The use of these sequences is discussed in detail in [4], where also further references can be found.

In this correspondence we describe a $(1, 8)$ RLL block-decodable rate $8/12$ code with code word length 12 bits. Here block-decodable, a term introduced in [4], refers to the fact that each code word, or *block*, decodes into a unique (8-bit) source word, irrespective of context. Encoding can be achieved with a finite state encoder; this encoder will be implicit in our description of the code. Due

---

© 1994, IEEE. Reprinted, with permission, from IEEE Transactions on Information Theory vol. 40, no. 4, pp. 1292–1296, July 1994.

to its format this code has favorable error propagation properties, especially when used in combination with error correcting codes such as Reed-Solomon codes over the finite field GF(2^8) (e.g., CIRC [5]). The code belongs to the family of Bounded Delay Block-decodable (BDB) codes with one symbol (here eight bits) look-ahead during encoding [1].

Bounded delay (BD) codes, a family of codes employing look-ahead encoding techniques, were introduced by Franaszek [7]. He described a necessary condition for their existence in terms of an *approximate eigenvector*, and showed how such a vector could be of help in code construction. (See also [8].) Franaszek also mentioned block-decodable BD codes, or "instantaneously decodable" codes as he named them.

Franaszek's methods do not constitute a polynomial-time algorithm. A breakthrough in that respect was the discovery of the ACH algorithm [6]. This algorithm produces a code by applying rounds of "state splitting" on the constraint graph. As in the BD method, the construction is guided by an approximate eigenvector.

Recently, the subject of code construction received new impetus with the work of Immink [4]. In this paper many new BDB (block-decodable) RLL codes, with sometimes surprisingly small code word sizes, are constructed via look-ahead techniques, based on *state combination* (as opposed to state splitting as in ACH) and guided by an approximate eigenvector with components equal to one or two.

Motivated by this work, the theoretical properties of BDB codes were investigated in [1]. In [1] it is shown that one-symbol look-ahead BDB codes can be advantageously described as well as constructed in terms of *principal state-sets*. We will return to this shortly.

An 8-bits look-ahead rate $2/3$ BDB $(1, 8)$ RLL code entails a partition of the collection of 12-bits code words or *blocks* (12-bit words not violating the prescribed $(1, 8)$ runlength constraint) into 256 mutually disjoint *code word classes*, where each code word class corresponds to a unique 8-bit source word (source symbol) and represents the alternative encodings of this source symbol. Of course, a description of the code in terms of this partition into code word classes alone is not very satisfactory without a means to verify that one-symbol look-ahead encoding is indeed possible. It was shown in [1] that for each BDB code an encoding rule can be described in terms of a collection of so-called *principal state-sets*. (As suggested by the terminology, this generalizes the notion of principal state as introduced by Franaszek [7]. In our case, a principal state-set will be a collection of blocks.) It turns out that correctness of the encoding rules can be established by verifying corrrectness with respect to the given principal state-sets for each code word class *separately*, that is, independent of the possible presence of other code word classes. We explain this in detail in Section 2.

Verifying correctness by hand for 256 code word classes would still be a huge task. Fortunately it is possible to group these classes into 22 distinct *(code word class) types* such that correctness depends on type only. This is further explained in Section 3.

In Section 4 we offer a description of the code in terms of code word class types together with all the information needed to verify correctness of the encoding rule and to count the resulting number of code word classes. The code found is not unique; indeed, we shall also describe a second code with the same parameters.

Finally, in Section 5 we discuss the results.

6.2 Principal state-sets for BDB codes

In this section we outline the description of a one-symbol look-ahead BDB code in terms of principal state-sets, as far as relevant to our case. Further details can be found in [1].

A $(1, 8)$ RLL sequence is a binary sequence in which none of the *forbidden words* 010, 101, 0^{10} or 1^{10} appear ("each run has a length between two and nine"). Let us call such sequences *admissible*, and let a *code word*, or *block*, be any 12-bit admissible sequence.

Define a digraph $\mathcal{D} = (V, A)$ with *states* V and *transitions* A as follows. The states are the code words (as defined above), with a transition from the code word v to the code word w if the 24-bit word vw is again admissible. In that case, we say that the code word w (the code word v) is a *successor* (a *predecessor*) of the code word v (the code word w). Obviously, each *walk* $\cdots v_{n-1} v_n v_{n+1} \cdots$ in $\mathcal{D}$ (v_{n+1} is a successor of v_n for all n) generates an admissible sequence. (Indeed, a forbidden word in a sequence of code words is necessarily confined to two consecutive code words.)

The knowledgeable reader will recognize the digraph $\mathcal{D}$ as a *Moore machine* for $(1, 8)$ RLL sequences. (Albeit of a special type in which all state-labels are distinct. To keep things as simple as possible, we will only consider Moore machines, of this special type, in what follows. For the general case, see [1].) Such a digraph may serve as starting point of the construction of a BDB code. Essentially, a BDB code with source alphabet B for a digraph $\mathcal{D}$ as above is a collection of mutually disjoint state-sets V_b, $b \in B$ (referred to as *code word classes*) together with a finite state finite look-ahead encoder to transform sequences $\{b_n\}_n$, $b_n \in B$, into walks $\{v_n\}_n$ in $\mathcal{D}$ with $v_n \in V_{b_n}$ for all n. Observe that decoding of a BDB code is particularly simple: a state v is decoded as b if $v \in V_b$, irrespective of context. Instead of giving an exact definition of BDB codes, we will use a result

from [1] stating that all one-symbol look-ahead BDB codes possess a special kind of encoder, involving a collection of *principal state-sets*. We will now explain this.

Let $\mathcal{D} = (V, A)$ be a digraph, and let $\mathcal{E}$ be a collection of non-empty subsets of V. Members of $\mathcal{E}$ will be referred to as *principal state-sets*, and we will think of these as representing possible *encoding alternatives* at various times during the encoding process. Let $C \subseteq V$. (We will think of C as representing all possible encodings of some fixed source symbol.)

Definition 1: We will say that C is a *potential code word class with respect to* $\mathcal{E}$ if the following holds:

For each principal state-set $E \in \mathcal{E}$ there exists a state $v \in E$ and another principal state-set $E' \in \mathcal{E}$ such that $E' \subseteq C$ and each state in E' is a successor of v in $\mathcal{D}$.

The importance of this definition can be seen from the following observation. Suppose that we can find a collection V_b, $b \in B$, of mutually disjoint potential code word classes with respect to $\mathcal{E}$. Then this collection constitutes a one-symbol look-ahead BDB code with source alphabet B. Indeed, this can be seen as follows. For each pair (E, b) of a principal state-set $E \in \mathcal{E}$ and a source symbol $b \in B$, let the state $v = \hat{v}(E, b) \in E$ and the principal state-set $E' = \hat{E}(E, b) \in \mathcal{E}$ be defined such that $E' \subseteq V_b$ and E' consists entirely of successors of v in $\mathcal{D}$. (This is possible according to Definition 1.) Now, to encode a sequence $\{b_n\}_{n n \geq 1}$ of source symbols, we inductively construct a sequence $\{E^{(n)}\}_{n n \geq 0}$ of principal state-sets and the required walk $\{v_n\}_{n n \geq 1}$ in $\mathcal{D}$ with $v_n \in V_{b_n}$ by letting $v_{n-1} = \hat{v}(E^{(n-1)}, b_n)$ and $E^{(n)} = \hat{E}(E^{(n-1)}, b_n)$. (Here we let $E^{(0)} \in \mathcal{E}$ be arbitrarily.) Observe that for all $n \geq 1$, v_{n-1} is a predecessor of $v_n \in E^{(n)} \subseteq V_{b_n}$ as required.

Alternatively, the above may be viewed as follows: Consider the digraph $\mathcal{G}$ with as states the principal state-sets, and an arc $E \longrightarrow E'$ with label (b, v) whenever $v = \hat{v}(E, b)$ and $E' = \hat{E}(E, b)$. Then $\mathcal{G}$ constitutes an encoder graph for a one-symbol BDB code with as code word classes the sets V_b, $b \in B$. (There is an encoding delay of one symbol.) In [2] we have shown that, alternatively, such an encoder graph can be obtained from the ACH algorithm. Surprisingly [9], the construction may require the use of an approximate eigenvector with relatively *large* components [3].

In [1] we have shown that in fact all one-symbol look-ahead BDB codes arise out of principal state-sets in the way described here.

The following example, taken from [4], illustrates the above.

Example 1: In this example, we construct a one-symbol look-ahead BDB $(1, \infty)$ RLL code with code word size four and rate $\log_2(6)$, so a code for six

source symbols. The forbidden words for this constrained system are 010 and 101. The digraph $\mathcal{D} = (V, A)$ has as states all admissible 4-bit words (called *code words* throughout this example), with a transition $v \longrightarrow w$ whenever the 8-bit word vw is again admissible. As principal state-sets, we take

(i) all sets consisting of a single code word ending in 00 or 11, and
(ii) all sets consisting of two code words, one ending in 01, the other ending in 10.

It can easily be verified that the following six sets of code words V_i are mutually disjoint potential code word classes with respect to these principal state-sets:

$$V_1 = \{0000\}, \qquad V_5 = \{1000, 0110, 0001\},$$
$$V_2 = \{0011\}, \qquad V_4 = \{1111\},$$
$$V_3 = \{1100\}, \qquad V_6 = \{0111, 1001, 1110\}.$$

Indeed, if for example $C = V_5 = \{1000, 0110, 0001\}$ and if $E = \{..00\}$ is a principal state-set, then $E' = \{0110, 0001\}$ is again a principal state-set and consists of successors in C of the code word $e = ..00$. Or, if $E = \{..01, ..10\}$, then we may take for example $E' = \{1000\}$, $e = ..01$. All other combinations can be treated similarly. The conclusion is that the six code word classes above indeed constitute a one-symbol look-ahead BDB code.

Note that the listed classes are not the only potential code word classes; others are, for example, $V_7 = \{0111, 1000\}$ and $V_8 = \{0110, 0001, 1110, 1001\}$. Consequently, $V_1, \ldots, V_4, V_7, V_8$ constitute a second BDB code for these principal state-sets, with the same parameters. □

An obvious advantage of a description of a one-symbol look-ahead BDB code in terms of both code word classes and principal state-sets is that verification of even a large code is relatively simple: the code word classes should be disjoint, and each code word class *separately* should be a potential code word class with respect to the given collection of principal state-sets. Also, construction of a code from a judiciously chosen collection of principal state-sets (which is the way in which we actually constructed our code) offers an alternative to the construction method by means of "state combination" as described in [4]. The latter method is much easier to implement in a computer program, but unfortunately does not find all one-symbol look-ahead BDB codes (see [1]).

6.3 Begin and end types of code words

In order to simplify the description of our code , we will assign to each code word a *begin type* and an *end type* (where the begin and end type together constitute the

Table 1: *The begin and end types of a code word.*

Begin type	Initial part of code word	End type	Final part of code word
$z1$	01	$z1$	10
$z5$	$001, 0001, 0^4 1, 0^5 1$	$z4$	$100, 1000, 10^4$
$z8$	$0^6 1, 0^7 1, 0^8 1$	$z8$	$10^5, 10^6, 10^7, 10^8$
$z9$	$0^9 1$	$z9$	10^9
$o1$	10	$o1$	01
$o5$	$110, 1110, 1^4 0, 1^5 0$	$o4$	$011, 0111, 01^4$
$o8$	$1^6 0, 1^7 0, 1^8 0$	$o8$	$01^5, 01^6, 01^7, 01^8$
$o9$	$1^9 0$	$o9$	01^9

Table 2: *The number of code words of the various types.*

	$z1$	$z4$	$z8$	$z9$	$o1$	$o4$	$o8$	$o9$
$z1$	16	22	5	1	17	20	6	-
$z5$	24	27	10	-	23	31	6	1
$z8$	3	6	-	-	3	2	2	-
$z9$	1	-	-	-	-	1	-	-
$o1$	17	20	6	-	16	22	5	1
$o5$	23	31	6	1	24	27	10	-
$o8$	3	2	2	-	3	6	-	-
$o9$	-	1	-	-	1	-	-	-

type of a code word), and formulate successor rules, principal state-sets and code classes in terms of types alone. A code word belongs to one of eight begin types, according to the number of initial zeroes (or ones) of the code word, as indicated in Table 1.

Similarly, each code word belongs to one of eight end types, according to the number of final zeroes (or ones) of the code word. The end types are also indicated in Table 1.

The type of a code word shall be indicated as si–sj, where si and sj denote the begin type and end type of the code word, respectively. In Table 2, we have listed the number of code words with a given begin and end type. This is the only data in this correspondence which cannot be verified easily by hand. (Verification by computer is easy).

The code word classes and principal state-sets of our code will be described in terms of begin and end types only. In particular, *we will use only those successor relations which are guaranteed by the types of the code words involved.* For example, a code word with begin type $z5$ is successor to any code word with end type $z4$, so we will say "begin type $z5$ is successor to end type $z4$". On the

Table 3: *Successor relations between end and begin types.*

end type	begin type successors							
$z1$	$z1$	$z5$	$z8$	-	-	-	-	-
$z4$	$z1$	$z5$	-	-	-	$o5$	$o8$	$o9$
$z8$	$z1$	-	-	-	-	$o5$	$o8$	$o9$
$z9$	-	-	-	-	-	$o5$	$o8$	$o9$
$o1$	-	-	-	-	$o1$	$o5$	$o8$	-
$o4$	-	$z5$	$z8$	$z9$	$o1$	$o5$	-	-
$o8$	-	$z5$	$z8$	$z9$	$o1$	-	-	-
$o9$	-	$z5$	$z8$	$z9$	-	-	-	-

other hand, begin type $z5$ is not successor to end type $z8$, although some particular code word of that begin type such as $001\cdots$ is a successor to a code word such as $\cdots 100000$ of that end type. In Table 3 we have listed the various successor relations according to type.

6.4 Description of the code

Our code consists of 256 disjoint code word classes. Each code word class is of a certain *(code word class) type*, where the type of a code word class is the collection of types of its member code words. The principal state-sets of our code will be described in terms of end states only; a set of code words is a principal state-set if and only if the collection of end types of the code words constitute one of the following six sets.

$$F_1 = \{z4\},\ F_2 = \{o4\},\ F_3 = \{z1, o1\},$$

$$F_4 = \{z8, o8\},\ F_5 = \{z8, o9\},\ F_6 = \{z9, o8\}.$$

(Compare this to the description of the principal state-sets in Example 1.) In what follows, when we use the expression "potential code word class", this will always be understood to mean "with respect to the above collection of principal state-sets". As a consequence of the above definition of the principal state-sets, the property of being a potential code word class depends only on *type*. (Recall our restriction on allowable successor relations as agreed in Section 3.) Let us consider a few examples.

Example 2: The set $\{z5–z4\}$ (representing a set that contains a single code word, of type $z5–z4$) is a potential code word class type. Indeed, first we remark that $\{z4\} = F_1$ is end type set of a principal state-set. Secondly, the begin type $z5$

Table 4: *The first block-decodable code.*

Index	Code class type	multiplicity
1	$\{z5\text{–}z4\}$	27
2	$\{o5\text{–}o4\}$	27
3	$\{z5\text{–}o4\}$	31
4	$\{o5\text{–}z4\}$	31
5	$\{z5\text{–}z8, z5\text{–}o9\}$	1
6	$\{o5\text{–}o8, o5\text{–}z9\}$	1
7	$\{z5\text{–}z8, z5\text{–}o8\}$	6
8	$\{o5\text{–}o8, o5\text{–}z8\}$	6
9	$\{z5\text{–}z1, z5\text{–}o1\}$	23
10	$\{o5\text{–}o1, o5\text{–}z1\}$	23
11*	$\{z1\text{–}z4, o1\text{–}o4\}$	22
12*	$\{z1\text{–}o4, o1\text{–}z4\}$	20
13*	$\{z1\text{–}z1, z1\text{–}o1, o1\text{–}z1, o1\text{–}o1\}$	16
14*	$\{z1\text{–}z8, z1\text{–}o8, o1\text{–}z8, o1\text{–}o8\}$	5
15*	$\{z1\text{–}z9, z1\text{–}o8, o1\text{–}z8, o1\text{–}o9\}$	1
16*	$\{o8\text{–}z4, z8\text{–}o4\}$	2
17*	$\{o8\text{–}o4, z8\text{–}z4\}$	6
18*	$\{o8\text{–}z1, o8\text{–}o1, z8\text{–}z1, z8\text{–}o1\}$	3
19	$\{o5\text{–}o8, o8\text{–}z8, z5\text{–}z8\}$	2
20	$\{z5\text{–}z8, z8\text{–}o8, o5\text{–}o8\}$	1
21	$\{o9\text{–}z4, o1\text{–}z1, o5\text{–}o1\}$	1
22	$\{z9\text{–}o4, z1\text{–}o1, z5\text{–}z1\}$	1

is successor to the end types $z1$, $z4$, $o4$, $o8$, and $o9$, and each of the sets F_i, $i = 1, \ldots, 6$, contains (at least) one of these end types. □

In the above example, we say that the list $< \{z5\} >$ forms a *cover list* for $F_1, \ldots, F_6$ since each F_i contains an end type to which $z5$ is a successor. The notion of a cover list is useful when checking potential code word class types. We explain this notion and its use in the next example.

Example 3: The list $< \{z5, z8\}, \{z5, o5\} >$ forms a cover list. To see this, we first note that $z5$ and $z8$ are both successors to the end types $z1$, $o4$, $o8$, and $o9$ (we say that the pair $\{z5, z8\}$ "covers" $F_2, \ldots, F_6$ since each contains one of the above end types). Similarly, $z5$ and $o5$ are both successor to the end types $z4$ and $o4$, and the pair $\{z5, o5\}$ covers F_1 and F_2. So together, the two pairs cover all F_i, thus our list is indeed a cover list.

Table 5: *Covering lists for the code in Table 4.*

Index	Covering lists
1–10	$< \{z5\} >$ and $< \{o5\} >$
11–15	$< \{z1\}, \{o1\} >$
16–18	$< \{z8\}, \{o8\} >$
19–20	$< \{o5, o8\}, \{o5, z5\} >$ and $< \{z5, z8\}, \{z5, o5\} >$
21–22	$< \{o9\}, \{o1, o5\} >$ and $< \{z9\}, \{z1, z5\} >$

Consequently, for example the set

$$\{z5\text{–}z8, z8\text{–}o9, z5\text{–}z1, o5\text{–}o1\}$$

constitutes a potential code word class type, since each of the two sets of begin types that make up the cover list are combined with sets of end types F_i, here $i = 5, 3$. A second example is formed by the set

$$\{z5\text{–}z8, z8\text{–}o9, o5\text{–}o8\},$$

where the same cover list is now combined with the sets F_5 and F_4. □

Our description of the code will consist of a list containing the potential code word class types, together with the number of code word classes of each type (Table 4). To ensure that the code word classes are indeed disjoint, the reader must verify that enough code words of the various types are available, using Table 2. To help in checking the code word class types, we also list in Table 5 a covering list for each of the code word class types. Observe that since the collection of principal state-sets is invariant under $0 \leftrightarrow 1$, the successor relations, and hence the collection of potential code word classes, are invariant under $z \leftrightarrow o$. In Table 4 symmetric types are marked with an asterisk. The code uses all available code words except for one of each of the three types $z9$–$z1$, $o9$–$o1$, and $z8$–$o8$. Therefore, we may, at the cost of one additional code word class type, obtain a fully symmetric code by adding a code word with type $z8$–$o8$ to one of the two code word classes of the type with index 19 in Table 4.

This code is not the only BDB code with these parameters. A second code is given in Table 6, with covering lists as in Table 7. (The first 10 code class types are identical to those of the first code, with the same multiplicities.)

6.5 Discussion

We have described two new rate $8/12$ $(1, 8)$ RLL block-decodable codes with block size twelve, well adapted to byte-oriented storage systems since error propagation is limited to one byte. The codes allows a relatively simple description

Table 6: *A second block-decodable code.*

Ind.	Code class type	mult.	Ind.	Code class type	mult.
1	$\{z5–z4\}$	27	16	$\{o1–z4, z1–z4\}$	20
2	$\{o5–o4\}$	27	17	$\{z1–o4, o1–o4\}$	20
3	$\{z5–o4\}$	31	18*	$\{o1–o4, z1–z4\}$	1
4	$\{o5–z4\}$	31	19	$\{z9–z1, o5–o1, o8–z1\}$	1
5	$\{z5–z8, z5–o9\}$	1	20	$\{o9–o1, z5–z1, z8–o1\}$	1
6	$\{o5–o8, o5–z9\}$	1	21	$\{o1–z8, o5–o8, o8–z8\}$	2
7	$\{z5–z8, z5–o8\}$	6	22	$\{z1–o8, z5–z8, z8–o8\}$	2
8	$\{o5–o8, o5–z8\}$	6	23	$\{z1–z9, o5–o8, z8–z1, z8–o1\}$	1
9	$\{z5–z1, z5–o1\}$	23	24	$\{o1–o9, z5–z8, o8–o1, o8–z1\}$	1
10	$\{o5–o1, o5–z1\}$	23	25	$\{z1–z1, o8–o1, z8–z1, z8–o1\}$	1
11	$\{z9–o4, z1–z4\}$	1	26	$\{o1–o1, z8–z1, o8–o1, o8–z1\}$	1
12	$\{o9–z4, o1–o4\}$	1	27*	$\{z1–z8, z1–o8, o1–z8, o1–o8\}$	4
13	$\{o8–z4, z8–z4\}$	2	28*	$\{z1–z1, z1–o1, o1–z1, o1–o1\}$	15
14	$\{z8–o4, o8–o4\}$	2			
15*	$\{o8–o4, z8–z4\}$	4			

Table 7: *Covering lists for the code in Table 6.*

Index	Covering lists
1–10	$<\{z5\}>$ and $<\{o5\}>$
11–12	$<\{z1\},\{z9\}>$ and $<\{o1\},\{o9\}>$
13–15	$<\{z8\},\{o8\}>$
16–18	$<\{z1\},\{o1\}>$
19–20	$<\{z9,o5\},\{o5,o8\}>$ and $<\{o9,z5\},\{z5,z8\}>$
21–22	$<\{o1,o5\},\{o5,o8\}>$ and $<\{z1,z5\},\{z5,z8\}>$
23–24	$<\{z8\},\{z1,o5\}>$ and $<\{o8\},\{o1,z5\}>$
25–26	$<\{z8\},\{z1,o8\}>$ and $<\{o8\},\{o1,z8\}>$
27–28	$<\{z1\},\{o1\}>$

and can be verified to satisfy the claimed requirements by hand. (Appearances notwithstanding, the first of these codes was in fact *constructed by hand.*)

Acknowledgement

The author wishes to thank Kees Schouhamer Immink for many inspiring discussions.

References

[1] H.D.L. Hollmann, "Bounded-delay encodable, block-decodable codes for constrained systems", *IEEE Trans. Inform. Theory*, vol. 42, no. 6, Nov. 1996.

[2] H.D.L. Hollmann, "On the construction of bounded-delay encodable codes for constrained systems", *IEEE Trans. Inform. Theory*, vol. 41, no. 5, pp. 1354–1378, Sept. 1995.

[3] H.D.L. Hollmann, "On an approximate eigenvector associated with a modulation code", to appear in *IEEE Trans. Inform. Theory*.

[4] K.A.S. Immink, "Block-decodable runlength-limited codes via look-ahead technique", *Philips J. Res.*, vol. 46, no. 6, pp. 293–310, 1992.

[5] K.A. Schouhamer Immink, *Coding techniques for digital recorders*, Englewood Cliffs, NY: Prentice-Hall, 1991.

[6] R.L. Adler, D. Coppersmith, and M. Hassner, "Algorithms for sliding block codes. An application of symbolic dynamics to information theory", *IEEE Trans. Inform. Theory*, vol. IT-29, no. 1, pp. 5–22, Jan. 1983.

[7] P.A. Franaszek, "Construction of bounded delay codes for discrete noiseless channels", *IBM J. Res. Develop.*, vol. 26, pp. 506–514, July 1982.

[8] A. Lempel and M. Cohn, "Lookahead coding for input-restricted channels", *IEEE Trans. Inform. Theory*, vol. IT-28, pp. 933–937, Nov. 1982.

[9] B. Marcus and R. Roth, "Bounds on the number of states in encoder graphs for input-constrained channels", *IEEE Trans. Inform. Theory*, vol. IT-37, no. 3, pp. 742–758, May 1991.

Chapter 7

On an approximate eigenvector associated with a modulation code

***Abstract*— Let $\mathcal{S}$ be a constrained system of finite type, described in terms of a labeled graph $\mathcal{M}$ of finite type. Furthermore, let $\mathcal{C}$ be an irreducible constrained system of finite type, consisting of the collection of possible code sequences of some finite-state encodable, sliding-block decodable modulation code for $\mathcal{S}$. It is known that this code could then be obtained by state-splitting, using a suitable approximate eigenvector. In this paper, we show that the collection of all approximate eigenvectors that could be used in such a construction of $\mathcal{C}$ contains a unique minimal element. Moreover, we show how to construct its linear span from knowledge of $\mathcal{M}$ and $\mathcal{C}$ only, thus providing a lower bound on the components of such vectors.**

For illustration we discuss an example showing that sometimes arbitrary large approximate eigenvectors are required to obtain the best code (in terms of decoding-window size) although a small vector is also available.

***Index Terms*— Constrained systems, state-splitting, sliding-block decodable, approximate eigenvector.**

7.1 Introduction

Digital data storage systems commonly employ a modulation code to provide the data as written on the storage medium with certain desirable properties. A well-known example is the use of (d, k) codes in magnetic storage systems to combat intersymbol interference and loss of clock. Typically, the collection of allowable

© 1996, IEEE. To appear in IEEE Transactions on Information Theory. Reprinted with permission.

sequences (called the constrained system) can be described by means of a labeled directed graph as the collection of all label sequences of paths in this graph.

In this paper, a modulation code for a constrained system consists of a finite-state encoder which maps arbitrary sequences of source symbols into allowable sequences, and a decoder capable of retrieving the original source sequence from the code sequence. The *code system* of a modulation code is the collection of all possible code sequences of the code.

Since modulation codes are often used in a noisy environment, code sequences get corrupted by noise, so will contain errors. It is therefore necessary to consider the problem of error propagation. To limit error propagation, most practical codes are sliding-block decodable. Here, the decoder needs to base its decision on a source symbol only on the present code symbol together with a at most m previous code symbols and at most a upcoming code symbols, for some numbers m and a. Evidently, one error in the code sequence thus propagates into at most $m+1+a$ errors in the source sequence. The number $m+1+a$ is called the decoding window size of the code. This number is an important parameter of a code, related to the maximum error propagation and also to the amount of hardware needed to implement the decoder, and should therefore be as small as possible.

Many code construction methods involve manipulation of the graph that describes the constraint, under guidance of an approximate eigenvector, a collection of non-negative integer weights associated with the vertices (states) of the graph with the property that N times the weight of a state is at most equal to the sum of the weights of the successors of this state. Here N is a number related to the desired rate of the code. A typical example of such methods is the state-splitting algorithm as described in [1].

Recently [2], [3, 4], this method has been shown to be universal, that is, essentially, each sliding-block decodable code for a constraint of finite type with a code system of finite type can be obtained by this method, with a proper choice for the approximate eigenvector. (In fact, the results in [2] apply to all constrained systems, not just those of finite type; in more general situations some additional operations are required.) The extend of this result is discussed in Chapter V, Appendix B, where some claims in [4] are further specified.

Generally speaking, distinct approximate eigenvectors lead to different codes. The usual approach is to choose such a vector with small components, since this leads to the smallest currently known upper bound on the decoding window size. However, it is known [4] that such a choice does not always leads to the best code. We will present a sequence of examples where the smallest available vector has maximum component equal to two while the vector that leads to the best (block-decodable) code necessarily has maximum component at least N, for each choice of N. (The smallest vector leads to a decoding window of two codewords.)

This paper has been written in an effort to clarify the intimate relation between approximate eigenvectors on the one hand, and code systems of modulation codes on the other hand. Our approach is consistent with the Willems' approach [9] to investigate properties of a system based only on knowledge about the sequences that make up the system.

We will show that each irreducible finite-type code system of a sliding-block decodable code that is contained in a given constrained system of finite type and is obtained by the state-splitting algorithm determines a unique approximate eigenvector, with the properties that (i) the code system can be constructed with this vector, and (ii) each other vector that can be used to construct this code system is at least as large (component-wise) as this vector. Unfortunately, in general we know of no method to find this vector. However, we do present some methods to obtain its *linear span*, thus providing a lower bound on approximate eigenvectors that can construct the given code.

7.2 Notation and background

We begin by establishing our notation. For further details, we refer to [3].

A constrained system is typically described in terms of a labeled directed graph, which in this context is often referred to as a Finite-State Transition Diagram or FSTD. Here, a FSTD $\mathcal{M}$ consists of a finite collection V of states and a finite collection A of labeled transitions. Each transition α of $\mathcal{M}$ has a label $L(\alpha)$, an initial state $\mathrm{beg}(\alpha)$ and a terminal state $\mathrm{end}(\alpha)$, and is referred to as a transition from $\mathrm{beg}(\alpha)$ to $\mathrm{end}(\alpha)$ with label $L(\alpha)$. We denote by A_v the collection of transitions from state v, that is, all transitions α with $\mathrm{beg}(\alpha) = v$.

The constrained system S presented by $\mathcal{M}$ is the collection of all label sequences $L(\alpha_1)\cdots L(\alpha_n)$ of paths $\alpha_1\cdots\alpha_n$ in $\mathcal{M}$. (Here and in what follows, when we say "path" we always refer to a *directed* path.)

We say that the FSTD $\mathcal{M}$ is *of finite type* if there are numbers m and a such that, for each path $\alpha_{-m}\cdots\alpha_0\cdots\alpha_a$, transition α_0 is uniquely determined by the label sequence $L(\alpha_{-m})\cdots L(\alpha_0)\cdots L(\alpha_a)$ of the path. A constrained system is said to be of finite type if it can be presented by a FSTD of finite type. (As is well-known, this is the case precisely when the system can be described in terms of a finite collection of "forbidden subwords", see e.g. [1].) $\mathcal{M}$ is said to be *deterministic* if in each state the transitions leaving that state carry distinct labels.

A (rate-N) *encoder* is a FSTD $\mathcal{M}$ in which a constant number of N transitions leave each state, and that is *lossless*, i.e., any two distinct paths in $\mathcal{M}$ with the same initial state and terminal state generate distinct label sequences. In this paper, when we speak of a code, we always mean a code that has an encoder of this type.

Mostly, we even require that the encoder is *sliding-block decodable*, i.e., that $\mathcal{M}$ is of finite type.

A FSTD $\mathcal{M}$ is *irreducible* if for each pair of states (v, w) there is a path in $\mathcal{M}$ from v to w. Also, a constrained system $\mathcal{S}$ is called irreducible if for each two sequences x and y contained in $\mathcal{S}$ and each two indices n and m there exists a sequence z in $\mathcal{S}$ and an integer $k > 0$ such that $z_i = x_i$ for all $i \leq n$ and $z_{n+k-m+i} = y_i$ for all $i \geq m$. In other words, there exists a word $w_1 \cdots w_{k-1}$ such that the sequence

$$\cdots x_{n-1} x_n w_1 \cdots w_{k-1} y_m y_{m+1} \cdots$$

is again contained in $\mathcal{S}$. Note that a constrained system $\mathcal{S}$ presented by an irreducible FSTD $\mathcal{M}$ is irreducible. Conversely, an irreducible constrained system $\mathcal{S}$ presented by an FSTD $\mathcal{M}$ is also presented by some irreducible component of $\mathcal{M}$ ([6]). A component of $\mathcal{M}$ is called a *sink* if no transitions leave this component. Each FSTD $\mathcal{M}$ contains at least one irreducible sink.

With a FSTD $\mathcal{M}$ we associate its *adjacency matrix* $D_{\mathcal{M}}$, a $|V| \times |V|$ matrix for which $D_{\mathcal{M}}(v, w)$ counts the number of transitions from state v to state w. An $[\mathcal{M}, N]$*-approximate eigenvector* ϕ consists of a collection of non-negative integer *weights* ϕ_v, not all zero, associated with the states v of $\mathcal{M}$, with the property that N times the weight of a state v is at most equal to the sum of the weights of terminal states of the transitions from state v, that is,

$$N\phi_v \leq \sum_{\alpha \in A_v} \phi_{\mathrm{end}(\alpha)} \tag{1}$$

holds for each state v. The above condition is usually expressed as

$$N\phi \leq D_{\mathcal{M}}\phi, \tag{2}$$

where the inequality is to be interpreted component-wise.

Next, we will give a very brief outline of state splitting. Further details can be found, e.g., in [7] or [3]. A *round of state-splitting* is a method to transform a given FSTD $\mathcal{M}$ into a new FSTD $\mathcal{M}'$, in the following way. For each state v of $\mathcal{M}$, we partition the set A_v of transitions leaving v into sets A_v^i, $i = 1, \ldots, n_v$. The FSTD $\mathcal{M}'$ is then constructed as follows. Each state v of $\mathcal{M}$ produces states v^i, $i = 1, \ldots, n_v$, in $\mathcal{M}'$, and each transition α, from state v to state w, and contained in A_v^i, say, produces transitions

$$\alpha^j : v^i \mapsto w^j,$$

$j = 1, \ldots, n_w$, in $\mathcal{M}'$, all of which inherit their codeword label from α.

So each state v splits into a number of states v^i, where state v^i represents the part of v that can be left by transitions α in A_v^i. Moreover, each transition α splits into transitions α^j, where α^j represents the part of α that leads to the j-th part of the terminal state of α. We will refer to states v^i and transitions α^j as *children* of state v and transition α in $\mathcal{M}'$, and to v and α as their *parents* in $\mathcal{M}$, respectively. We extend this notion of children and parents to paths in $\mathcal{M}$ and $\mathcal{M}'$ in the obvious way.

From the above description it is immediate that all children $\alpha'\beta'$ in $\mathcal{M}'$ of a given path $\alpha\beta$ in $\mathcal{M}$ agree in transition α'. ("A transition in $\mathcal{M}$ with a given *future* determines a *unique* transition in $\mathcal{M}'$.") Thus, the parent/child relation in fact provides a one-to-one correspondence between bi-infinite paths in $\mathcal{M}$ and $\mathcal{M}'$ and, in particular, $\mathcal{M}$ and $\mathcal{M}'$ generate the same constrained sequences.

We say that an FSTD $\mathcal{M}$ is transformed into the FSTD $\mathcal{M}^*$ by *k rounds of state-splitting* if $\mathcal{M}^*$ is the result of a sequence of transformations

$$\mathcal{M} = \mathcal{M}^{(0)} \mapsto \mathcal{M}^{(1)} \cdots \mapsto \mathcal{M}^{(k)} = \mathcal{M}^*,$$

where each intermediate transformation consists of a round of state-splitting.

The parent/child relation extends in the obvious way to states, transitions, and paths in $\mathcal{M}$ and $\mathcal{M}^*$. Moreover, it is easily seen that each state (transition) in $\mathcal{M}^*$ has a unique parent state (transition) in $\mathcal{M}$, and that all children $\alpha_0^* \cdots \alpha_k^*$ in $\mathcal{M}^*$ of a given path $\alpha_0 \cdots \alpha_k$ in $\mathcal{M}$ agree in the first transition α_0^*.

Now we come to the notion of *ae-consistent state-splitting*. This method in fact transforms a *pair* $(\mathcal{M}, \phi)$ consisting of an FSTD $\mathcal{M}$ and an $[\mathcal{M}, N]$-approximate eigenvector ϕ. We say that a round of state-splitting that transforms $\mathcal{M}$ into $\mathcal{M}'$ as explained earlier is *ae-consistent* (with respect to the approximate eigenvector ϕ) if, for each state v of $\mathcal{M}$, the weight ϕ_v associated with v can be distributed over its children v^i in $\mathcal{M}'$ in such a way that the resulting assignment ϕ' of weights to the states of $\mathcal{M}'$ now constitutes an $[\mathcal{M}', N]$-approximate eigenvector. Similarly, we say that the pair $(\mathcal{M}, \phi)$ is transformed into the pair $(\mathcal{M}^*, \phi^*)$ by *k rounds of ae-consistent state-splitting* if the pair $(\mathcal{M}^*, \phi^*)$ is the result of k rounds of ae-consistent state-splitting. In this situation, we say that $\mathcal{M}^*$ is obtained from $\mathcal{M}$ by *ae-consistent state-splitting* (with respect to the approximate eigenvector ϕ).

The importance of the notion of ae-consistent state-splitting follows from the following observation: if the weights of all the states of an irreducible sink of $\mathcal{M}^*$ are equal, then by the definition of an approximate eigenvector the out-degree in each of these states is at least N, so that from $\mathcal{M}^*$ we can obtain an encoder for the constrained system S presented by $\mathcal{M}$. In [1] the authors describe an algorithm, the ACH state-splitting algorithm, which demonstrates that for *each* approximate

eigenvector there exists a sequence of ae-consistent state-splittings leading in the above way to an encoder.

It is almost, but not quite true that each finite-state encodable, sliding-block decodable code for a finite-type constraint can be obtained in this way (one further operation called "state-expansion", duplication of states, is required in the case where the encoder graph is not of finite type, see [2]). It is true, though, that, given a sliding-block decodable finite-state encodable code, a code can be obtained by state-splitting that has the same code system as the given code and that can be decoded by the same sliding-block decoder. We refer to [2], [3, 4], and Chapter 5 for further details.

7.3 Preliminaries

We begin with a simple but important observation. Suppose that the FSTD $\mathcal{N}$ is obtained from $\mathcal{M}$ by state-splitting, followed by the deletion of certain states and transitions, and suppose furthermore that each state of $\mathcal{N}$ has out-degree at least N. For each state v of $\mathcal{M}$, let γ_v denote the number of its children in $\mathcal{N}$.

Proposition 1: The vector γ is an $[\mathcal{M}, N]$-approximate eigenvector, and the state-splitting steps are ae-consistent with respect to γ.

Proof: In each intermediate FSTD $\mathcal{M}^{(i)}$, we can take as the weight of a state its number of children in $\mathcal{N}$. That this is indeed an approximate eigenvector for $\mathcal{M}^{(i)}$ can be shown directly by induction on $i = k, k-1, \ldots, 0$, but follows also from Theorem 4 below applied to $\mathcal{M}^{(i)}$ instead of $\mathcal{M}$. Further details are left to the reader. □

In the remainder of this paper, we will make the following assumptions. We are given a constrained system S, presented by an irreducible FSTD $\mathcal{M}$ of finite type with memory m and anticipation a, and an $[\mathcal{M}, N]$-approximate eigenvector ϕ. The FSTD $\mathcal{M}^*$ is obtained from the pair $(\mathcal{M}, \phi)$ by $k-1$ rounds of ae-consistent state splitting with respect to ϕ. The FSTD $\mathcal{N}$ is obtained from $\mathcal{M}^*$ by deleting certain vertices and transitions. We assume that $\mathcal{N}$ is irreducible, and that exactly N transitions leave each state of $\mathcal{N}$ (so $\mathcal{N}$ is an *encoder graph*). Moreover, we assume that all vertices of $\mathcal{N}$ carry the *same* non-zero weight in $\mathcal{M}^*$. Note that this includes the case where all vertices carry the same non-zero weigth in $\mathcal{M}$; in that case, our assumptions comply with the scenario when $\mathcal{N}$ is obtained from $\mathcal{M}$ and ϕ by ae-consistent state-splitting. We will refer to the constrained system $\mathcal{C} \subseteq S$ generated by $\mathcal{N}$ as the *code system*. Note that the encoder $\mathcal{N}$ will again be of finite type, and hence the code system $\mathcal{C}$ generated by $\mathcal{N}$ is the code system

of a sliding-block decodable code. The adjacency matrices of $\mathcal{M}$ and $\mathcal{N}$ will be denoted by $D_{\mathcal{M}}$ and $D_{\mathcal{N}}$, respectively.

A state v in $\mathcal{M}$ will have n_v children v^i, $1 \le i \le n_v$, in $\mathcal{M}^*$, of which the states v^i, $i \in I_v$, are contained in $\mathcal{N}$. The number $|I_v|$ of children of state v in $\mathcal{N}$ will be denoted by γ_v. (Note that $\gamma_v \le \phi_v$.) Similarly, a transition α in $\mathcal{M}$ will have n_α children α^j, $1 \le j \le n_\alpha$, in $\mathcal{M}^*$.

The collection of paths in $\mathcal{M}$ of length n starting in state v or starting with transition α, and having a child in $\mathcal{N}$, will be denoted by $\tilde{\Sigma}_v^{(n)}$ or $\tilde{\Sigma}_\alpha^{(n)}$, respectively. Moreover, for $n \ge k$ we let $\tilde{\Sigma}_{v^i}^{(n)}$ and $\tilde{\Sigma}_{\alpha^j}^{(n)}$ denote the subsets of $\tilde{\Sigma}_v^{(n)}$ and $\tilde{\Sigma}_\alpha^{(n)}$ consisting of those paths whose child (or children) in $\mathcal{N}$ start in state v^i or start with transition α^j, respectively. Note that the first k transitions of a path in $\mathcal{M}$ uniquely determine the first transition of any of its children in $\mathcal{M}^*$, that is, for $n \ge k$ the sets $\tilde{\Sigma}_{v^i}^{(n)}$, $1 \le i \le n_v$, partition $\tilde{\Sigma}_v^{(n)}$ and the sets $\tilde{\Sigma}_{\alpha^j}^{(n)}$, $1 \le j \le n_\alpha$, partition $\tilde{\Sigma}_\alpha^{(n)}$. This observation will be used repeatedly in what follows.

Our aim will be to find a lower bound on ϕ using only knowledge about $\mathcal{M}$ and $\mathcal{C}$. Note that $\mathcal{M}$ and $\mathcal{C}$ together determine the number N.

7.4 The number of children in $\mathcal{N}$ of paths in $\mathcal{M}$

Our aim in this section will be to derive an expression for the number $|\tilde{\Sigma}_v^{(n)}|$ of paths of length n in $\mathcal{M}$ starting in state v that have a child in $\mathcal{N}$, and to study its behavior if $n \to \infty$. In order to do so, we will need the following result on the asymptotic behavior of powers of the adjacency matrix $D_{\mathcal{N}}$ of $\mathcal{N}$.

Proposition 2: We have that

$$\lim_{n\to\infty} N^{-n} D_{\mathcal{N}}^n = \mathbf{1} \cdot \boldsymbol{b}^\top,$$

where $\mathbf{1}$ denotes the all-one vector and $\boldsymbol{b}$ denotes the (unique) positive left-eigenvector of $D_{\mathcal{N}}$ with corresponding eigenvalue N for which $\boldsymbol{b}^\top \mathbf{1} = 1$.

Proof: Our assumptions on $\mathcal{N}$ imply that $D_{\mathcal{N}}$ is irreducible, non-negative, and has right-eigenvector $\mathbf{1}$ with eigenvalue N. Now the existence of $\boldsymbol{b}$ and the asymptotic result in the theorem follow from standard Perron-Frobenius theory, see e.g. [8]. □

Next, for each state v^* of $\mathcal{N}$ and each integer $n \ge k$, we define

$$p_{v^*}^{(n)} = |\tilde{\Sigma}_{v^*}^{(n)}|, \tag{3}$$

the number of paths of length n in $\mathcal{M}$ that have a child in $\mathcal{N}$ starting in state v^*. We observed earlier that for $n \geq k$, the sets $\tilde{\Sigma}_{v^i}^{(n)}$, $i \in I_v$, partition $\tilde{\Sigma}_v^{(n)}$, hence

$$|\tilde{\Sigma}_v^{(n)}| = \sum_{i \in I_v} p_{v^i}^{(n)}. \tag{4}$$

Let $p^{(n)}$ denote the vector with components $p_{v^*}^{(n)}$.

Proposition 3: The vectors $p^{(n)}$, $n \geq k$, satisfy

$$p^{(n+1)} = D_{\mathcal{N}} p^{(n)}.$$

Proof: For $n \geq k$, we have that

$$\begin{aligned} p_{v^*}^{(n+1)} &= |\tilde{\Sigma}_{v^*}^{(n+1)}| \\ &= \sum_{\alpha^*} |\tilde{\Sigma}_{\alpha^*}^{(n+1)}| \\ &= \sum_{\alpha^*} |\tilde{\Sigma}_{\text{end}(\alpha^*)}^{(n)}|, \end{aligned}$$

where the sums are over all α^* for which $\text{beg}(\alpha^*) = v^*$. □

Now, for each state v of $\mathcal{M}$, we define

$$\tilde{\theta}_v^{(n)} = |\tilde{\Sigma}_v^{(n)}|/N^n, \tag{5}$$

and we let $\tilde{\theta}^{(n)}$ denote the vector with components the $\tilde{\theta}_v^{(n)}$. We will use the above results to derive the following.

Theorem 4: (i) The vectors $\tilde{\theta}^{(n)}$, $n = 1, 2, \ldots$, satisfy

$$N\tilde{\theta}^{(n)} \leq D_{\mathcal{M}} \tilde{\theta}^{(n-1)}.$$

(ii) For each state v, the limit

$$\tilde{\theta}_v = \lim_{n \to \infty} \tilde{\theta}_v^{(n)}$$

exists, and there is a constant $c > 0$ such that $\tilde{\theta} = c\gamma$ holds, where $\tilde{\theta}$ is the vector with components $\tilde{\theta}_v$.

Proof: (i) To prove (i), it is sufficient to note that

$$\begin{aligned} |\tilde{\Sigma}_v^{(n)}| &= \sum_{\alpha} |\tilde{\Sigma}_{\alpha}^{(n)}| \\ &\leq \sum_{\alpha} |\tilde{\Sigma}_{\text{end}(\alpha)}^{(n-1)}|, \end{aligned}$$

where the sums are over all transitions α of $\mathcal{M}$ with $\mathrm{beg}(\alpha) = v$. The inequality reflects the observation that there could be paths $\alpha_1\alpha_2\cdots\alpha_n$ from v that are not in $\tilde{\Sigma}_v^{(n)}$ but for which $\alpha_2\cdots\alpha_n$ has a child in $\mathcal{N}$; the set $\tilde{\Sigma}_v^{(n)}$ can even be empty.

(ii) From Proposition 3, we may conclude that

$$p^{(n)} = D_{\mathcal{N}}^{n-k} p^{(k)},$$

for $n \geq k$. Hence by Proposition 2, we have that

$$\lim_{n\to\infty} N^{-n} p^{(n)} = N^{-k}(\boldsymbol{b}^\top \cdot p^{(k)})\boldsymbol{1}.$$

Put

$$c = N^{-k}(\boldsymbol{b}^\top \cdot p^{(k)}).$$

Note that $c > 0$. From (4), we now conclude that

$$\begin{aligned} \lim_{n\to\infty} \tilde{\theta}_v^{(n)} &= c\sum_{i\in I_v} 1 \\ &= c\gamma_v. \end{aligned}$$

□

Let θ be the *smallest* non-negative non-zero integral vector in the linear span of the vector $\tilde{\theta}$, or, equivalently, in the linear span of γ. Then the above theorem has the following consequence.

Theorem 5: Both γ and θ are $[\mathcal{M}, N]$-approximate eigenvectors, and

$$\phi \geq \gamma \geq \theta.$$

Proof: Since both γ and θ are integral and contained in the linear span of $\tilde{\theta}$, the fact that both vectors are approximate eigenvectors follows from Theorem 4. From the definition of θ, the inequality $\gamma \geq \theta$ is obvious. Finally, the number of children γ_v of a state v of $\mathcal{M}$ in $\mathcal{N}$ is certainly not greater than the number of children of v in $\mathcal{M}^*$, which in turn is at most equal to ϕ_v if $\phi_v > 0$; if $\phi_v = 0$, then $\gamma_v = 0$ by our assumption on $\mathcal{N}$. □

7.5 How $\mathcal{M}$ and $\mathcal{C}$ determine $\tilde{\theta}$

In this section we will derive an expression for $|\tilde{\Sigma}_v^{(n)}|$ in terms of $\mathcal{M}$ and the codeword sequences $\mathcal{C}$ generated by $\mathcal{N}$ only. In view of Theorem 5, we thus obtain a lower bound for ϕ in terms of $\mathcal{M}$ and $\mathcal{C}$ alone. The result that we need is given by our next theorem.

Theorem 6: The following statements are equivalent.
(i) The path $\alpha_0\alpha_1\cdots\alpha_{n-1}$ is contained in $\tilde{\Sigma}_v^{(n)}$.
(ii) There exists a word $x = x_{-m}\cdots x_0\cdots x_{n+a-1}$ in $\mathcal{C}$ that is generated in $\mathcal{M}$ by a path $\omega = \omega_{-m}\cdots\omega_{n+a-1}$ for which $\omega_i = \alpha_i, 0 \le i \le n-1$.

Proof: First, let $\alpha_0\alpha_1\cdots\alpha_{n-1} \in \tilde{\Sigma}_v^{(n)}$, with child $\alpha_0^*\cdots\alpha_{n-1}^*$ in $\mathcal{N}$, say. Since $\mathcal{N}$ is irreducible, this path can be extended to a path $\alpha_{-m}^*\cdots\alpha_{n+a-1}^*$ in $\mathcal{N}$, with parent $\alpha_{-m}\cdots\alpha_0\cdots\alpha_{n-1}\cdots\alpha_{n+a-1}$ in $\mathcal{M}$. So if x is the word generated by these paths, then x has the property in (ii).

Now suppose that x is a word in $\mathcal{C}$ as in (ii). Since x is in $\mathcal{C}$, there is a path $\beta^* = \beta_{-m}^*\cdots\beta_{n+a-1}^*$ in $\mathcal{N}$ that generates x. Let $\beta = \beta_{-m}\cdots\beta_{n+a-1}$ denote the parent in $\mathcal{M}$ of this path. By our assumption that $\mathcal{M}$ has memory m and anticipation a, the paths β and ω agree in positions $0,\ldots,n-1$, that is, $\beta_i = \omega_i = \alpha_i$, $0 \le i \le n-1$, hence $\alpha_0\alpha_1\cdots\alpha_{n-1}$ has child $\beta_0^*\cdots\beta_{n-1}^*$ in $\mathcal{N}$ and is contained in $\tilde{\Sigma}_v^{(n)}$. □

The above theorem thus shows that $\tilde{\Sigma}_v^{(n)}$, and hence $\tilde{\theta}$, is determined by $\mathcal{M}$ and $\mathcal{C}$ only, and therefore does not depend on the particular encoder $\mathcal{N}$.

7.6 Main results

We can now summarize the main results in this paper as follows.

Theorem 7: Let $\mathcal{M}$ be an FSTD of finite type, and let ϕ be an $[\mathcal{M}, N]$-approximate eigenvector. Let $\mathcal{C}$ be an irreducible code system generated by some irreducible rate-N encoder FSTD $\mathcal{N}$ obtained from $\mathcal{M}$ and ϕ by ae-consistent state splitting, and let $\tilde{\theta}$ be determined by $\mathcal{M}$ and $\mathcal{C}$ as in Theorem 6. Let γ_v be the number of children in $\mathcal{N}$ of state v in $\mathcal{M}$. Then the following holds.
(i) The encoder $\mathcal{N}$ can also be obtained by ae-consistent state-splitting from $\mathcal{M}$ and γ.
(ii) The inequality $\phi_v \ge \gamma_v$ holds for each state v of $\mathcal{M}$.
(iii) The vectors $\tilde{\theta}$ and γ are linearly dependent.
(iv) If θ denotes the smallest non-zero non-negative integral vector in the linear span of $\tilde{\theta}$ (or, equivalently, in the linear span of γ), then $\phi_v \ge \theta_v$ holds for each state v of $\mathcal{M}$.

This theorem has an important consequence.

Theorem 8: Let $\mathcal{S}$ be a constrained system of finite type, presented by an FSTD $\mathcal{M}$ of finite type, and let $\mathcal{C}$ be an irreducible code system of some encoder obtained from an $[\mathcal{M}, N]$-approximate eigenvector by ae-consistent state-splitting. (Here the value of N is determined by $\mathcal{M}$ and $\mathcal{C}$.) Then there exists a

(unique) vector $\gamma^* = \gamma^*(\mathcal{M}, \mathcal{C})$ with the following properties.
(i) The vector γ^* is an $[\mathcal{M}, N]$-approximate eigenvector, and has the further property that $\mathcal{C}$ is the code system of a rate-N encoder that can be constructed by ae-consistent state-splitting from $\mathcal{M}$ and γ^*.
(ii) Any other $[\mathcal{M}, N]$-approximate eigenvector ϕ with this property satisfies $\phi \geq \gamma^*$, that is, $\phi_v \geq \gamma_v^*$ holds for each state v of $\mathcal{M}$.

Proof: Since the vector $\tilde{\theta}$ as defined in Theorem 7 only depends on $\mathcal{M}$ and $\mathcal{C}$, but not on the particular encoder $\mathcal{N}$, it follows from part (iii) of Theorem 7 that *all* the vectors γ that can be obtained from encoders as described in that theorem are scalar multiples of the *same* vector $\tilde{\theta}$. Let $\gamma^* = \gamma^*(\mathcal{M}, \mathcal{C})$ be the *smallest* of all such vectors γ. (This minimum exists since all γ have integer components.) Our two claims now follow from parts (i) and (ii) of Theorem 7. □

The above justifies to call this vector γ^* *the* approximate eigenvector of the code system $\mathcal{C}$. Unfortunately, the definition of γ^* is highly non-constructive. However, we can determine the *linear span* of γ^*, for example from an encoder $\mathcal{N}$ for $\mathcal{C}$ together with the state-splitting procedure used to obtain it and inspection of the corresponding vector γ. But the definition of γ^* gives no further clues on how γ^* itself could be found. It would therefore be interesting to obtain a *constructive* description of γ^*, purely in terms of the sequences contained in $\mathcal{C}$. Perhaps the methods in [2] can be of use to obtain such a result. Note also that the lower bounds on the number of encoder states derived in [6] deliver also a lower bound on $\sum_v \gamma_v^*$, but in general the number of states in $\mathcal{N}$ can be reduced by state merging and therefore such a bound need not be tight.

7.7 Some remarks

1. Provided that the constraint is of finite type, the approximate eigenvector θ can also be obtained in a different way. Let us define $\bar{\Sigma}_v^{(n)}$ to denote the number of paths of length n in $\mathcal{M}$ that start in state v and generate a word in the code system $\mathcal{C}$, and let

$$\bar{\theta}_v^{(n)} = |\bar{\Sigma}_v^{(n)}|/N^n.$$

If $\bar{\theta}^{(n)}$ is the vector with components the numbers $\bar{\theta}_v^{(n)}$, then we can show, by the same method as used in Section 7.4, that the limit

$$\bar{\theta} = \lim_{n \to \infty} \bar{\theta}^{(n)}$$

exists and that $\bar{\theta}$ is again a scalar multiple of the vector γ. We leave the verification of these claims to the reader.

2. Severall possible *converses* of Theorem 7 are certainly false. Firstly, it may happen that a code system cannot be obtained by ae-consistent state splitting from an approximate eigenvector ϕ although $\phi \geq \gamma = \theta$ holds (e.g., Example 3). Secondly, it is also possible that C cannot be obtained by ae-consistent state-splitting from θ (e.g., Example 2). And thirdly, there are examples of codes that can be obtained by ae-consistent state splitting from *distinct* approximate eigenvectors (e.g., Example 2). Let us say that an approximate eigenvector ϕ is *tight* if none of the vectors ϕ' obtained from ϕ by lowering some component by one is again approximate eigenvector. Perhaps it is true that independent *tight* approximate eigenvectors always produce distinct code systems.

7.8 Applications

We will use our previous results to show that certain "good" codes can be obtained by ae-consistent state-splitting only with an approximate eigenvector containing "large" components. The main problem of code construction with the ae-consistent state-splitting algorithm is the enormous amount of freedom, both in the choice of state-splitting steps and labelling with source symbols, and in the choice of an approximate eigenvector. Given an approximate eigenvector, some heuristics have been developed [7], [3] to guide the choice of the state-splitting steps in order to obtain "good" codes. Also, both practical and theoretical considerations seem to suggest to choose an approximate eigenvector that is "small" and, in particular, to choose one with smallest maximal component. The following example shows in a rather strong sense that such a choice does not always lead to the best code available. Indeed, for each $N \geq 3$, we will construct a constrained system of finite type $\mathcal{S}_N$, of capacity at least N, for which an approximate eigenvector can be found with maximal component equal to two. We will show that each $\mathcal{S}_N$ affords a one-symbol encodable, block-decodable code $\mathcal{C}_N$, but that $\mathcal{C}_N$ can only be constructed by ae-consistent state-splitting with an approximate eigenvector whose maximal component is at least N. In fact, we will shown that *no* BDB code [4] can be found by ae-consistent state-splitting with an approximate eigenvector whose maximal component is less than N. Here, a BDB code essentially is a block-decodable code that can be encoded by a finite-state encoder with a finite, bounded encoding delay. In this context, the labels of $\mathcal{M}$ are often referred to a codewords. The collection of labels that decode to a fixed source symbol is called a *codeword class*.

Example 1: Let N be a fixed integer with $N \geq 3$, and let $n = \lfloor N/2 \rfloor$. We consider the constrained system $\mathcal{S}$ with alphabet $\boldsymbol{F} = \{1, 2, \ldots, N+1\}$ and forbidden words

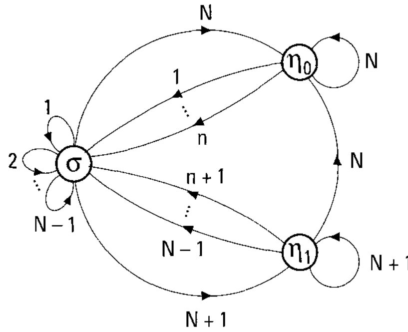

Figure 1: *The FSTD $\mathcal{M}$.*

(i) $Nj, \qquad n+1 \leq j \leq N-1$ or $j = N+1$,
(ii) $(N+1)j, \qquad 1 \leq j \leq n$.

This constrained system $\mathcal{S}$ can be presented by the three-state FSTD $\mathcal{M}$ in Figure 1. Note that $\mathcal{M}$ is deterministic, and even of finite type, with memory 1 and anticipation 0. In fact, $\mathcal{M}$ is the minimal deterministic presentation, the Shannon cover, of $\mathcal{S}$.

It is not difficult to see that the vector ϕ with components $\phi_\sigma = 2$ and $\phi_{\eta_0} = \phi_{\eta_1} = 1$ is the (component-wise) smallest $[\mathcal{M}, N]$-approximate eigenvector. The vector ϕ' with components $\phi'_\sigma = N$, $\phi'_{\eta_0} = n$, and $\phi'_{\eta_1} = N - n$ is also an $[\mathcal{M}, N]$-approximate eigenvector.

We first describe a one-symbol look-ahead encodable, block-decodable code for $\mathcal{S}$ with source alphabet $\boldsymbol{B} = \{1, 2, \ldots, N\}$. The codeword classes of this code are given by $\boldsymbol{F}_i = \{i\}$, $1 \leq i \leq N-1$, and $\boldsymbol{F}_N = \{N, N+1\}$. In fact, it is not difficult to verify that *any* block-decodable code for $\mathcal{S}$ with source alphabet of size $N \geq 3$ necessarily consists of these codeword classes. This observation will be used later on.

We now describe an encoding procedure which shows that one-symbol look-ahead encoding is indeed possible. An encoder for this code starts in state σ and encodes source symbol i, $1 \leq i \leq N-1$, by the transition from the actual state to state σ with label i and the source symbol N by the transition from the actual state to state η_0 or η_1, depending on whether the next source symbol is at most n or larger than n.

It is easily seen that the collection $\mathcal{C}$ of possible code sequences consists of all sequences from $\mathcal{S}$ that do not contain "NN", and so $\mathcal{C}$ is generated by the FSTD $\mathcal{M}'$ obtained from $\mathcal{M}$ by deleting the loop at state N. (Note that $\mathcal{M}'$ has capacity exactly $\log N$.) From this description of $\mathcal{C}$ and from Theorem 6, it is evident that the collection $\tilde{\Sigma}_v^{(n)}$ consists precisely of all paths of length n from v in $\mathcal{M}'$. As a consequence, the vector ϕ', which is (up to a constant) the unique

$[\mathcal{M}', N]$-approximate eigenvector, is contained in the linear span of the vector $\tilde{\theta}$ as defined in Section 7.4. This can be shown directly, but also by constructing an encoder graph $\mathcal{N}$ for this code by state-splitting and then using Theorem 4.
To construct such an encoder $\mathcal{N}$, we first delete the loop at η_0 in $\mathcal{M}$, then we split state σ into N states σ^j, for $1 \leq j \leq N$, state η_0 into n states η_0^j, for $1 \leq j \leq n$, and state η_1 into $N - n$ states η_1^j, for $n + 1 \leq j \leq N$; here we assign transitions with label j to split states s^j with index j (or to η_1^N if $j = N + 1$). Note that no children of the loop at η_0 in $\mathcal{M}$ are present in $\mathcal{N}$ and that all possible children of other transitions *are* present in $\mathcal{N}$. Since each state in $\mathcal{N}$ has out-degree N, it follows that $\mathcal{N}$ is an encoder graph. Moreover, $\mathcal{N}$ is indeed an encoder for the block-decodable code described earlier (with an encoding delay of one symbol), and each state v from $\mathcal{M}$ has ϕ'_v children in $\mathcal{N}$, that is, $\phi' = \gamma$. The decoder for this code will have a window-size of two symbols, but decoding only depends on the second symbol in the window.

It is not difficult to verify that for odd N, the smallest positive integer vector contained in the linear span of ϕ' is ϕ' itself. So we have shown that for odd N, both the vector θ obtained as in Theorem 7 from $\mathcal{M}$ and $\mathcal{C}$ and the vector γ^* are equal to ϕ'. As a consequence, Theorem 7 now implies that for odd N, the code system $\mathcal{C}$ can only be obtained by ae-consistent state-splitting with an approximate eigenvector φ for which $\varphi \geq \phi'$. In particular, such a approximate eigenvector φ will have a maximal component at least equal to N.

However, when N is even, we have that $\gamma^* = \phi$. Indeed, for even N, the vector ϕ is an $[\mathcal{M}', N]$-approximate eigenvector, where $\mathcal{M}'$ is as defined earlier, and $\mathcal{M}'$ has capacity $\log N$. So code construction by state splitting from the pair $(\mathcal{M}, \phi)$ may start by the deletion of the loop in $\mathcal{M}$ at η_0, and, in that case, will lead to a code with code system $\mathcal{C}$, which is the system presented by $\mathcal{M}$. Nevertheless, the *code* thus obtained will not be block-decodable. In fact, we will show that if a block-decodable code is obtained by ae-consistent state-splitting with an approximate eigenvector φ, then necessarily $\varphi_v \geq N$, $v = 1, \ldots, N - 1$. (Those readers not interested in the details may skip the remainder of this example.)

To see this, let us assume that we obtain a decoder graph $\mathcal{N}$ by $M \geq 1$ rounds of state-splitting (and possibly the deletion of some transitions and states), and that the resulting code is equivalent to a block-decodable code, in the sense of [3]. (That is, the code will be block-decodable up to a fixed "delay", see below.) The decoder for this code will have a decoding window of size w, say, where $2 \leq w \leq M + 1$. (Since fewer than N transitions leave states N and $N + 1$, w is at least two.) Moreover, as shown in [3], the fact that the code is equivalent to a block-decodable code implies that the actual encoding will depend on the *last* symbol in the window only; here we may asume without loss of generality that the codeword classes are the $\boldsymbol{F}_i$ defined earlier and that a member of $\boldsymbol{F}_i$ decodes

as the source symbol i.

We claim that such a code cannot exist if state σ has less than N children in $\mathcal{N}$. This can be seen as follows. Fix i, $1 \leq i \leq N-1$. First note that there must be a child of state σ in $\mathcal{N}$, say state σ^1, and a child of the loop α with label i at state σ in $\mathcal{M}$, say α^1, such that α^1 is a loop at σ^1 in $\mathcal{N}$. Indeed, the source sequence

$$\ldots i, i, i, \ldots$$

is necessarily encoded by a path in $\mathcal{N}$ that (with the possible exception of the first transition) consists of children of transition α only, and therefore the path eventually consists of the repetition of a single loop in $\mathcal{N}$.

Now, consider the encoding of a source sequence of the form

$$\ldots i, i, i, k, k, k, \ldots, \tag{6}$$

where $1 \leq k \leq N$. After encoding of the last source symbol i, the encoder will be in state σ^1. Then the next source symbol k will be encoded by a transition in $\mathcal{N}$ that carries source label k and (codeword) label v, say. Hence the decoding window of the $(w-2)$-but-last source symbol i looks like

$$i, i, \ldots, i, v.$$

By our earlier remark, this window decodes as v (or as N if $v = N+1$) and should decode as i, therefore $v = i$, so the transition that encodes k is also a child of α.

Now, if state σ has fewer than N children in $\mathcal{N}$, then the transition α can have at most $N-1$ children in $\mathcal{N}$, hence some source symbol k does not occur as source label on any of these children, and, by the above reasoning, for this k the source sequence (6) cannot be encoded properly. □

7.9 Discussion

There has always existed an intimate relation between code construction methods on the one hand and approximate eigenvectors on the other hand. In an attempt to explain some of the reasons for this connection, we have investigated the relation between a modulation code for a given constrained system and the approximate eigenvector used in ae-consistent state-splitting to construct the code.

In the case that the constraint is of finite type, our main result is a lower bound on this approximate eigenvector, determined by the code sequences (the code system) of the code only. It seems possible to derive a similar result in

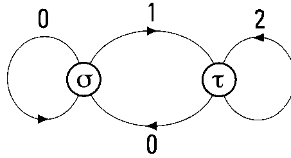

Figure 2: *The FSTD $\mathcal{M}$ for Example 2.*

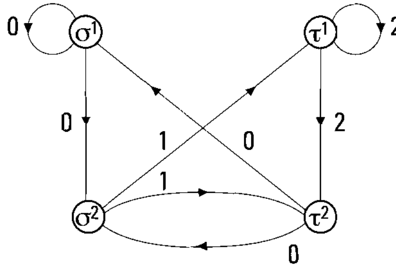

Figure 3: *The encoder $\mathcal{N}$ for Example 2.*

the case where the constraint is of almost-finite type, but we did not pursue this question here.

We also show that for each irreducible finite-type code system there exists a unique smallest approximate eigenvector to construct this code system, but the problem to give a constructive description of this vector remains unresolved.

Appendix

For the sake of completeness we collect some examples illustrating claim 2 in Section 7.7.

Example 2: Let $\mathcal{M}$ be the FSTD in Figure 2. Let $\mathcal{S}$ denote the constrained system presented by $\mathcal{M}$. Again, we take $N = 2$.

Consider the encoder $\mathcal{N}$ for $\mathcal{S}$ in Figure 3. Obviously, we can obtain $\mathcal{N}$ using an approximate eigenvector $\phi = (2, 2) = \gamma$, or $\phi = (3, 2)$, or $\phi = (2, 3)$, but not with $\phi = (1, 1)$. Also obviously, $\mathcal{N}$ and $\mathcal{M}$ generate the same system. The presentation $\mathcal{N}$ is not minimal; it can be reduced by merging states σ^1 and τ^2. Note that the pair of states that are merged do not consist of children of the same state in $\mathcal{M}$. □

Example 3: Let $\mathcal{M}$ be the FSTD in Figure 4. Again, we take $N = 2$. First, we consider the $[\mathcal{M}, 2]$-approximate eigenvector $\theta = (2, 1, 1)$. We have to split state 1, which can be done in only one way, and the result is a state 1^0 with

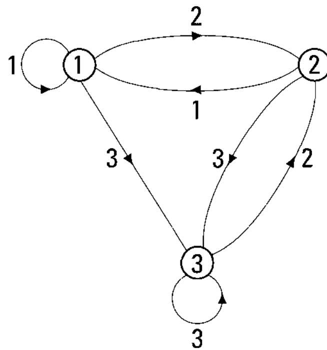

Figure 4: *The FSTD $\mathcal{M}$ for Example 3.*

successors 2, 3 and a state 1^1 with successors 1^0 and 1^1. (After this step all states have weight 1 and at least two successors.) To obtain an encoder, we have to remove one of the three successors 1^0, 1^1, or 3, of state 2. Therefore, depending on which of the three successors will be deleted, the resulting code system $\mathcal{C}$ will have as a forbidden blocks either 211, or 212 and 213, or 23.

Next, we consider the $[\mathcal{M}, 2]$-approximate eigenvector $\phi = (3, 2, 1)$. Here, to obtain an encoder we have to split states 1 and 2, and then to delete some successor of state 3 in the graph thus obtained. We leave it to the reader to verify that in all cases the forbidden blocks referred to above will be present in the resulting code system $\mathcal{C}'$, so that in all cases $\mathcal{C} \neq \mathcal{C}'$. □

References

[1] R.L. Adler, D. Coppersmith, and M. Hassner, "Algorithms for sliding block codes. An application of symbolic dynamics to information theory", *IEEE Trans. Inform. Theory*, vol. IT-29, no. 1, pp. 5–22, Jan. 1983.

[2] J. Ashley and B.H. Marcus, "Canonical encoders for sliding block decoders", *Siam J. Disc. Math.*, vol. 8, No. 4, pp. 555–605, Nov. 1995.

[3] H.D.L. Hollmann, "On the construction of bounded-delay encodable codes for constrained systems", *IEEE Trans. Inform. Theory*, vol. 41, No. 5, pp. 1354–1378, Sept. 1995.

[4] H.D.L. Hollmann, "Bounded-delay encodable, block-decodable codes for constrained systems", *IEEE Trans. Inform. Theory*, vol. 42, no. 6, Nov. 1996.

[5] H.D.L. Hollmann, "A block-decodable $(d, k) = (1, 8)$ runlength-limited rate $8/12$ code", *IEEE Trans. Inform. Theory*, vol. 40, No. 4, pp. 1292–1296, July 1994.

[6] B.H. Marcus and R. Roth, "Bounds on the number of states in encoder graphs for input-constrained channels", *IEEE Trans. Inform. Theory*, vol. IT-37, no. 3, pp. 742–758, May 1991.

[7] B.H Marcus, P.H. Siegel, and J.K. Wolf, "Finite-state modulation codes for data storage", *IEEE J. Sel. Areas Commun.*, vol. 10, no. 1, pp. 5–37, Jan. 1992.

[8] E. Seneta, *Nonnegative matrices and Markov chains*, New York: Springer, 1981.

[9] J.C. Willems, "Models of dynamics", in *Dynamics Reported, vol. 2*, pp. 171–266, New York: Wiley, 1989.

Summary

Modulation codes

This thesis concerns digital modulation codes or channel codes as employed in digital transmission or storage systems (e.g., a compact disc system). The data processing in such systems can be sketched as follows. At the senders side, the analogue data stream at the input (e.g., sound captured by a microphone) is first sampled and digitized. Then the digital data thus obtained is compressed. Next, the compressed data is protected by means of an error-correcting code. Finally, the resulting digital data is transformed by means of a modulation code into a data sequence that possesses certain desirable properties, referred to as an admissible sequence, which is subsequently transmitted or stored on a medium such as disk or tape, in analogue form (e.g., written on a compact disk).

At the receiver's side, the whole process is reversed. First the original digital form of the transmitted or stored data is recovered as well as possible from the analogue data that is received or read, and is transformed back by means of the modulation code. Then the error-correcting code is used to correct possibly corrupted parts of the data. Finally, the corrected data is decompressed and restored to its original analogue form.

In the theory of modulation codes the following topics play an important role: kinds of collections of admissible sequences; construction of encoders that realise the transformation from arbitrary sequences to admissible sequences and the corresponding decoders to recover the original sequences from their transformed form; the complexity and other aspects of encoders and decoders; and code efficiency.

As said above, modulation codes are employed to transform arbitrary data sequences into special, admissible sequences. Which sequences are admissible depends on various technical constraints imposed by the system for which the code is designed. Well-known examples that also play a role in this thesis are the "dc-balanced" sequences, bipolar ($\{-1, 1\}$-valued) sequences whose running digital sum does not surpass certain specified values (as a consequence, such se-

quences have a spectral zero at the null-frequency, "at dc"); the (d, k)-constrained sequences, binary sequences in which each two consecutive symbols "1" are separated by at least d and at most k symbols "0"; and the related class of (d, k) RLL sequences, binary sequences in which each group of successive symbols "0" or "1" in the sequence has size at least $d + 1$ and at most $k + 1$. (The practical relevance of such constraints is explained in the first chapter of this thesis.)

In all important applications, the collection of admissible sequences can be conveniently described as the collection of label sequences generated by the walks in a certain labeled directed finite graph. (Think of such a graph as of a description of the crossings and their connecting roads of a town where only one-way traffic is allowed. The labels are the street-names; the admissible sequences are all those sequences of street-names that correspond to possible tours in the town.)

The graph describing a given collection of admissible sequences is not unique, and in fact many code construction methods for modulation codes (i.e., of encoders and decoders) essentially depend on finding a suitable graph to describe the given collection of sequences.

An important type of constraint often encountered in applications is that no member from a given finite set of patterns is allowed to occur in a transmitted sequence. (For example, it is not difficult to see that the (d, k) constraint mentioned earlier is of this type.) A collection of sequences that satisfy such a "constraint of finite type" always admits a description by means of a suitable graph.

Obviously, there are more arbitrary sequences of a certain length than admissible sequences of that length. This simple observation has as important consequence that encoding in general will necessarily result in a longer admissible sequence: more bits will be required to represent the information. The average ratio of the length of the original sequence to the length of the corresponding encoded sequence is called the "rate" of the code. For example, if encoding doubles the length, then the rate equals $1/2$. ("One bit of information per two encoded bits".) The rate of a code is always a number between zero and one.

In general we can say, "the higher the rate the better". However, it turns out that for each collection of admissible sequences that can be described by means of a graph, there actually is a limit to the achievable rate. This maximal rate is called the capacity of the collection of admissible sequences. The capacity can be computed relatively straightforward from a given graph description. The efficiency of a code is the ratio of the rate of the code to the capacity.

Generally speaking, the received sequence to be decoded will not be exactly the same as the originally transmitted sequence, due to, e.g., possible damage to the storage medium (scratches on the disk) in the case of data storage, or bad weather conditions (lightning) in the case of data transmission. In other words, the received sequence may contain errors. The decoder will have to be able to

deal with these errors: we cannot allow the situation where a single erroneous bit can cause the decoder to produce an enormous amount of errors in the original sequence. (The effect that a single error may lead to several errors after decoding is called error propagation.)

For this reason, most practical decoders employ a decoding window of a fixed size. Here, to recover a specific original bit, only a part of the received sequence is involved, namely only the part that falls within the decoding window of this bit. We can now imagine the decoding process as a sequence of decoding decisions, where we slide the decoding window over the sequence that is to be decoded. This explains the pictorial name "sliding-block decoder" in use for this type of decoder. Note that an error in the received sequence only influences the decoding decision as long as it is contained in the decoding window, hence indeed affects only a limited number of the recovered bits.

The size of the decoding window is an important parameter of a code: not only does this size provide an upper bound to the expected amount of error propagation, it also gives an indication of the complexity (and hence of the size of a hardware implementation) of the decoder.

A special case of these sliding-block decodable codes is constituted by the block-decodable codes. These codes can be decoded by first grouping the received bits into blocks of a fixed size, followed by separate decoding of each of the blocks, independent of preceding and succeeding blocks.

In Chapter 1 we first give an overview of the different parts that constitute a digital communication scheme. Here, we emphasize the mathematical principles that underly the functioning of these parts and their interaction with various practical constraints. Then we present an extensive overview of the various aspects of the theory of modulation codes, as well as an extensive discussion of and motivation for the further material in this thesis. All the topics mentioned above are reviewed again in more detail, on a somewhat deeper level but as far as possible in an elementary way. With an exception here and there, this chapter consists of known material.

In Chapter 2 we compare two modulation codes for the dc-balanced constraint, namely the "polarity-bit" code, and a code introduced by Knuth. It had been shown earlier that the "sum variance" of such codes constitutes a good performance criterium. My contribution here is an exact evaluation of a good approximation to the sum variance of the Knuth code, which required the solution of a combinatorial counting problem. The results show that the performance of this code in general does not justify its greater complexity.

In many applications, the encoded data is grouped into large blocks of a fixed size called "frames". Here, the beginning of each frame is marked by a fixed synchronization pattern that is not allowed to occur anywhere else in the data

stream. This requirement leads to a certain loss of capacity. In Chapter 3 we describe a method to compute this loss of capacity for each given pattern when moreover the encoded data has to obey a (d, k) constraint. My contribution here mainly concerns the employed enumerative techniques.

The next three chapters are concerned with BD codes, sliding-block decodable codes for constraints of finite type that can be encoded with a bounded encoding delay (or, equivalently, by looking ahead at a limited number of bits).

One of the most important code construction methods is the ACH state-splitting algorithm. This method acts directly on a graph that presents the constraint; in a number of rounds of state-splitting, this graph is guaranteed to be transformed into an encoder for the constraint. This process is guided by an "approximate eigenvector", certain integer weights associated with the vertices ("states") of the graph. In general, this algorithm offers much freedom-of-choice, and, unfortunately, it is not obvious how to use this freedom to obtain a code with a small decoding window.

In Chapter 4, a method is developed that at least partly overcomes this drawback. The method is based on a new, simple "local" construction method for BD codes. Here, in each vertex of the weighted graph, a certain partition problem has to be solved, a problem that is easily dealt with as long as the number of edges leaving the relevant vertex is small. If in each vertex this partition problem is successfully solved, then a code is readily obtained; moreover, optimization of the decoding window size of the resulting code is possible.

It may happen that the partition problem in some of the vertices is unsolvable. However, this difficulty can always be dealt with by means of a number of rounds of state-splitting, where moreover the very difficulty now offers additional guidance. It turns out that this approach allows the construction of both already known as well as new, good codes.

In Chapter 5, a construction method is presented for block-decodable BD codes (BDB codes); in addition we show that each "suitable" BDB code can— at least in principle— be obtained by this method. Our results here imply that the construction method from Chapter 4 is universal: in principle, each "suitable" BD code can be obtained by that method. This, in turn, provides an alternative proof for the fact that the ACH algorithm is also universal. Furthermore, we use our results to solve some decision problems.

Subsequently, the method is employed in Chapter 6 to construct a block-decodable $(1, 8)$ RLL code of rate $8 \to 12$. Such a code is attractive to employ in combination with symbol-error-correcting codes such as Reed-Solomon codes over an 8-bit alphabet.

Many known code construction methods employ an approximate eigenvector in an essential way; different choices for this vector in general lead to different

codes. The work in Chapter 7 tries to shed some light on this intimate connection between codes and approximate eigenvectors. First we describe a construction that produces an approximate eigenvector for a graph, assuming this graph and a modulation code for the constraint presented by this graph. Then we show for example that the following holds: whenever it is possible to construct the given code using the ACH algorithm in combination with another approximate eigenvector, then this other vector is, components-wise, at least as large as the constructed vector. Moreover, we present an example showing that the "smallest" approximate eigenvector does not always allow the construction of the simplest possible code.

Henk D.L. Hollmann

Samenvatting

Modulatie codes

Dit proefschrift handelt over digitale modulatie codes, ook wel kanaal codes genoemd, zoals aangewend in digitale data transmissie- en opslag-systemen (een compact disc systeem bijvoorbeeld). De data processing in zulke systemen verloopt ruwweg als volgt. Aan de kant van de zender wordt de aangeboden analoge data stroom (bijvoorbeeld geluid geregistreerd door een microfoon) eerst bemonsterd en gedigitaliseerd. Daarna wordt de aldus verkregen digitale data gecomprimeerd. Vervolgens wordt de gecomprimeerde data beschermd middels een fouten-corrigerende code. Tenslotte wordt de hierdoor verkregen rij digitale data door middel van een modulatie code getransformeerd in een data reeks met zekere gewenste eigenschappen, een zogeheten toegestaan rijtje, en vervolgens in analoge vorm uitgezonden of geschreven op een medium zoals disk of tape (bijvoorbeeld opgeslagen op een compact disc).

Aan de kant van de ontvanger wordt het hele proces in omgekeerde volgorde doorlopen. Eerst wordt de oorspronkelijke digitale vorm van de uitgezonden of opgeslagen data zo goed mogelijk gereconstrueerd uit de ontvangen of gelezen analoge data, en terug getransformeerd middels de modulatie code. Vervolgens wordt de fouten-corrigerende code gebruikt om eventueel verminkte delen van de data te herstellen. Daarna wordt de gecorrigeerde data gedecomprimeerd en omgezet in z'n oorspronkelijke analoge vorm.

In de theorie van modulatie codes spelen de volgende zaken een belangrijke rol: soorten collecties van toegestane rijtjes; constructie van encoders om de transformatie van willekeurige rijtjes naar toegestane rijtjes te realiseren en de bijbehorende decoders nodig voor het terugwinnen van het oorspronkelijke rijtje uit z'n getransformeerde vorm; de complexiteit en andere eigenschappen van encoders en decoders; en efficientie van codes.

Zoals gezegd wordt de modulatie code gebruikt om willekeurige data rijtjes om te zetten in speciale, toegestane rijtjes. Welke rijtjes toegestaan zijn wordt bepaald door allerlei technische eisen, opgelegd door het systeem waarvoor de code

wordt ontworpen. Bekende voorbeelden die in dit proefschrift ook een rol spelen zijn de 'DC-balanced' rijtjes, bipolaire ($\{-1, 1\}$-waardige) rijtjes waarvan de lopende digitale som bepaalde gespecificeerde waarden niet overstijgt (het spectrum van zo'n rijtje heeft dan een nulpunt in de nul-frequentie, 'in DC'); de (d, k) rijtjes, binaire ($\{0, 1\}$-waardige) rijtjes waarin elk tweetal opeenvolgende enen gescheiden wordt door tenminste d en ten hoogste k nullen, en de hiermee verwante klasse van '(d, k) RLL' rijtjes, binaire rijtjes waarin het aantal achtereenvolgende nullen of enen tenminste $d + 1$ en ten hoogste $k + 1$ bedraagt. (De practische relevantie van zulke eisen wordt verklaard in het eerste hoofdstuk van dit proefschrift.)

In alle belangrijke toepassingen kan de collectie toegestane rijtjes beschreven worden als de collectie van label rijtjes gegenereerd door de wandelingen in een zekere gelabelde gerichte eindige graaf. (Denk aan een gerichte eindige graaf als aan een beschrijving van de kruispunten en bijbehorende verbindingswegen van een stad met alleen éénrichtingsverkeer. De labels zijn de straatnamen; de toegestane rijtjes zijn alle rijtjes van straatnamen die horen bij mogelijke routes door de stad.)

Er zijn vele grafen die een gegeven verzameling toegestane rijtjes beschrijven, en veel constructie methoden voor modulatie codes (dat wil zeggen, van encoders en decoders) berusten in essentie op het vinden van een geschikte graaf voor de gegeven collectie rijtjes.

Een belangrijke en veel in de praktijk voorkomende soort beperking is dat géén van een gegeven, eindig aantal patronen in de uit te zenden rijtjes mag voorkomen. (Het is bijvoorbeeld eenvoudig na te gaan dat de eerder genoemde (d, k) eisen van dit type zijn.) De collectie van rijtjes die voldoen aan zo'n 'beperking van eindig type' kan altijd worden beschreven middels een geschikte graaf.

Er zijn meer willekeurige rijtjes van een gegeven lengte dan toegestane rijtjes van die lengte. Deze simpele observatie heeft als belangrijke consequentie dat encoderen van een willekeurig rijtje in het algemeen zal moeten resulteren in een langer toegestaan rijtje: er zullen dus meer bits nodig zijn om de informatie te representeren. De gemiddelde verhouding van de lengte van het oorspronkelijke rijtje tot de lengte van het bijbehorende gecodeerde rijtje heet de 'dichtheid' van de code. Als bijvoorbeeld encoderen leidt tot twee maal zo lange rijtjes, dan zeggen we dat de dichtheid van de code gelijk is aan $1/2$. ('Eén informatie bit per twee gecodeerde bits'.) De dichtheid van een code is een getal dat altijd tussen nul en één ligt.

In het algemeen geldt: hoe hoger de dichtheid hoe beter. Echter, het blijkt dat voor iedere verzameling van toegestane rijtjes beschreven door een graaf er een grens is aan de mogelijke dichtheid van een code voor deze verzameling. Deze maximaal te realiseren dichtheid heet de capaciteit van de verzameling toegestane

rijtjes. Deze capaciteit kan relatief eenvoudig worden bepaald uit de beschrijvende graaf. De efficientie van een code is de verhouding tussen de dichtheid van de code en de capaciteit.

In het algemeen zal het ontvangen, te decoderen rijtje niet precies hetzelfde zijn als het oorspronkelijk uitgezonden of opgeslagen rijtje, dit ten gevolge van bijvoorbeeld optredende beschadigingen van het opslagmedium (krassen op de cd) in het geval van data opslag, of slechte weersomstandigheden (bliksem) in het geval van data transmissie. Met andere woorden, het ontvangen rijtje zal fouten bevatten. De decoder zal tegen dit verschijnsel bestand moeten zijn: het mag niet zo zijn dat een enkel fout bit na decoderen kan resulteren in het optreden van een enorm aantal fouten in het oorspronkelijke rijtje. (Het effect dat een enkele fout kan leiden tot meerdere fouten na decoderen heet fout-propagatie.)

Om deze reden wordt van practische decoders meestal geeist dat zij gebruik maken van een decodeer-venster van vaste afmeting. Hierbij wordt voor het terugwinnen van elk specifiek bit slechts een beperkt deel van het te decoderen rijtje gebruikt, namelijk alleen dat deel dat valt binnen het decodeer-venster van dit bit. Het decodeer proces laat zich nu voorstellen als een reeks van decodeer beslissingen, waarbij het decodeer-venster over het te decoderen rijtje heen geschoven wordt. Dit verklaart de beeldende naam 'sliding-block decoder' voor dit type decoder. Een fout in het ontvangen rijtje heeft nu slechts invloed op het decodeer proces zolang deze fout valt binnen het decodeer-venster, en beinvloed dus inderdaad slechts een gelimiteerd aantal te decoderen bits.

De grootte van het decodeer-venster is een belangrijke parameter van een code: behalve dat deze grootte een bovengrens geeft aan de te verwachten hoeveelheid fout-propagatie, vormt het ook een indicatie voor de complexiteit (en dus de grootte van een hardware implementatie) van de decoder.

Een speciaal geval van sliding-block decodeerbare codes zijn de blok-decodeerbare codes. Zulke codes kunnen worden gedecodeerd door de ontvangen bits te groeperen in blokken van vaste grootte, en vervolgens elk blok afzonderlijk te decoderen, onafhankelijk van voorafgaande en volgende blokken.

In Hoofdstuk 1 geven we eerst een overzicht van de verschillende delen van een digitaal communicatie systeem. De nadruk ligt hier op de wiskundige principes waarop deze onderdelen zijn gebaseerd en hun interactie met allerlei practische eisen. Vervolgens geven we een zeer gedetailleerd overzicht van de theorie van modulatie codes, alsmede een uitgebreide bespreking van en motivatie voor het verdere werk in dit proefschrift. Alle hierboven genoemde aspecten komen hier opnieuw en in meer detail aan de orde, wat meer uitgediept maar voorzover mogelijk op elementaire wijze. Op hier en daar een uitzondering na bestaat dit hoofdstuk uit bekend materiaal.

In Hoofdstuk 2 worden twee modulatie codes voor de DC-balanced eis, na-

melijk de 'polarity-bit' code en een code bedacht door Knuth, met elkaar vergeleken. Al eerder was aangetoond dat de 'sum-variance' van dergelijke codes een goede maat is voor hun performantie. Mijn bijdrage hier is een exacte evaluatie van een goede benadering van deze parameter voor de Knuth-code, waarvoor een combinatorisch tel-probleem moest worden opgelost. De resultaten tonen aan dat de grotere complexiteit van deze code in het algemeen niet wordt gerechtvaardigd door zijn performantie.

In veel toepassingen is de gemoduleerde data gegroepeerd in grote blokken van vaste grootte, 'frames' geheten; hierbij wordt het begin van elk frame gemarkeerd door een vast synchronisatie patroon dat verder op geen andere plaatsen in de data stroom mag voorkomen. Deze eis leidt tot een zeker verlies aan capaciteit. In Hoofdstuk 3 beschrijven we een methode om voor elk gegeven patroon dit resulterende capaciteitsverlies te berekenen als tevens de gemoduleerde data aan een (d, k) eis moet voldoen. Mijn bijdrage hier betreft voornamelijk de gebruikte enumeratieve technieken.

De volgende drie hoofdstukken handelen over BD codes, sliding-block decodeerbare codes voor eisen van eindig type die met begrensde vertraging (of, equivalent, met in beperkte mate 'vooruitkijken) kunnen worden geëncodeerd.

Een van de belangrijkste constructie methoden voor sliding-block modulatie codes is de ACH staat-splits algorithme. Deze methode werkt direct op een beschrijvende graaf voor de gegeven eis; in een aantal ronden van staat-splitsing wordt deze gegarandeerd getransformeerd in een encoder voor een modulatie code. Dit proces wordt gestuurd door een 'approximatieve eigen-vector', zekere geheeltallige gewichten geassocieerd met de knooppunten ('staten') van de graaf. De algorithme biedt echter meestal zeer veel keuzevrijheid en het is helaas niet duidelijk hoe deze te gebruiken om te komen tot een code met klein decodeervenster.

In Hoofdstuk 4 wordt een methode ontwikkeld om dit nadeel voor een deel te ondervangen. De methode is gebaseerd op een nieuwe eenvoudige 'locale' constructie methode voor BD codes. Hierbij moet in elk knooppunt van de gewogen graaf afzonderlijk een zeker partitie-probleem worden opgelost, een eenvoudig probleem zolang het aantal uitgaande kanten in de desbetreffende staat klein is. Is dit in elk knooppunt gelukt dan kan een code verkregen worden; optimalisatie naar venstergrootte is hierbij mogelijk. Het is mogelijk dat het partitie-probleem in sommige knooppunten onoplosbaar is. Deze moeilijkheid kan echter altijd worden opgelost middels een aantal ronden van staat-splitsing, waarbij de eerdere moeilijkheid nu juist extra sturing verschaft. Met deze aanpak blijkt het mogelijk om zowel reeds bekende als ook nieuwe goede codes te verkrijgen.

In Hoofdstuk 5 wordt een methode beschreven voor de constructie van blokdecodeerbare BD codes (BDB codes), en wordt aangetoond dat elke 'geschikte'

BDB code—tenminste in principe—middels deze methode verkregen kan worden. Een belangrijke consequentie is dat de constructie methode uit Hoofdstuk 4 universeel is: in principe kan elke 'geschikte' BD code worden verkregen middels die methode. Dit, op z'n beurt, levert een alternatief bewijs van het feit dat de ACH algorithme ook universeel is. Verder gebruiken we onze resultaten voor de oplossing van enkele beslissingsproblemen.

Vervolgens wordt in Hoofdstuk 6 de bovengenoemde methode aangewend voor de constructie van een blok-decodeerbare $(1, 8)$ RLL code van dichtheid $8 \to 12$. Een dergelijke code kan profijtelijk gebruikt worden in combinatie met fouten-corrigerende codes zoals Reed-Solomon codes over een 8-bits alphabet.

Vele bekende constructie methoden maken een essentieel gebruik van approximatieve eigen-vectoren, waarbij verschillende keuzen in het algemeen tot verschillende codes leiden. In Hoofdstuk 7 wordt getracht het verband tussen codes en approximatieve eigen-vectoren te verduidelijken. Eerst beschrijven we een constructie van een approximatieve eigen-vector voor een gegeven graaf uitgaande van deze graaf en een modulatie code voor de eis beschreven door die graaf. Vervolgens tonen we onder andere aan dat het volgende geldt: als de ACH algorithme toegepast met een andere approximatieve eigen-vector de gegeven modulatie code kan construeren, dan is deze andere vector components-gewijs minstens zo groot als de geconstrueerde vector. Door middel van een voorbeeld wordt aangetoond dat de 'kleinste' approximatieve eigen-vector niet altijd leidt tot de eenvoudigste code.

Henk D.L. Hollmann

Curriculum Vitae

Henk Hollmann was born in Utrecht, the Netherlands, on March 10, 1954. He received his Masters degree in mathematics (cum laude) from Eindhoven University of Technology, with a thesis on association schemes.

In 1982 he joined the CNET, Issy-les-Moulineaux, France, where he worked mainly on Number Theoretic Transforms. Since 1985 he is with Philips Research Laboratories, Eindhoven, the Netherlands.

His research interests include discrete mathematics/combinatorics, information theory, cryptography, and digital signal processing as well as their technical applications.

Index

A

A/D conversion 1, 3
A/D converter 2
ACH algorithm 37
adjacency matrix 18, 25
analogue 1, 2
analogue-to-digital conversion, *see* A/D
anticipation 22, 28
approximate eigenvector 30, 31, 47
average amount of information 16

B

balanced 33
band-limited 4
BD method 37
bi-phase code, *see* code
binary representation 40
binary symmetric memoryless channel 6
bit 2, 3, 16
– channel 3, 14
– information 6
– merging 29, 32
– parity-check 6
– polarity 33
– source 14, 25
bit-stream 3, 43
block 29
block-code, *see* code
block-decodable 29, 37
bounded-delay encodable code, *see* code
bounded-delay encoding, *see* encoding
bounded-delay method 36
bounded-distance decoding, *see* decoding
burst 10
bursty channel 10

C

capacity 7, 14, 24, 25
channel 2, 6
channel bit, *see* bit
characteristic polynomial 19
code
– bi-phase 25
– block- 7
– bounded-delay encodable 36
– column 8
– convolutional 7
– enumerative 41
– error-correcting 3, 6, 8, 11
– Frequency Modulation (FM) 25
– high-rate 39
– Miller 29
– Modified FM (MFM) 29
– modulation 3, 11, 14
– periodic enumerative 43, 44
– polarity-bit 33
– prefix 5
– principal-state
  – fixed-length 32
  – variable-length 36
– product 8
– Reed-Solomon 10
– row 8
– source 5
– symbol-error-correcting 10
– turbo 7
code sequence, *see* sequence

code system, *see* system
codeword 7, 25, 31
column code, *see* code
complexity (of decoder), *see* decoder
compression 2
compression rate 5
concatenation 25
constrained sequence, *see* sequence
constrained system, *see* system
constraint
– (d, k) 12
– frequency domain 13
– incidental 30
– presentation of 17
– run-length 12
– spectral null 13
constraint of finite type, *see* finite-type
convolutional code, *see* code
cooperating codes 8

D

DC-balanced 13
DC-free 13
decoder 14, 27
– complexity 28
– size 28
– sliding-block 20n, 28
– Viterbi 9
decoding 6, 8, 26, 27
– bounded-distance 8
decoding rules 6
decoding window 39
decompression 3
delay 27
demodulation 3
deterministic, *see* presentation
digital communication system, *see* system
disparity 33
distance 6, 7
distortion measure 6
(d, k) constraint, *see* constraint
(d, k) sequence, *see* sequence
(d, k)-RLL sequence, *see* sequence
(d, k)-constrained sequence, *see* sequence
(d, k)-constrained system, *see* system
$(dklr)$-sequence, *see* sequence

E

efficiency 14, 26
eigenvector 19
empty list 35
encode 11
encoder 14, 25, 39
– look-ahead 27, 35
encoding
– bounded-delay 46
– enumerative 27, 39
encoding delay 27
encoding path 26, 30, 31
encoding rules 6
encoding tree 37
entropy 4
enumerative code, *see* code
enumerative encoding, *see* encoding
enumerative method 32, 39
equalization 12
error 3, 6
error propagation 28
error-correcting code, *see* code
even system, *see* system
exponential growth 15

F

factor map 20n, 22n
Feteke's Lemma 24n
Fibonacci numbers 15
Fibonacci system, *see* system
finite type 32
finite-state device 20, 25
finite-state transition diagram, *see* FSTD
finite-type constraint 13, 18
finite-type labeling 22n, 36
finite-type presentation 22, 22n, 23
finite-type shift 20
finite-type system 22, 23
Fisher cover 21, 25

fixed-length principal-state code, *see* code, principal-state
FM code, *see* code
follower set 21
Fourier analysis 4
frame 30
frame header 30
– unique 30
frequency domain constraint, *see* constraint
Frequency Modulation (FM) code, *see* code
FSTD 17, 24
full system, *see* system

G

generated (by a graph) 17
graph 17, 24
growth rate 14

H

Hamming distance 7
high-rate code, *see* code
history 37

I

incidental constraint, *see* constraint
independent path set 37
information bit, *see* bit
initial state, *see* state
interference
– intersymbol (ISI) 12
interleaving 10
internal state, *see* state
intersymbol interference (ISI), *see* interference

K

Kraft-McMillan 36

L

labeling of finite type, *see* finite-type
length (of word or pattern) 16
local requirement 39
logarithm 7
look-ahead 37
look-ahead encoder, *see* encoder
lossless 5
lossy 6

M

memory 22, 28
merging, *see* state merging
merging bit, *see* bit
merging rule 29
MFM code, *see* code
Miller code, *see* code
minimal deterministic presentation, *see* presentation
minimum distance 7
miscorrection 9
Modified FM (MFM) code, *see* code
modulation 3
modulation code, *see* code

N

noise 28
noisy channel 2
notch width 13

P

parity-check 6
parity-check bit, *see* bit
path 17
peak-detection 11
periodic (code), *see* code
periodic enumerative code, *see* code
Perron-Frobenius eigenvalue 19, 25
Perron-Frobenius theory 19, 24
phase-locked loop, *see* PLL
PLL 11
polarity bit, *see* bit
polarity-bit code, *see* code
power graph 31
power spectral density function 13
prefix code, *see* code
prefix list 35
presentation 17, 21, 22
– deterministic 21, 24
– minimal deterministic 21
presentation (of a constraint), *see* constraint

presentation of finite type, *see* finite-type
principal state, *see* state
principal-state method 26, 32, 36
product code, *see* code

Q

quantization 4
quantization noise 4

R

rate 6, 14
rate-distortion theory 6
RDS, *see* running digital sum
receiver 2, 3
redundancy 3, 4, 6, 14
Reed-Solomon code, *see* code
regular language 22, 24n
reliability information 3
reversed order 42
row code, *see* code
run-length constraint, *see* constraint
running digital sum (RDS) 13

S

sampling 4
sampling theorem 4
saturation recording 11
sender 2
sequence
– (d, k) 12
– (d, k)-RLL 12
– (d, k)-constrained 12
– code 14
– constrained 13
– $(dklr)$- 29, 32
Shannon cover 21
shift, *see under* system
shift of finite type, *see* finite-type
signal 3
size (of decoder), *see* decoder
sliding-block decoder, *see* decoder
sofic shift 25
sofic system, *see* system
source 4
source bit, *see* bit
source code, *see* code
source coding 4–6
source data 2
source symbol, *see* symbol
source-to-codeword assignment 32
spectral null constraint, *see* constraint
spectral shaping 13
state 17
– initial 17, 18, 36
– internal 25
– principal 26, 32
– sub- 38
– terminal 17, 40
state merging 21, 35
state-combination 37
state-splitting 22, 37
– ae-consistent 38
– weighted 38
storage system, *see* system
sub-state, *see* state
subset construction 22
substitution method 27, 36
subword-closed 24n
sum-variance 13
symbol
– source 25, 27, 31, 35
symbol-error 10
symbol-error-correcting code, *see* code
symbolic dynamics 22n, 25
synchronization 26, 29
synchronous 25
system 25
– code 14, 31
– constrained 14, 25
– digital communication 2
– (d, k)-constrained 17
– even 20
– Fibonacci 14, 16–18, 20, 43
– full 21
– sofic 20, 22n, 23, 24, 24n, 25
– storage 2
– transmission 2
system of finite type, *see* finite-type

T

terminal state, *see* state
time-domain constraint 13
time-slot 2
transformation (of a graph) .. 22, 23, 38
transformation (of a sequence) .3, 7, 11, 25
transformation (of an encoder) 35
transformation (of source sequence) .. 3
transition 11, 17
transmission system, *see* system
tree, *see* encoding tree
turbo code, *see* code

U

unique frame header, *see* frame header
universal 36

V

variable-length principal-state code, *see* code, principal-state
Viterbi decoder, *see* decoder

W

walk 17
weighted state-splitting, *see* state-splitting
window size 28
write equalization 12

Z

zero-disparity 33

STELLINGEN

behorende bij het proefschrift

Modulation codes

door Henk D.L. Hollmann

I

The Knuth code is not a very efficient dc-balanced modulation code.

[This thesis, Chapter II.]

II

State splitting techniques can be profitably applied to construct sliding-block decodable modulation codes with small decoding windows for finite-type constraints, at a specified rate $p : q$, provided that the maximum number of transitions leaving a state in the q-th power presentation G^q of the constraint is "small".

[This thesis, Chapter IV.]

III

Rational irreducible factors of polynomials $X^n - q$, q rational, essentially belong to one of the following three families: the cyclotomic polynomials ϕ_n (when $q = a^n$), polynomials $\theta_{n,s}$ for which $\theta_{n,s}(x)\theta_{n,s}(-x) = \phi_n(x^2/s)$ (when n is even and $q = a^n s^{n/2}$ with $s \in S(n)$), and polynomials $\psi_{n,s}$ for which $\psi_{n,s}(x)\psi_{n,s}(-x) = \phi_n(x^4/(-4s^2))$ (when $n \equiv 0 \bmod 4$ and $q = -a^n(2s)^{n/4}$ with $s \in S(n)$); here $S(n)$ is the set of square-free integers s for which $\sqrt{s}$ is contained in the nth order cyclotomic field.

[H.D.L. Hollmann, *Factorization of $x^N - q$ over* **Q**, Acta Arithmetica, vol. XLV, pp. 329–335, 1986.]

IV

The subgroup of PGL$(3, q)$, $q = 2^m \geq 8$, fixing an oval in PG$(2, q)$ is isomorphic with PSL(q). The action of this group on the exterior lines of the oval affords a (symmetric) pseudocyclic $(q/2 - 1)$-class association scheme on $q(q-1)/2$ points. For $q = 8$, this scheme together with another, related scheme are the only three-class pseudocyclic association schemes on 28 points. Combining suitable relations in this scheme for $q = 32$ produces a three-class pseudocyclic association scheme on 496 points.

[H.D.L. Hollmann, *Association schemes*, Masters thesis, Dept. of Math., Eindhoven U.T., Feb. 1982;
—, *Pseudocyclic 3-class association schemes on 28 points*, Disc. Math. 52, pp. 209–224, 1984.]

V

Wide-sense non-blocking self-routing switching networks between N senders and M receivers, constructed from switches that employ fixed-directory techniques, necessarily contain approximately NM switches; *minimal* networks can always be realized by employing switches of four special types only.

[H.D.L. Hollmann J.H. v. Lint, Jr., *Nonblocking self-routing switching networks*, Disc. Appl. Math., vol. 37/38, pp. 319–340, 1992.
L. Halpenny and C. Smyth, *Minimal nonblocking standard-path networks*, Electr. Lett. 28-12, pp. 1107–1108, 1992.
—, *A classification of minimal standard-path* 2×2 *switching networks*, J. Theor. Comp. Sc., vol. 102, pp. 329–354, 1992.]

VI

We say that an array $\boldsymbol{B}$ can be obtained from an array $\boldsymbol{A}$ of equal size by a *row-shuffle* if each row of $\boldsymbol{B}$ only contains entries from the corresponding row of $\boldsymbol{A}$; a *column-shuffle* is defined similarly. With these definitions, each array $\boldsymbol{B}$ can be obtained from a given array $\boldsymbol{A}$ of equal size that contains each entry from $\boldsymbol{B}$, by a sequence of at most five row- or column-shuffles starting with a row-shuffle; this is no longer true when “five” is replaced by “four”, as shown, e.g., by the arrays

$$\boldsymbol{A} = \begin{bmatrix} 1 & 2 & \cdots & 98 \\ 99 & 100 & \cdots & 196 \\ * & * & \cdots & * \end{bmatrix}, \quad \boldsymbol{B} = \begin{bmatrix} 1 & 1 & \cdots & 4 & 5 & 8 & \cdots & 194 \\ 2 & 2 & \cdots & 6 & 6 & 9 & \cdots & 195 \\ 3 & 4 & \cdots & 7 & 7 & 10 & \cdots & 196 \end{bmatrix}.$$

Here an $*$ indicates an empty cell; the first 35 columns of $\boldsymbol{B}$ correspond to the 35 subsets of size three from $\{1, \ldots, 7\}$.

[R.J. Sluijter, H.D.L. Hollmann, C.M. Huizer, and H. Dijkstra, *Coupling network for a data processor, including a series connection of a cross-bar switch and an array of silos*, U.S. patent 5 280 620, 1994.]

VII

We consider on-line source coding of a file of length N over an r-symbol alphabet $\{1, \ldots, r\}$ (typically, $r = 2^m$), in which letter i occurs $p_i N$ times, by means of a binary uniquely decodable code $C = \{c_1, \ldots, c_r\}$. Write d_i to denote the length of the encoding c_i of the letter i. Let $L = \sum_{i=1}^{r} p_i d_i$ denote the average length of an encoded symbol, let $H = \sum_{i=1}^{r} -p_i \log p_i$ denote the entropy of the file, and let $\delta = \sum_{\{i \mid d_i > \log r\}} p_i (d_i - \log r)$ be the *maximum data expansion* in bits per symbol. (If $r = 2^m$, then each letter in the file occupies m bits; if now all letters i with $d_i > m$ are in the initial part of the file, then on-line encoding of the file will at first result in an *expansion* of the file, where the maximal increase of the length is $N\delta$ bits, before the file starts to shrink.) With these definitions, $\delta - L + H < 1$ holds. If C is a Huffman code, then (Huffman codes are optimal) $L < H + 1$, so the above result implies that $\delta < 2$, improving the bound $\delta \leq 4$ in [1].

[[1] J-F. Cheng *et. al.*, *Data expansion with Huffman codes*, Proc. ISIT, Whistler, Canada, pp. 325, 1995.]

VIII

A nonsingular $k \times k$ matrix $\boldsymbol{A}$ over a finite field GF(q) has order at most $q^k - 1$, with equality only possible if det($\boldsymbol{A}$) is a primitive element in GF(q).

[Henk D.L. Hollmann, Solution to *Problem* 90-14*, SIAM Review 33-3, pp. 480–481, 1991.]

IX

If Hamming-space $H(n, 2)$ is partitioned into k spheres with possibly different radii, then either $k \leq 2$ or $k \geq n + 2$; the case $k = n + 2$ can hold only if one of the spheres has radius $n - 2$ and all other spheres have radius zero.

[H.D.L. Hollmann, J. Korner, and S. Lytsin, preprint.]

X

The *modified Levenshtein distance* $\delta^*(x, y)$ between words x and y is defined as $\delta^*(x, y) = \big||x| - |y|\big| + \delta(x, y)$. Here, $|x|$ and $|y|$ denote the length of the words x and y and $\delta(x, y)$ is the (ordinary) Levenshtein distance between x and y, the minimum number of insertions and deletions needed to transform x into y. A collection $\mathcal{C}$ of words is a d-deletion i-insertion correcting code if and only if $2(d + i) < \delta^*(\mathcal{C})$, where $\delta^*(\mathcal{C})$ is the minimum modified Levenshtein distance between distinct words of $\mathcal{C}$.

[H.D.L. Hollmann, *A relation between Levenshtein-type distances and insertion-and-deletion correcting capabilities of codes*, IEEE Trans. Inform. Theory 39-4, pp. 1424–1427, 1993.]

XI

1) Practical interest in binary error-correcting codes should center around the investigation of *decoding maps*, (partial) maps φ of $\{0, 1\}^n$ into itself with the property that all vectors "sufficiently close" to a fixed point c of φ are mapped to c.

2) One of the reasons for the practical success of product codes is that the product-code construction in fact delivers a decoding map for a large space from decoding maps on smaller spaces.

3) The product code $\mathcal{C}$ with constituent row code $\mathcal{C}_r \subseteq \{0, 1\}^n$ and column code $\mathcal{C}_c \subseteq \{0, 1\}^m$ consists of all $m \times n$ matrices C for which each row is a member of $\mathcal{C}_r$ and each column is a member of $\mathcal{C}_c$. Invariably, textbooks on coding theory require that both $\mathcal{C}_r$ and $\mathcal{C}_c$ are linear codes, and show that $\mathcal{C}$ is also linear, with minimum distance $\mathrm{d}(\mathcal{C}) = \mathrm{d}(\mathcal{C}_r) \times \mathrm{d}(\mathcal{C}_c)$ and size $|\mathcal{C}|$ for which $\log|\mathcal{C}| = \log|\mathcal{C}_r| \times \log|\mathcal{C}_c|$. However, the definition remains valid for arbitrary row and column codes. The expression for the minimum distance now becomes a lower bound. Moreover, the size of such "generalized product codes" can be (much) larger than in the linear case: a trivial illustration of this fact is the case where both $\mathcal{C}_r$ and $\mathcal{C}_c$ consist of all words of weight at most one. In view of the two points mentioned above, these generalized product codes merit further investigation.

XII

Let $\mathcal{B} = (B_i \mid i \in I)$ be a family of subsets of a finite set V. Write r_v to denote the number of $i \in I$ for which $v \in B_i$. If $\max_{v \in B_i} r_v \leq |B_i| - 1$ for all $i \in I$, then there exists a subset U of V for which $1 \leq |U \cap B_i| \leq |B_i| - 1$ for all $i \in I$.

XIII

We consider ordering in binary sequence spaces, as introduced in [1]. In the notation of [1] and [2], we have that $\tau_2(0, 2, 1) = (1/3)\log(2 + \sqrt{3})$ and $\tau_2(0, 2, 2) = (1/6)\log\lambda$, where λ is the largest real zero of $x^3 - 12x^2 - 4x - 1$ and the logarithm is to the base of two.

[[1] R. Ahlswede, J-P. Ye, and Z. Zhang, *Creating order in sequence spaces with simple machines*, Inform. and comp. 89, pp. 47-94, 1990.

[2] H.D.L. Hollmann and P. Vanroose, *Entropy reduction, ordering in sequence spaces, and semigroups of non-negative matrices*, Preprint "Diskrete Structuren in der Mathematik" 95-092, Universität Bielefeld, Germany, 1995.]

XIV

Computation of the inverse of an element in GF(2^{2^m}) from a given vector representation can be implemented in hardware with approximately 2^{3m} XOR- and AND-gates; in general such an implementation is to be preferred over one that involves log-tables.

[H.D.L. Hollmann, *Data processing method and apparatus for calculating a multiplicatively inverted element of a finite field*, U.S. patent 4 989 171, 1991.]

XV

The length $N = 2^n$ split radix FFT algorithm requires the lowest possible number of multiplications and possesses a regular "butterfly" structure. These and other properties range this algorithm among the best FFT methods presently available.

[P. Duhamel and H.D.L. Hollmann, *Split radix FFT algorithm*, Electr. Letters 20-1, pp. 14–16, 1984.]

XVI

The theory concerning the endgame of the game of "GO" as developed by Berlekamp is of theoretical value only.

[E.W. Berlekamp and D. Wolfe, *Mathematical go endgames: Nightmares for the professional go player*, Ishi Press International, San Jose, CA, 1994.]

XVII

Ook beginnende bridgers dienen te worden onderwezen in het gebruik van transfer biedingen na een 1SA opening van de partner.