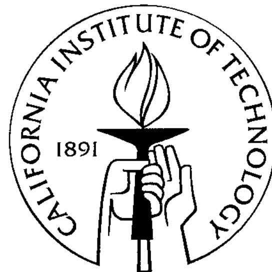
Iterative Decoding and Pseudo-Codewords

Thesis by

Gavin B. Horn

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy



California Institute of Technology
Pasadena, California

1999
(Submitted May 12, 1999)

© 1999
Gavin B. Horn
All Rights Reserved

Acknowledgements

I would like to acknowledge the many people who helped make this work possible. First, I would like to thank my thesis advisor, Professor Robert J. McEliece, for giving me the freedom to independently pursue my research, while providing constant input and advice along the way. His many suggestions and ideas have often led to exciting new ways to approach a problem. His enthusiasm in solving difficult problems has greatly influenced mine.

I am grateful to Professor Frank Kschischang for introducing me to the field of coding theory and the many hours he has spent giving me advice and counsel. His great knowledge of the field has been a tremendous resource for me.

I would like to thank Professor Padhraic Smyth for his help in understanding message passing on graphs from an artificial intelligence perspective. I am also grateful to Professor Henk van Tilborg for our many discussions on counting pseudo-codewords and particularly for his suggestion to use state diagrams as a unifying approach to solve the counting problem in Chapter 5. Professor Michael Tanner has been an invaluable source of interesting problems and insight into what makes iterative decoding work so well.

I would like to thank Professor P. P. Vaidyanathan and Dr. Dariush Divsalar for their time spent on my defense committee and their advice.

I would like to thank the Natural Sciences and Engineering Research Council and the Communications Research Centre in Canada for the generous funding they have provided throughout the course of my degree.

I am very grateful to Ray and Lilian Porter, Linda Dozsa and all the department secretaries for looking after me on and off the field. I thank the players on the Caltech club soccer and rugby teams for their commitment and spirit over the last four years.

I would like to thank Jung Fu Cheng for giving me his computer code, which made all my simulations easier to write. I am also grateful for the many discussions

I have had with my current and former group mates: Srinivas Aji, Hui Jin, Aamod Khandekar, Wei Lin, Meina Xu, Zhong Yu, Mohamed-Slim Alouini, Neelesh Mehta, Lifang Li, Sriram Vishwanath and Syed Ali Jafar.

I am grateful to Bob Freeman and Joseph Chiu for keeping the computers running smoothly and their technical assistance.

Lastly, I would like to thank all my friends and family. To Dalya, Jo-Ann, David, Rosalie, Mullie and Kim, for being my family away from home, and to my parents for giving me the freedom and encouragement to do what I wanted, and my sisters, aunt and grandmother for putting up with me while I did. This thesis is partly theirs.

Abstract

In the last six years, we have witnessed an explosion of interest in the coding theory community, in iterative decoding and graphical models, due primarily to the invention of turbo codes. While the structural properties of turbo codes and low density parity check codes have now been put on a firm theoretical footing, what is still lacking is a satisfactory theoretical explanation as to why iterative decoding algorithms perform as well as they do. In this thesis we make a first step by discussing the behavior of various iterative decoders for the graphs of tail–biting codes and cycle codes. By increasing our understanding of the behavior of the iterative min–sum (MSA) and sum–product (SPA) algorithms on graphs with cycles, we can design codes which achieve better performance.

Much of this thesis is devoted to the analysis of the performance of the MSA and SPA on the graphs for tail–biting codes and cycle codes. We give sufficient conditions for the MSA to converge to the maximum likelihood codeword after a finite number of iterations. We also use the familiar union bound argument to characterize the performance of the MSA after many iterations. For a cycle code, we show that the performance of the MSA decoder is asymptotically as good as maximum likelihood. For tail–biting codes this will depend on our choice of trellis.

Contents

Acknowledgements iii

Abstract v

1 Introduction 1

1.1 Graphical Models and Codes 1

1.2 Thesis Outline 2

2 Graphical Models and Message Passing 4

2.1 Junction Graphs 6

2.2 Tanner Graphs 11

2.3 Message Passing on Junction Graphs 14

2.4 Message Passing on Tanner Graphs and Computation Trees 17

2.5 The Choice of Graphical Model and Decoding Algorithm 19

3 The Iterative Sum–Product Decoding of Tail–Biting Codes 20

3.1 Message Passing Schedule 21

3.2 Message Passing Convergence 22

3.3 The Objective Function vs the Final State of an Edge 26

3.4 Interpretation of the Elements of M_1 27

3.5 Binary Valued Hidden Variables 29

3.6 Non–Binary Valued Hidden Variables 31

3.7 Conclusions 39

4 The Iterative Min–Sum Decoding of Tail–Biting Codes 41

4.1 The Iterative Min–Sum Algorithm on Single Cycle Junction Graphs . 42

4.2 The Iterative Min–Sum Algorithm on a Tail–Biting Trellis 46

4.3 Conclusions 68

5 Counting Pseudo–Codewords for Tail–Biting Convolutional Codes 69

5.1 The Transfer Matrix Method 69

5.2 Convolutional Codes, Tail-biting Trellises and Pseudo–Cycles 71

5.3 Pseudo–Codeword State Diagrams 73

5.4 Examples of Counting Pseudo–Codewords for Small m and l 78

5.5 Conclusions 89

6 The Iterative Decoding of Cycle Codes 91

6.1 Basic Notation and Definitions 92

6.2 Iterative Decoding on the BSC 96

6.3 Min–Sum Decoding over an AWGN Channel 102

6.4 Conclusions 113

7 Conclusions 114

A Proofs and Derivations 118

A.1 Proof of Theorem 3.6.1 118

A.2 Proofs for Section 4.2 119

A.3 Proofs for Chapter 5 123

A.4 Proofs for Section 6.3 124

B Definition of a Commutative Semi–ring 131

Bibliography 132

List of Figures

2.1 The junction graph for the (8,4,4) tail–biting extended Hamming code. 10

2.2 An alternative junction graph for the (8,4,4) tail–biting extended Hamming code. 11

2.3 The Tanner graph for a (5,3,2) cycle code. 13

2.4 A junction tree with edges directed towards v_4. 14

2.5 The computation tree for the (5,3,2) cycle code in Figure 2.3 after 2 iterations of message passing. 18

3.1 (a) The original junction graph and (b) the junction graph after the first stage of message passing has completed. 22

3.2 The vertices v_{i-1}, v_i and v_{i+1} that form a part of the cycle $v_1 v_2 \dots v_l$ in V'. 23

3.3 (a) The original junction graph with a single cycle and (b) the graph that results when the vertex v_1 has been split to form a chain with an additional copy of the edge e_1. 28

3.4 (a) A simple belief network consisting of 5 nodes and 1 loop and a corresponding (b) junction graph shown after the first stage of message passing has completed. 32

3.5 The decision regions for the hidden variable x where $p = 0.25$ and $\sigma^2 = 0.2$ using (a) the SPA and (b) the exact probabilities from the objective function. The black, grey and white regions correspond to deciding $x = 0$, 1 and 2 respectively. Plot (c) shows the regions where the SPA and the objective function make different decisions and plot (d) the $P(E)$ vs channel crossover probability p. 33

3.6 Plot of the state of x vs the objective function of x for $x = 0$ for 5000 trials for the network in Figure 3.4. 34

3.7 Plots of (a) how for a given S and N, the state probability vector varies as a is varied from 1 to 0 and (b) the SPA probability vs the optimal probability, computed for $x = 0$, for 5000 trials in which we have picked the off–diagonal elements uniformly between 0 and 1. . . 36

3.8 Plot of P(C) for the SPA vs the mean DODR for different distributions of the off–diagonal elements for various distributions of the off–diagonal elements. 37

3.9 Plot of the P(C) vs the variance of the off–diagonal elements where the mean DODR is held constant. 38

3.10 Plot of (a) how for a given S and $N = J - I$, the state probability vector varies as a is varied from 0 to 1 and (b) the SPA probability vs the optimal probability m, computed for for the matrix M in (3.12) for various values of a. 39

4.1 A digraph. 47

4.2 A tail–biting trellis of length 3. 51

4.3 A rate 1/2, constraint length 2 convolutional encoder. 54

4.4 The (a) state diagram and (b) trellis section for the convolutional encoder with input and output weights on the edges. 55

4.5 A 5–section tail–biting trellis for the encoder in Figure 4.3. 56

4.6 Plots of the performance of the $(2N, N)$ tail–biting code based on the convolutional encoder in Figure 4.3 for various N, including the union bounds from (4.29) and (4.30). 65

4.7 The performance of the MSA decoder for the (24,12,8) tail–biting extended Golay code on an AWGN channel with a maximum of 30 iterations. 67

5.1 A constraint length m convolutional encoder with feedback (shown without any output taps). 72

5.2 (a) The pseudo–codeword $\mathbf{s}$ with $l = 4$ and $m = 2$ written in a ring. (b) The initial contents of the 4 windows and the contents after the windows have been rotated clockwise by N bits. 75

5.3 The state diagram for a device which generates any string that contains no substring of m or more zeros. 79

5.4 (a) The pseudo–codeword state diagram for pseudo–codewords with the first N bits zero and (b) the reduced pseudo–codeword state diagram, for $m = 2$ and $l = 3$. 82

5.5 The reduced pseudo–codeword state diagram for $m = 2$ and $l = 4$. . . 85

5.6 The reduced pseudo–codeword state diagram for $m = 3$ and $l = 3$. . . 89

6.1 (a) The graph G and (b) the Tanner graph G_T, for a (9,4,3) cycle code. 93

6.2 The performance of the 3 message passing algorithms on both regular and random graphs for a code rate of 1/2 and a maximum of 150 iterations. 100

6.3 Plot (a) shows the median number of iterations for the SPA to successfully converge to a codeword. The bars show the 25th and 75th percentiles. Plot (b) shows the probability of convergence to a valid codeword. Plots (c) and (d) show an empirical distribution of the number of iterations before convergence for a channel crossover probability of 0.01 and 0.045 respectively. 101

6.4 The graph of a (5,3,2) cycle code with a good pseudo-cycle illustrated by the dashed line. 107

6.5 The Tanner graph for an (8,3,3) cycle code with a bad pseudo–cycle illustrated by the dashed line. 108

7.1 The performance of the SPA decoder for the (15,5,6) code on an AWGN channel with a maximum of 25 iterations. 116

List of Tables

4.1 The pseudo-weights for the (10,5,3) tail–biting code with $m = 2$. 64

5.1 The number of pseudo–codewords $w_{N,2}$ for various m. 81

5.2 The number of pseudo–codewords $w_{N,l}$ for $m = 2$. 86

Chapter 1

Introduction

In the last six years, we have witnessed an explosion of interest in the coding theory community, in iterative decoding and graphical models. The spark was ignited, in 1993, when Berrou, Glavieux, and Thitimajshima [1] introduced to the world turbo codes, whose performance approached Shannon's theoretical limit. Since then there has also been a renewed interest in Gallager's low density parity check codes [2] and a number of other parallel and serial concatenated coding schemes [3, 4].

While the structural properties of turbo codes [5, 6, 7, 8, 9] and low density parity check codes [10, 11, 12] have now been put on a firm theoretical footing, what is still lacking is a satisfactory theoretical explanation as to why iterative decoding algorithms perform as well as they do. In this thesis we make a first step by discussing the behavior of various iterative decoders for the graphs of tail–biting codes and cycle codes. This work is intended as a sequel to the thesis of Wiberg [13] in which he presents a general framework for iterative decoding on graphs.

1.1 Graphical Models and Codes

There are many ways to represent a given code with a graph. For any such representation, we will consider the class of iterative decoding algorithms that compute an internal state at a vertex, which is based on the local information at that vertex and any information received from adjacent vertices in the graph. The decoder's performance and complexity will depend both on the graph and the decoding algorithm we choose. Ideally we would like to be able to select a graph for the code on which an efficient decoding algorithm can achieve optimal decoding performance. However, for all known codes, algorithms based on these graphs quickly become intractable,

as the code approaches the Shannon limit. Thus, if we want to achieve really good performance, we are forced to choose a graph on which we cannot perform optimal decoding, but for which the subsequent increase in strength of the code is greater than the loss we sustain due to sub–optimal decoding. This is the case for both turbo codes and low density parity check codes.

Much of the research for improving the performance of turbo codes has focussed on designing better interleavers [3, 5], or better component codes [4, 9], to increase the strength of the code. On the other hand, low density parity check codes are already known to be good codes [10], so there the search is to find representations on which the iterative decoder works well [11, 12, 14, 15]. If we can better understand what causes the iterative decoder to make sub-optimal decisions, then we will be able to design codes which are both good codes and have graphical representations for which the iterative decoder works well.

1.2 Thesis Outline

In Chapter 2, we will present the two graphical models, namely junction graphs [16, 17] and Tanner graphs [13, 18], that we will consider in this thesis. We also present the iterative min–sum and sum–product algorithms for these models [13, 16].

In Chapters 3 and 4, we will analyze the performance of the iterative sum–product and min–sum algorithms respectively, for a junction graph with a single cycle. This work is directly relevant to the study of iterative decoding of tail–biting codes, since their underlying graph has just one cycle. In the case of the iterative min–sum algorithm, we will use a union bound argument to quantify the loss in performance due to the sub-optimality of the decoder. Much of this work was presented earlier in [19] and [20], for the iterative sum–product algorithm, and in [21] and [22], for the iterative min–sum algorithm. In Chapter 5, we continue with our analysis of iterative decoding of tail–biting codes, by counting the number of closed paths in a tail–biting trellis, which never visit the same state twice. These paths, which we call pseudo–codewords, are the cause of the degradation in both algorithms' performance.

In Chapter 6, we will analyze various decoding algorithms on the Tanner graph for a cycle code. We will show that for the additive white Gaussian noise (AWGN) channel, the iterative min–sum decoder performs asymptotically as well as an optimal decoder in minimizing the word error rate. Part of this work was presented previously in [23].

Finally, in Chapter 7, we present our conclusions and a simple experiment to gain some insight into understanding the iterative decoding of low density parity check codes and turbo codes.

Chapter 2

Graphical Models and Message Passing

A number of graphical models and message passing algorithms have been proposed for use as decoders over the years. They have proven to be not only an efficient means of decoding complex codes, but also a useful tool to describe classes of codes. Gallager's 1962 low density parity check (LDPC) codes [24, 2] were the first instance of decoding using message passing on a graph. Since then Viterbi's algorithm [25] and the "forward–backward" algorithm [26] (also known as the Baum–Welch [27] and BCJR [28] algorithms) have been applied to the trellis of a code to minimize word and bit error probability respectively. More recently Tanner [18] generalized Gallager's work in proposing a method of constructing codes based on bipartite graphs.

Currently there is great interest in a variety of graphical models in the coding community. These models include the Bayesian networks of Pearl [29, 30, 31], junction graphs [16, 17], factor graphs [32] and Tanner graphs which have been modified by Wiberg to include state nodes [13]. In this thesis we will make use of junction graphs and Tanner graphs to analyze codes.

There is also an equally large variety of message passing algorithms that have been used for decoding. Among these are Pearl's "belief propagation algorithm" [31], the Shafer–Shenoy probability propagation algorithm [33] and the generalized distributive law (GDL) [16]. Many of these algorithms and graphical models are equivalent to one another and the choice one makes seems to be more of a personal preference, rather than a difference in the decoding performance of the algorithm, or the descriptive power of the graphical model.

For instance, the sum–product algorithm of Wiberg [13] is simply a generalization of the BCJR algorithm [28], the turbo decoding algorithm [1] and the decoding algorithm used by Gallager for low density parity check codes [2]. It will have identical

behavior and performance on the Tanner–Wiberg–Loeliger graph of a code, as the belief propagation algorithm will have on the equivalent Bayesian network for that code, assuming of course that the same message passing schedule is used for both.

A graphical model can represent, among other things, a global probability distribution as a factoring of local distributions by edge connections and vertices in a graph. It is well known that message passing algorithms converge to the correct a posteriori probability (APP) when this underlying graph is cycle free. When the graph is multiply connected, message passing may not converge and if it does, the APP's are generally incorrect. However, when decoding, we are usually only concerned with finding the maximum a posteriori (MAP) or maximum likelihood (ML) decision for a codeword bit, so only the ordering of the APP values is important.

There is a large body of empirical evidence that suggests that message passing has good performance in the presence of cycles [1, 5, 34, 10, 9]. In fact, recently, Luby et al. [11] and Richardson and Urbanke [12] have extended Gallager's work on LDPC codes to show one can achieve an arbitrarily small error probability for a given rate, even in the presence of cycles. Thus it is important to study the "approximate correctness" of various message passing algorithms on graphs with cycles in order to gain an understanding of why these algorithms perform so well.

In this thesis we will discuss the behavior of message passing algorithms on two classes of codes, namely tail–biting codes, whose underlying graphs contain a single cycle [35, 36], and cycle codes [37].

In Section 2.1, we will introduce the notation of Aji and McEliece to describe a junction graph and give two examples of junction graphs for tail–biting codes. In Section 2.2, we will introduce Tanner graphs and give an example of the Tanner graph for a cycle code. We will also show how to construct a junction graph from the Tanner graph for a code.

In Section 2.3, we will present a general class of message passing algorithms on a junction graph called the generalized distributive law (GDL) [16]. In Section 2.4 we will construct the computation tree for a given message passing schedule on a Tanner graph [13]. Finally in Section 2.5, we will present some concluding remarks on the

various graphical models we have chosen.

2.1 Junction Graphs

In this section we introduce the notation of Aji and McEliece [16] to describe a junction graph. We also present two examples of a junction graph for a tail–biting code.

Let $x_1, \ldots, x_n$ be variables taking values from the finite sets, $A_1, \ldots, A_n$ respectively. Let $S = \{i_1, \ldots, i_r\}$ be a subset of $\{1, \ldots, n\}$, such that x_S represents the variable list $\{x_{i_1}, \ldots, x_{i_r}\}$ and A_S represents the product $A_{i_1} \times \cdots \times A_{i_r}$. The variable list $\{x_1, \ldots, x_n\}$ will be denoted by $\mathbf{x}$.

Now we define $\mathcal{S} = \{S_1, \ldots, S_M\}$ to be M subsets of $\{1, \ldots, n\}$, which we shall call *local domains*. For each local domain $S_i \in \mathcal{S}$, we define a function called the *local kernel*, $\alpha_i : A_{S_i} \rightarrow R$, where R is a commutative semiring with 2 binary operators "+" and "·", referred to as addition and multiplication. (See Appendix B for a complete definition of a commutative semiring). The global kernel β is now defined as the product of the local kernels,

$$\beta(x_1, \ldots, x_n) = \prod_{i=1}^{M} \alpha_i(x_{S_i}). \tag{2.1}$$

Now, define a set of vertices $V_J = \{v_1, \ldots, v_M\}$, where each $v_i \in V_J$ corresponds to a local domain, $S_i \in \mathcal{S}$, for $i = 1, \ldots, M$. We can connect the vertices of V_J together to form a *junction graph*, $G_J = (V_J, E_J)$, where $E_J = \{e_1, \ldots, e_N\}$ is the set of edges in the graph with $|E_J| = N$. A junction graph, is any graph, such that for any $x_j \in \mathbf{x}$, the induced subgraph, formed by taking all vertices $v_i \in V_J$ where $x_j \in x_{S_i}$, is connected. The junction graph for a given $\mathcal{S}$ is in general not unique. In fact, it may not even exist. We refer the reader to Aji and McEliece [16] and Jensen [17] for further details on how to determine the existence and construct the lowest complexity, junction graph for a given set of local domains.

We define $\mathcal{E} = \{E_1, \ldots, E_N\}$ to be N subsets of $\{1, \ldots, n\}$, which we shall call

edge domains. The edge domain E_i, is determined by the edge e_i, in the junction graph for $i = 1, \ldots, N$. If $e_i = (v_j, v_k)$, then $E_i = (S_j \cap S_k)$.

We would like to compute the marginalization of the global kernel over some subset of the local domains in $\mathcal{S}$. We shall define the S_i*–marginalization* of β, over the local domain S_i as

$$\beta_i(x_{S_i}) = \sum_{x \backslash x_{S_i}} \beta(\mathbf{x}). \tag{2.2}$$

We would also like to compute the marginalization of the global kernel over some subset of the edge domains in $\mathcal{E}$. We shall define the E_i*–marginalization* of β, over the edge domain E_i as

$$\theta_i(x_{E_i}) = \sum_{x \backslash x_{E_i}} \beta(\mathbf{x}). \tag{2.3}$$

We will also call the marginalizations $\beta_i(x_{S_i})$ or $\theta_i(x_{E_i})$, the β_i or θ_i *objective functions.*

2.1.1 Junction graphs for tail–biting codes

Consider an (n, k, d) binary linear code with information vector $\mathbf{u} = (u_1, u_2, \ldots, u_k)$ encoded by a generator matrix G to form the codeword vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. The codeword $\mathbf{x}$ is then transmitted across a discrete memoryless channel and received as the vector $\mathbf{y} = (y_1, y_2, \ldots, y_n)$. We would like to minimize the bit error probability at the receiver by finding the string of bits $\hat{\mathbf{u}} = (\hat{u}_1, \hat{u}_2, \ldots, \hat{u}_k)$ such that

$$\hat{u}_i = \arg\max_{u_i} p(u_i|\mathbf{y}), \tag{2.4}$$

for $i = 1 \ldots k$.

The "likelihood" of a particular codeword $(x_1, x_2, \ldots, x_n)$ is

$$p(y_1, y_2, \ldots, y_n | x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} p(y_i|x_i) \tag{2.5}$$

where $p(y|x)$ are the transition probabilities of the channel. For each column of the generator matrix we define a local domain with local kernel χ_i, for $i = 1 \ldots n$. Here

χ_i is a function of the codeword bit x_i and the information bits that contain column i in their span[1]. If column i is in the span of $(u_1, \ldots, u_j)$, then χ_i will be the function

$$\chi_i(x_i, u_1, \ldots, u_j) = \begin{cases} 1 & \text{if } x_i + g_{1,i}u_1 + \cdots g_{j,i}u_j = 0 \\ 0 & \text{if } x_i + g_{1,i}u_1 + \cdots g_{j,i}u_j = 1 \end{cases}$$

where $g_{l,i}$ is the (l,i)–entry of the generator matrix.

Example 2.1.1 (Calderbank et al. [35]) Consider the (8,4,4) extended Hamming code with generator matrix

$$G = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{matrix} \\ \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array}. \tag{2.6}$$

The spans of u_1, u_2, u_3 and u_4 are $[1,4]$, $[3,7]$, $[5,8]$ and $[7,3]$ respectively. We will have a local domain for each information and codeword bit and one for each column of the generator matrix.

---

[1]The *span* of a row of a generator matrix is defined as the interval $[j, j']$ which includes all the non–zero components of that row. For a tail–biting code we may have $j > j'$ since the index arithmetic is modulo n. For more details on how to construct a tail–biting trellis from a generator matrix, we refer the reader to Calderbank et al. [35].

	local domain	local kernel
1	$\{u_1\}$	1
$\vdots$	$\vdots$	$\vdots$
4	$\{u_4\}$	1
5	$\{x_1\}$	$p(y_1 \| x_1)$
$\vdots$	$\vdots$	$\vdots$
12	$\{x_8\}$	$p(y_8 \| x_8)$
13	$\{x_1, u_1, u_4\}$	$\chi_1(x_1, u_1, u_4)$
$\vdots$	$\vdots$	$\vdots$
20	$\{x_8, u_3, u_4\}$	$\chi_8(x_8, u_3, u_4)$

Here a local kernel of 1 implies that the local kernel is a constant for all possible variable assignments in the local domain.

The global kernel is then

$$\beta(u_1, \ldots, u_4, x_1, \ldots, x_8) = \begin{cases} p(y_1, \ldots, y_8 | x_1, \ldots, x_8) & \text{if } uG = x \\ 0 & \text{otherwise} \end{cases}$$

and a possible junction graph for the tail–biting code is shown in Figure 2.1. Note that the subgraph induced by each variable is connected.

It follows that the objective function for the local domain $\{u_i\}$ is

$$\beta_i(a_i) = \sum_{\mathbf{x}: u_i = a_i} p(y_1, \ldots, y_8 | x_1, \ldots, x_8),$$

where the sum is over all codewords $\mathbf{x}$, encoded by $u_i = a_i$. If we assume a uniform distribution on the input bits, then β_i is the APP of the bit u_i given the received vector $\mathbf{y}$.

The junction graph for a given generator matrix can have many single cycle representations.

Example 2.1.2 (Calderbank et al. [35]) Consider the generator matrix in (2.6) and define a local domain for the pairs of columns $[1, 2]$, $[3, 4], \ldots, [7, 8]$ as (x_1, x_2),

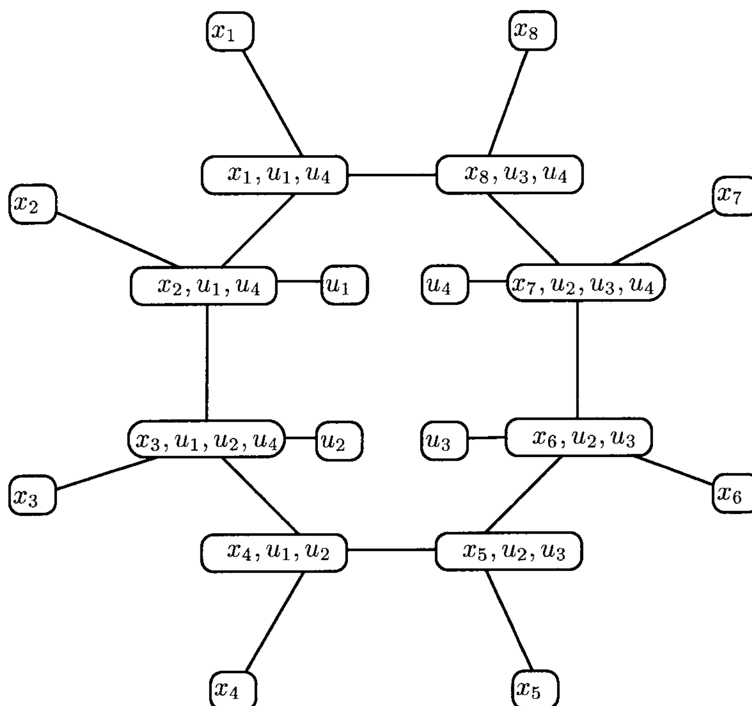


Figure 2.1: The junction graph for the (8,4,4) tail–biting extended Hamming code.

$(x_3, x_4), \ldots, (x_7, x_8)$ respectively. The spans of u_1, u_2, u_3 and u_4 become $[1, 4]$, $[3, 8]$, $[5, 8]$ and $[7, 4]$ respectively. We now have a local domain for each information bit and pair of codeword bits, as well as one for each pair of columns of the generator matrix.

	local domain	local kernel
1	$\{u_1\}$	1
$\vdots$	$\vdots$	$\vdots$
4	$\{u_4\}$	1
5	$\{x_1, x_2\}$	$p(y_1\|x_1)p(y_2\|x_2)$
$\vdots$	$\vdots$	$\vdots$
8	$\{x_7, x_8\}$	$p(y_7\|x_7)p(y_8\|x_8)$
9	$\{x_1, x_2, u_1, u_4\}$	$\chi_1(x_1, x_2, u_1, u_4)$
$\vdots$	$\vdots$	$\vdots$
12	$\{x_8, u_3, u_4\}$	$\chi_4(x_7, x_8, u_2, u_3, u_4)$

If columns $2i-1$ and $2i$ are in the span of $(u_1, \ldots, u_j)$, then the local kernel χ_i, is

now defined to be the function

$$\chi_i(x_{2i-1}, x_{2i}, u_1, \ldots, u_j) = \begin{cases} 1 & \text{if } x_{2i-1} + g_{1,2i-1}u_1 + \cdots g_{j,2i-1}u_j = 0 \\ & \text{and } x_{2i} + g_{1,2i}u_1 + \cdots g_{j,2i}u_j = 0 \\ 0 & \text{otherwise} \end{cases}$$

and the global kernel and objective functions remain the same. A possible junction graph for this set of local domains is shown in Figure 2.2.

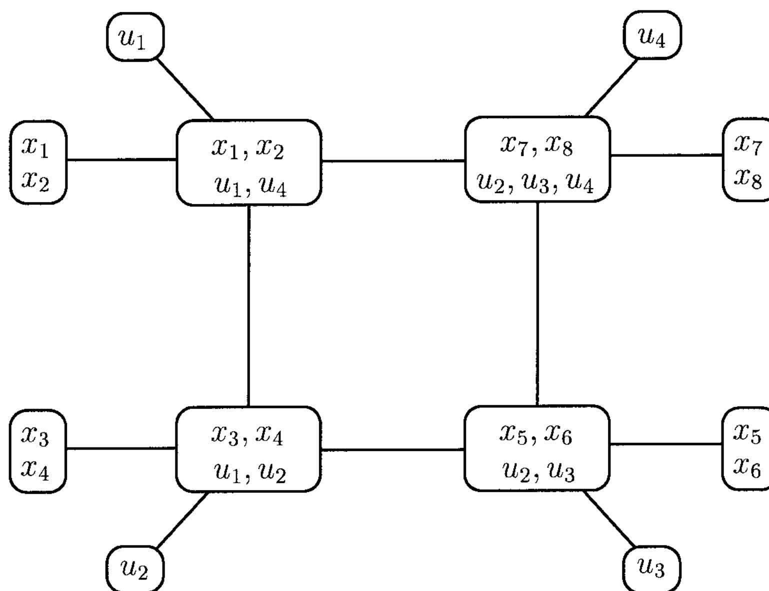


Figure 2.2: An alternative junction graph for the (8,4,4) tail–biting extended Hamming code.

2.2 Tanner Graphs

A Tanner graph [13, 18], $G_T = (V_T, E_T)$, is a bipartite graph, consisting of a set of codeword nodes, V_c and check nodes V_p, where $V_T = V_c \cup V_p$, with $V_c \cap V_p = \emptyset$ and $E_T \subseteq V_c \times V_p$. Define the cardinality of $|V_p| = m$ and $|V_c| = n$. We will generally label the codeword nodes $v_1, \ldots, v_n$ and the check nodes $c_1, \ldots, c_m$, so the edges will be of the form (v_i, c_j), for $i \in [n]$ and $j \in [m]$, where $[n]$ denotes the set of integers from $1, 2, \ldots, n$.

We will represent a codeword by a binary n–tuple whose ith component is 1 (or 0) depending on whether the vertex v_i is assigned a 1 (or 0) in the Tanner graph. For a binary linear code, each row of the parity check matrix H defines a parity check that this binary n–tuple must satisfy in order to be a valid codeword. These checks are called local constraints since each check only affects some subset of the codeword nodes. The rows of H thus define a check structure for the code, and determine which binary n–tuples are valid codewords.

Consider an (n, k, d) code with parity check matrix H. We can construct the Tanner graph for the code by letting each row of H correspond to a check node in V_p, and letting each column correspond to a codeword node in V_c. An edge occurs between vertex $v_j \in V_c$ and vertex $c_i \in V_p$ if and only if the (i, j)–entry of H is a one. If we assign a 0 or a 1 to each codeword node, then a valid codeword is one that satisfies all of its local parity checks.

The more general case of arbitrary constraints for V_p, is found in Wiberg's thesis [13, Chapter 2] and in Tanner [18]. There a check node may represent more than just a simple linear equation and the set of check nodes V_p defines an "equation system." However, for our purposes we consider the check nodes to be simply parity check constraints for the code.

It is easy to construct a junction graph from a Tanner graph. First we define a set of variables $x_1, \ldots, x_n$, with variable x_i corresponding to the codeword node v_i, for $i \in [n]$. We then define local domains $\{x_i\}$, for all $i \in [n]$ which are equivalent to the codeword nodes in the Tanner graph. Next we define a local domain for each check node where the set of variables in each local domain is the set of variables corresponding to codeword nodes connected to that check node in the Tanner graph. An edge occurs in the junction graph if there is an edge between the equivalent vertices in the Tanner graph. The resulting junction graph is bipartite and isomorphic to the original Tanner graph. The local kernels in the junction graph are determined by the local cost functions [13] and the check structure of the Tanner graph.

2.2.1 Tanner graphs for cycle codes

A (j, r) regular LDPC code has a parity check matrix for which there are j ones per column and r ones per row. The corresponding Tanner graph will have codeword nodes of degree j and check nodes of degree r. For a cycle code, we have $j = 2$. In this thesis, we will also consider irregular cycle codes, defined as LDPC codes for which there is an arbitrary number of ones per row.

If the Tanner graph is connected, then an (n, k, d) cycle code has: n codeword nodes; dimension $k = |V_c| - |V_p| + 1$; and a minimum distance d, equal to the number of codeword nodes in the shortest cycle in the graph [37].

Example 2.2.1 Consider the (5,3,2) cycle code with parity check matrix

$$H = \begin{array}{c} \\ c_1 \\ c_2 \\ c_3 \end{array} \begin{array}{c} \begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array}. \tag{2.7}$$

Note that the last row of the matrix is the modulo 2 sum of the first 2 rows and so is redundant. The Tanner graph is shown in Figure 2.3. The codeword nodes V_c correspond to the codeword bits and are represented by the circles labelled $v_1, \ldots, v_5$. The check nodes V_p, correspond to the rows of the parity check matrix and are represented by a black dot for each row of the matrix. The code has dimension $|V_c| - |V_p| + 1 = 3$ and minimum distance $d = 2$.

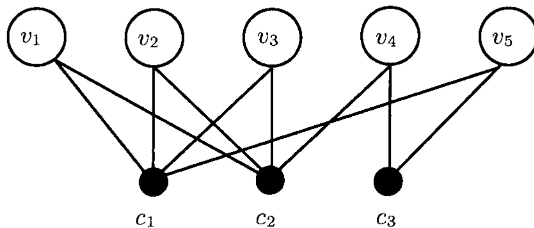


Figure 2.3: The Tanner graph for a (5,3,2) cycle code.

2.3 Message Passing on Junction Graphs

For any junction graph we can pass "messages" along an edge between adjacent vertices. Consider two vertices $v_i, v_j \in V_J$, such that $(v_i, v_j) \in E_J$. The "message" passed from v_i to v_j is a table containing the values of a function $\mu_{i,j} : A_{S_i \cap S_j} \to R$. We define the *state* of the vertex v_i to be a table containing the values of a function $\sigma_i : A_{S_i} \to R$. Each table contains $|A_{S_i}| = |A_{i_1}| \times \cdots \times |A_{i_r}|$ entries. Finally, we define the *state* of the edge e_i to be a table containing the values of a function $\rho_i : A_{E_i} \to R$. We will use the following rules to update $\mu_{i,j}$, σ_i and ρ_i from [16]:

$$\mu_{i,j}(x_{S_i \cap S_j}) = \gamma \sum_{x_{S_i \setminus S_j}} \alpha_i(x_{S_i}) \prod_{\substack{(v_i,v_k) \in E_J \\ k \neq j}} \mu_{k,i}(x_{S_k \cap S_i}), \tag{2.8}$$

$$\sigma_i(x_{S_i}) = \gamma \; \alpha_i(x_{S_i}) \prod_{(v_k,v_i) \in E_J} \mu_{k,i}(x_{S_k \cap S_i}) \tag{2.9}$$

and

$$\rho_i(x_{E_i}) = \gamma \; \mu_{k,j}\mu_{j,k} \tag{2.10}$$

where $e_i = (v_j, v_k)$ and the γ in each case is a normalization constant. When the junction graph is a tree, message passing can be used to compute β_i (or θ_i) at any desired vertex (or edge) using the appropriate scheduling [16, 17, 33]. The objective function β_i (or θ_i) is the state of the vertex v_i (or edge e_i), after the message passing terminates.

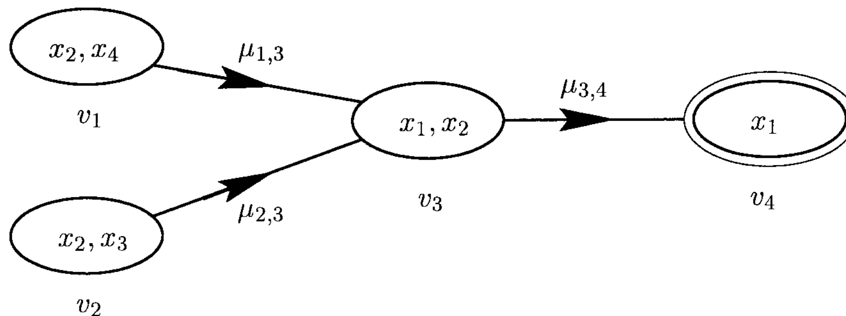


Figure 2.4: A junction tree with edges directed towards v_4.

Example 2.3.1 Consider the junction tree with 4 vertices and 4 random variables shown in Figure 2.4. We would like to compute the objective function at v_4 (shown in double lines),

$$\beta_4(x_1) = \sum_{x_2,x_3,x_4} \alpha_1(x_2,x_4)\alpha_2(x_2,x_3)\alpha_3(x_1,x_2)\alpha_4(x_1) \tag{2.11}$$

We begin by directing each edge in the tree on its unique path towards v_4. Messages are sent only in the direction of the edges. A vertex will only send a message to an outgoing neighboring vertex, when for the first time it has received messages from each of its incoming neighbors. With this scheduling, message passing begins at the leaves and proceeds until we reach v_4.

For example, v_1 and v_2 will initially send the messages

$$\begin{aligned} \mu_{1,3}(x_2) &= \sum_{x_4} \alpha_1(x_2,x_4) \\ \text{and} \quad \mu_{2,3}(x_2) &= \sum_{x_3} \alpha_2(x_2,x_3) \end{aligned}$$

respectively to v_3. Once v_3 has received both messages, it will send the message

$$\mu_{3,4}(x_1) = \sum_{x_2} \alpha_3(x_1,x_2)\mu_{1,3}(x_2)\mu_{2,3}(x_2)$$

to v_4. The objective function β_4 is simply the state of v_4 which is

$$\begin{aligned} \sigma_4(x_1) &= \alpha_4(x_1)\mu_{3,4}(x_1) \\ &= \alpha_4(x_1)\sum_{x_2} \alpha_3(x_1,x_2)\mu_{1,3}(x_2)\mu_{2,3}(x_2) \\ &= \alpha_4(x_1)\sum_{x_2} \alpha_3(x_1,x_2)\left(\sum_{x_4} \alpha_1(x_2,x_4)\right)\left(\sum_{x_3} \alpha_2(x_2,x_3)\right) \\ &= \beta_4(x_1). \end{aligned}$$

If we define $e_1 = (v_3, v_4)$, then the objective function θ_1 is simply

$$\begin{aligned}\beta_1(x_1) &= \mu_{3,4}(x_1)\alpha_4(x_1) \\ &= \sigma_4(x_1).\end{aligned}$$

For more details on message passing in junction trees, as well as a proof that the objective function at vertex v equals the state of v after message passing has terminated, see Jensen [17] and Aji and McEliece [16].

For a junction graph that is not a tree, message passing cannot in general compute the objective function when the above message and state computation rules are used, regardless of the message passing schedule. The main focus of this thesis will be the investigation of the relationship between the objective function and the vertex or edge state when we perform message passing on a junction graph that contains one or more cycles.

2.3.1 The iterative min–sum and sum–product algorithms

We will focus on two main classes of message passing algorithms, the iterative min–sum and sum–product algorithms. For a junction graph, the class of algorithm is determined by which semiring R, we choose to define our local kernels over.

The sum–product semiring has $R = \mathbb{R}^{\geq 0}$ and uses ordinary addition and multiplication as its binary operators. The min–sum semiring has $R = \mathbb{R} \cup \infty$ where $x + y = \min(x, y)$ with identity element ∞, so $x \cdot \infty = \infty$, and the operator "$\cdot$" is ordinary addition with identity 0 [16].

These algorithms will not actually make decisions on the received codeword, rather they will be used to compute a set of states from which we can make a final decision. The type of decision will depend on our choice of semiring. For instance on the min–sum semiring, we can initialize the local kernels of the local domains for the codeword bits x_i, with the channel log–likelihoods $p(y_i|x_i)$ for an AWGN channel, or the Hamming distance for the BSC channel. For a cycle free graph the min–sum algorithm then can be used to find the ML codeword. For a cycle free graph, the

sum–product algorithm can compute the per symbol APP conditioned on the received $\mathbf{y}$, and thus find the MAP decision for each codeword bit.

2.4 Message Passing on Tanner Graphs and Computation Trees

We will not redefine the iterative min–sum and sum–product algorithms in terms of a Tanner graph, but refer the reader to Wiberg's thesis [13, Chapter 3] for further details. We will, instead, construct the junction graph from the Tanner graph and use the message passing algorithms from the previous section to decode.

Our main concern is to analyze a message passing algorithm from the perspective of a single codeword node v_i, in the Tanner graph for a cycle code. In order to do this we first have to define a message passing schedule.

Our message passing schedule proceeds as follows. Each codeword node v_i is initialized with the channel output y_i, for $i \in [n]$. One iteration of message passing consists of two steps. First, all the codeword nodes send messages out on all their edges, to the parity check nodes. For example, in Figure 2.3, messages are passed from all the top nodes $v_1, \ldots, v_5$, to the bottom nodes $c_1, \ldots, c_3$, along every edge. In the second step, the check nodes send a message back to each of their codeword nodes, which corresponds to messages being sent from the bottom nodes, to the top nodes, in Figure 2.3.

For a fixed number of iterations, we can trace the computation path of the messages received at v_i through the graph in a similar manner to Example 2.3.1. This traceback will take the form of a tree which we will call the *computation tree* for v_i. The idea of unwrapping a graph with cycles into a tree was first introduced by Gallager [2] and further developed in [13, 34, 38].

For instance, for the node v_1 in Figure 2.3, we get the computation tree shown in Figure 2.5 after 2 iterations using the above message passing schedule. We will denote the computation tree at the vertex $v_i \in V_T$ for the lth iteration as $T_i(l) = (V_l, E_l)$.

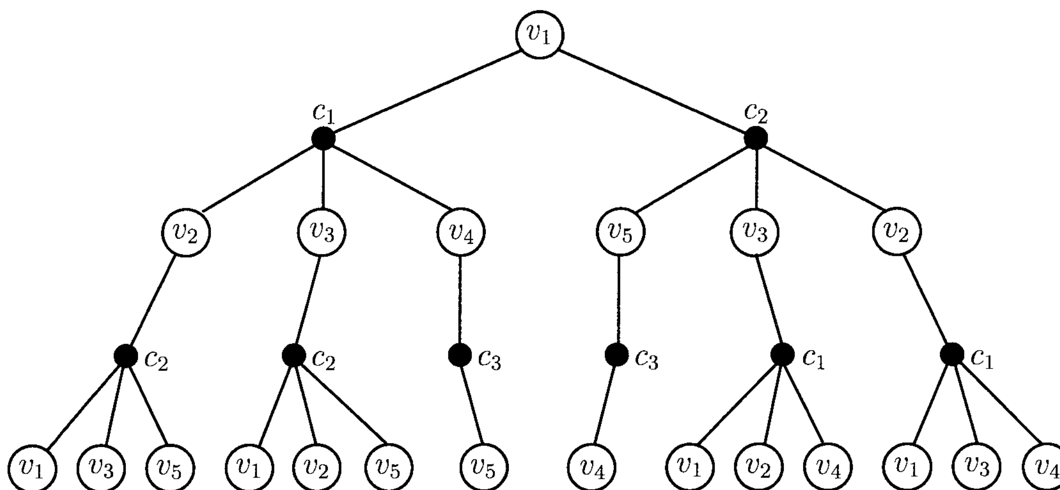


Figure 2.5: The computation tree for the (5,3,2) cycle code in Figure 2.3 after 2 iterations of message passing.

We can see in Figure 2.5 that the nodes v_2 and v_3 each appear four times in the computation tree. This means that the information from these nodes would be counted four times if we were to make a decision for v_1 after 2 iterations.

If we assume that a graph has a very large girth d, then a node will not occur twice in the tree for at least $(d-1)/2$ iterations. Gallager [24] used this idea to show that one can achieve an arbitrarily small decoding error probability with message passing on a graph of infinite girth, for LDPC codes with a given channel. Luby et al. [11], and Richardson and Urbanke [12], have further demonstrated that message passing can still achieve arbitrarily small decoding error even in the presence of finite cycles, i.e., repeated nodes in the computation tree, by invoking Azuma's inequality [39].

We now restrict our attention to the iterative min–sum algorithm (MSA). We will call a bit assignment to each codeword node in the computation tree, a *configuration* of the computation tree. If the bit assignment satisfies every local parity check in the tree, then we say it is a *locally consistent* configuration. If, for every codeword node $v \in V_c$, the corresponding set of codeword nodes in the computation tree, all have the same bit assignment and the computation tree is also locally consistent, then we say the bit assignment is a *globally consistent* configuration. Wiberg [13, Corollary 4.1] proved that the MSA finds the minimum weight locally consistent configuration

for the computation tree at each iteration. If we assume that all zeros codeword is transmitted, then the MSA decoder makes an error after l iterations, if there is a locally consistent configuration for the tree which has a one in the root node and a lower weight than the all zeros configuration. Wiberg [13] called such a configuration a *deviation* in the computation tree. In Chapter 6, we will quantify the effect of a deviation to bound the performance of the MSA decoder for cycle codes.

2.5 The Choice of Graphical Model and Decoding Algorithm

In this thesis we make use of a number of graphical models and iterative decoding algorithms in order to try and gain a better understanding of the behavior of iterative decoding algorithms on different types of graphs. For instance, for tail–biting codes we consider the behavior of both the iterative min–sum and sum–product algorithms, on a single cycle junction graph. We have done so, so that we can more easily observe the similarities and differences of the two algorithms for the same model. However, we also will look at the tail–biting trellis in the min–sum case to introduce the concept of pseudo–codewords and bound the decoding performance. In the case of cycle codes, we will consider several decoding algorithms, all on the Tanner graph for the code.

Of course, many of our results can, and have been, converted to an equivalent result for other models. In each case we have tried to choose the model which leads to the best understanding of the behavior of the iterative algorithm.

Chapter 3

The Iterative Sum–Product Decoding of Tail–Biting Codes

In this chapter we will discuss the behavior of the iterative sum–product algorithm (SPA), on graphs with a single cycle. This work is directly relevant to the study of iterative decoding of tail–biting codes, since their underlying graph has just one cycle [35, 36]. In the sections that follow we will investigate the relationship between the objective function θ_i, and the edge state ρ_i, when we perform message passing on a junction graph that contains a single cycle.

In Section 3.1, we will present a message passing schedule for a junction graph with a single cycle. In Section 3.2, we shall show that for strictly positive local kernels, the iterations of the SPA will always converge to a fixed point (which was also observed by Anderson and Hladik [40] and Weiss [38]). Moreover, the length of the cycle does not play a role in this convergence. We will also show that message passing in a single cycle junction graph can be viewed as a matrix operation.

In Section 3.3 we will compare the final state of an edge to the objective function for that edge and in Section 3.4, we will present an intuitive explanation for the entries of the matrix that we will associate with each edge. In Section 3.5, we shall generalize a result of McEliece and Rodemich [41], by showing that the SPA always converges to the correct MAP decision for a binary hidden variable. (This was also observed independently by Weiss [38]). Finally, in Section 3.6, we present two experiments and a theorem which illustrate the behavior of the SPA in the non–binary case. We will show that when the hidden variables can assume 3 or more values, the SPA may converge to an erroneous, i.e., non–MAP, decision, but this is apparently rare.

3.1 Message Passing Schedule

Consider a junction graph $G_J = (V_J, E_J)$, where each $v_i \in V_J$ has a local kernel $\alpha_i(x_{S_i})$. Let G_J contain exactly one cycle which has length l. We relabel the vertices in the cycle as $V' = \{v_1, \dots, v_l\}$, where $(v_i, v_{i+1}) \in E_J$ for $i \in [l-1]$ and $(v_1, v_l) \in E_J$. We also relabel the rest of the vertices arbitrarily as $\{v_{l+1}, \dots, v_M\}$. Define $E' = \{e_1, \dots, e_l\}$ where $e_i = (v_i, v_{i+1})$ for $i \in [l-1]$, and $e_l = (v_1, v_l)$.

A possible scheduling for the message passing begins by directing each edge that is not in the cycle, towards a vertex in the cycle. In the first stage, messages are sent only along these edges. A vertex not on the cycle will send a message to a neighbor, when, for the first time it has received messages from each of its other neighbors. With this scheduling, message passing begins at the leaves and proceeds until each vertex on the cycle has received messages from all of its neighbors not on the cycle.

Define $S'_i = (S_i \cap S_{i-1}) \cup (S_i \cap S_{i+1})$, for $i \in [l]$, i.e., all the variables in S_i that are shared with another vertex on the cycle. For each $v_i \in V'$, we define a new local kernel $\alpha'_i(x_{S'_i})$ as follows:

$$\alpha'_i(x_{S'_i}) = \sum_{x_{S_i \setminus S'_i}} \alpha_i(x_{S_i}) \prod_{\substack{(v_j, v_i) \in E_J \\ j \neq i-1, i+1}} \mu_{j,i}(x_{S_j \cap S_i}) \tag{3.1}$$

where we define $i - 1 = l$ for $i = 1$ and $i + 1 = 1$ for $i = l$, so v_{i-1} and v_{i+1} are the vertices adjacent to v_i in the cycle.

We can now perform message passing on the new junction graph $G' = (V', E')$ where each $v \in V'$ has a new local kernel, $\alpha'_i(x_{S'_i})$. Note that G' consists of only a single cycle.

Example 3.1.1 Consider the junction graph shown in Figure 3.1a where the vertices on the cycle have already been relabeled. The first stage of message passing results in G', shown in Figure 3.1b. The new junction graph is just a single cycle, with vertices, v_1, v_2, and v_3 having new local kernels $\alpha'_1(x_{S'_1})$, $\alpha'_2(x_{S'_2})$ and $\alpha'_3(x_{S'_3})$ respectively. Note that this junction graph can also be drawn as a junction tree.

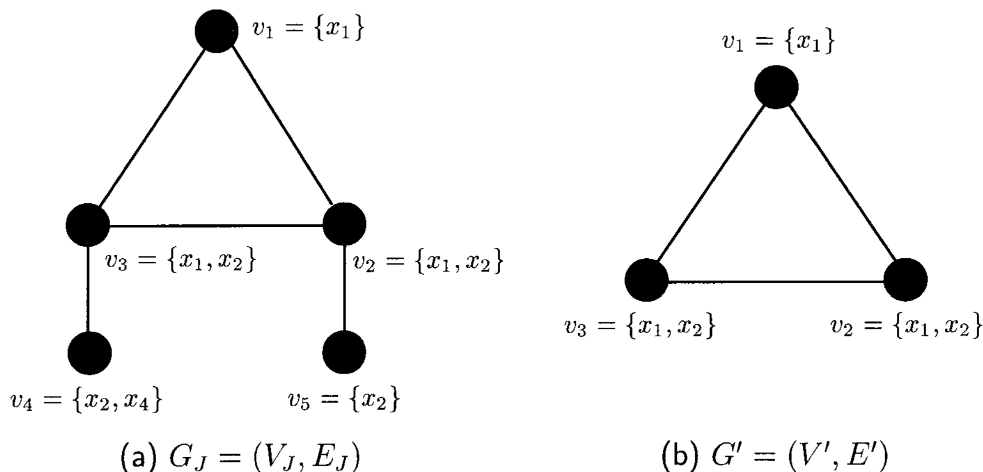


Figure 3.1: (a) The original junction graph and (b) the junction graph after the first stage of message passing has completed.

In the second stage of message passing, we send messages in parallel in both directions from every vertex in G'. In the next section we will show that computing the messages is equivalent to matrix multiplication and these messages will converge for strictly positive local kernels.

3.2 Message Passing Convergence

In Figure 3.2, we have three vertices v_{i-1}, v_i and $v_{i+1} \in V'$ on the single cycle that remains after the first stage of message passing has completed. If we define a lexicographic ordering of the alphabet sets $A_1, \ldots, A_n$, then we can write the message tables $\mu_{i-1,i}$ and $\mu_{i,i+1}$ as vectors of length $|A_{S'_{i-1} \cap S'_i}|$ and $|A_{S'_i \cap S'_{i+1}}|$ respectively, where the vector elements conform to this ordering. We can also rewrite the local kernel, $\alpha'_i(x_{S'_i})$, as an $|A_{S'_i \cap S'_{i+1}}| \times |A_{S'_{i-1} \cap S'_i}|$ matrix Φ_i, where we label the columns as elements of $A_{S'_{i-1} \cap S'_i}$ and the rows as elements of $A_{S'_i \cap S'_{i+1}}$ in the same lexicographic ordering. The entries of the matrix consist of the value of $\alpha'_i(x_{S'_i})$, when the row and column labeling is a consistent assignment of values to the variable set $x_{S'_i}$, and 0 otherwise. We construct the matrix Φ_i for all $v_i \in V'$.

Example 3.2.1 Consider the junction graph shown in Figure 3.1a. We define the

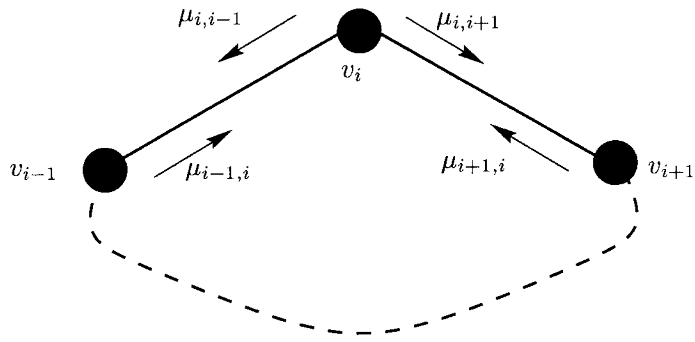


Figure 3.2: The vertices v_{i-1}, v_i and v_{i+1} that form a part of the cycle $v_1 v_2 \ldots v_l$ in V'.

local kernels as

v_1:

x_1	$\alpha_1(x_1)$
0	$\frac{1}{4}$
1	$\frac{3}{4}$

v_2:

$x_1 x_2$	$\alpha_2(x_1, x_2)$
00	$\frac{1}{3}$
01	$\frac{1}{6}$
10	$\frac{1}{4}$
11	$\frac{1}{4}$

v_3:

$x_1 x_2$	$\alpha_3(x_1, x_2)$
00	$\frac{1}{2}$
01	$\frac{1}{8}$
10	$\frac{1}{8}$
11	$\frac{1}{4}$

v_4:

$x_2 x_4$	$\alpha_4(x_2, x_4)$
00	$\frac{1}{3}$
01	$\frac{1}{3}$
10	$\frac{1}{6}$
11	$\frac{1}{6}$

v_5:

x_2	$\alpha_5(x_2)$
0	$\frac{2}{3}$
1	$\frac{1}{3}$

After the first stage of message passing we have the new set of local kernels for Figure 3.1b,

v_1:

x_1	$\alpha'_1(x_1)$
0	$\frac{1}{4}$
1	$\frac{3}{4}$

v_2:

$x_1 x_2$	$\alpha'_2(x_1, x_2)$
00	$\frac{8}{19}$
01	$\frac{2}{19}$
10	$\frac{6}{19}$
11	$\frac{3}{19}$

v_3:

$x_1 x_2$	$\alpha'_3(x_1, x_2)$
00	$\frac{8}{13}$
01	$\frac{1}{13}$
10	$\frac{2}{13}$
11	$\frac{2}{13}$

where the tables have been renormalized so the contents sum to 1. The matrices for

each node are then

$$\Phi_1 = \begin{pmatrix} \frac{1}{4} & 0 \\ 0 & \frac{3}{4} \end{pmatrix}, \quad \Phi_2 = \begin{pmatrix} \frac{8}{19} & 0 \\ \frac{2}{19} & 0 \\ 0 & \frac{6}{19} \\ 0 & \frac{3}{19} \end{pmatrix}, \quad \Phi_3 = \begin{pmatrix} \frac{8}{13} & \frac{1}{13} & 0 & 0 \\ 0 & 0 & \frac{2}{13} & \frac{2}{13} \end{pmatrix}.$$

From the update rule in (2.8), if we send messages in a clockwise direction from vertex v_i to v_{i+1}, we have

$$\mu_{i,i+1} = \Phi_i \; \mu_{i-1,i}. \tag{3.2}$$

If we send messages between these vertices in an anti–clockwise direction then, from (2.8) we have

$$\mu_{i,i-1} = \Phi_i^T \; \mu_{i+1,i}. \tag{3.3}$$

Messages are simply multiplied by a matrix for each iteration of the SPA. The elements of Φ_i are a function of the original local kernel, $\alpha_i(x_{S_i})$, and the incoming messages received from the vertices adjacent to v_i and outside the cycle. Therefore, the elements of the matrix Φ_i can be set to any value from 0 to 1, depending on how we define the local kernel and the incoming messages.

Now consider the messages passed in both directions along the edge $(v_i, v_{i+1}) \in E'$. A message passed in one direction will propagate through the vertices in the cycle due to the message passing schedule. Since the message is multiplied by a matrix at each vertex on the cycle, we can rewrite the updated message $\mu'_{i,i+1}$, in terms of the old message as

$$\mu'_{i,i+1} = M_i \; \mu_{i,i+1}, \tag{3.4}$$

where $M_i = \Phi_i \cdots \Phi_1 \Phi_l \cdots \Phi_{i+1}$ is the ordered product of the matrices associated with each vertex visited in the cycle. Similarly,

$$\mu'_{i+1,i} = M_i^T \; \mu_{i+1,i}. \tag{3.5}$$

If M_i is strictly positive, then by the Perron–Frobenius Theorem [42] there exists a unique largest positive eigenvalue with a pair of corresponding positive left and right eigenvectors called the *principal* eigenvectors of M_i. The normalized $\mu_{i,i+1}$ and $\mu_{i+1,i}$ will converge to the respective left and right principal eigenvectors of M_i. Since the largest eigenvalue of a strictly positive matrix is unique, the convergence to the same fixed point is guaranteed for any non–zero choice of initial vectors, $\mu_{i,i+1}$ and $\mu_{i+1,i}$.

Example 3.2.2 Continuing with Example 3.2.1, the matrices for each edge are

$$M_1 = \begin{pmatrix} \frac{11}{20} & 0 \\ 0 & \frac{9}{20} \end{pmatrix}, \quad M_2 = \begin{pmatrix} \frac{32}{99} & \frac{4}{99} & 0 & 0 \\ \frac{8}{99} & \frac{1}{99} & 0 & 0 \\ 0 & 0 & \frac{2}{11} & \frac{2}{11} \\ 0 & 0 & \frac{1}{11} & \frac{1}{11} \end{pmatrix}, \quad M_3 = \begin{pmatrix} \frac{11}{20} & 0 \\ 0 & \frac{9}{20} \end{pmatrix}$$

where the matrices have been normalized so the entries sum to 1.

If we compute the component–wise product of the left and right principal eigenvectors for each matrix, then the message passing will converge to the following final states:

e_1:

x_1	$\rho_1(x_1)$
0	1
1	0

e_2:

x_1x_2	$\rho_2(x_1, x_2)$
00	$\frac{4}{5}$
01	$\frac{1}{5}$
10	0
11	0

e_3:

x_1	$\rho_3(x_1)$
0	1
1	0

Note that the matrices M_1, M_2 and M_3 are not strictly positive (since x_1 occurs in all three local domains in the cycle), but in this case they all have a unique eigenvalue of largest modulus with non–negative left and right principal eigenvectors. Since there are zero entries, we now have to be careful on our choice of initial message vectors.

The matrix M_i is a matrix product of l, possibly arbitrary, matrices. We can therefore produce every matrix M_i from a cycle of any length. Since the state of each edge $e_i \in E'$ is the product of the messages on that edge, the state of e_i converges to a fixed point determined by M_i. Thus we can arrive at the same final state for any

length cycle. So the performance of the SPA relative to a MAP decoder is independent of the cycle length of the graph.

The SPA will converge for a single cycle junction graph when we use the message passing schedule outlined in the previous section. It can also be shown that the SPA will converge to the same fixed point for more general message passing schedules. For example, a serial message passing schedule, which starts by sending out messages from a single vertex and then sends a message to a neighboring vertex, each time a message is received from the other neighbor, will also converge to the same fixed point. The question remains as to what exactly it converges to.

In a junction tree the state of an edge is the objective function for that edge once message passing has terminated. However, this is not true in general for a junction graph with cycles. In the next section we will compare the SPA computed state of an edge to the objective function for that edge.

3.3 The Objective Function vs the Final State of an Edge

We would like to compute the objective function for some set of variables, $x_{E_i} \subset \mathbf{x}$ found on the edge e_i, of the cycle. The state of the edge e_i is the product of the messages on that edge. To see how the state computed at an edge compares to the objective function for that edge, let us first look at the following example.

Example 3.3.1 Consider again the junction graph shown in Figure 3.1b. We would like to compute the objective function $\theta_i(x_{E_i})$ for $E_i = \{1\}$. We can either examine the state of the edge (v_1, v_2), or the edge (v_1, v_3). Consider the edge $e_1 = (v_1, v_2)$. When the message passing has converged

$$\rho_1(x_1) = \mu_{2,1}(x_1)\,\mu_{1,2}(x_1)$$

where $\mu_{2,1} = \mu_{1,2} = [1, 0]^T$ are the respective left and right eigenvectors of the matrix

M_1, so $\rho_1 = [1, 0]$ is the component–wise product of these vectors.

The objective function $\theta_1(x_1)$ is

$$\begin{aligned}\theta_1(x_1) &= \sum_{x_2} \beta(x_1, x_2) \\ &= \sum_{x_2} \alpha_1'(x_1)\alpha_2'(x_1, x_2)\alpha_3'(x_1, x_2),\end{aligned}$$

which in vector form is $\theta_1 = [\frac{11}{20}, \frac{9}{20}]^T$. It is clear that in general the state is not equal to the objective function, i.e., $\rho_1 \neq \theta_1$. If we look at the diagonal entries of M_1, we find that $\theta_1 = \gamma \text{diag}(M_1)$, where γ is a normalizing constant.

In general the final state of the vertex v_i is the component–wise product of the left and right eigenvectors associated with the largest eigenvalue of the matrix M_i, which has the values of the objective function for v_i along its diagonal. Equivalently, we can say that the message passing estimates the matrix M_i by a matrix consisting of the outer product of the left and right eigenvectors of its largest eigenvalue. We know that the rank 1 estimate of M_i that minimizes the mean square error is the matrix consisting of the outer product of the left and right singular vectors associated with the largest singular value of M_i [43]. However, we are really only interested in minimizing the mean square error on the diagonal of M_i, in which case the singular value estimate is not optimal in general. In the next section we will examine the matrix M_i in detail and present an intuitive explanation of the meaning of the off–diagonal elements.

3.4 Interpretation of the Elements of M_1

Consider the junction graph shown in Figure 3.3a. We would like to compute the objective function for the variable set x_{E_1} associated with the edge e_1. The final state of e_1 is simply the component–wise product of the left and right eigenvectors associated with the largest eigenvalue of the matrix M_1. If we split the edge e_1 into two identical edges e_{1a} and e_{1b}, and break the cycle, we get the chain shown in Figure 3.3b. However, the vertices v_{1a} and v_{1b} and the vertices v_{2a} and v_{2b} have a common

variable set, so the resulting chain may not be a junction graph. In order to satisfy the junction property we can rename the variables associated with v_{1b} and v_{2b}, as $x_{S_{1b}} = \{x'_{1_1}, \ldots, x'_{1_r}\}$ and $x_{S_{2b}} = \{x'_{2_1}, \ldots, x'_{2_r}\}$ respectively and relabel the variables associated with the vertices $v_i, \ldots, v_l, 2 < i \leq l$ accordingly.

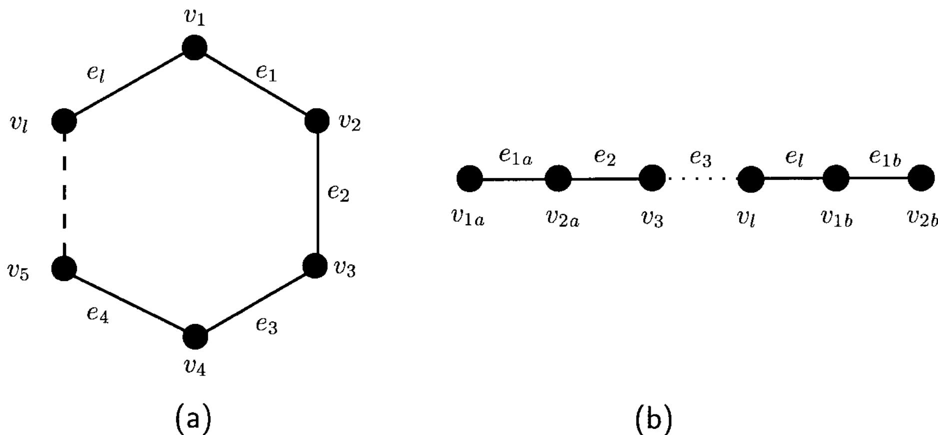


Figure 3.3: (a) The original junction graph with a single cycle and (b) the graph that results when the vertex v_1 has been split to form a chain with an additional copy of the edge e_1.

If we pass messages serially from v_{1a} to v_{2b}, then we can write $\mu_{1b,2b}$ as

$$\mu_{1b,2b} = M_1 \ \mu_{1a,2a}. \tag{3.6}$$

If the local kernels are probability matrices, then we can interpret M_1 as a probability transition matrix in a Markov chain. However, since the choice of the original kernels is arbitrary, M_1 is not necessarily a stochastic matrix and each row may sum to a different value.

The (i, j)–entry of M_1 is proportional to the sum of the probabilities of all paths through the chain, starting in state i and ending in state j. If we assume that there is a path from every starting state to every ending state, then the entries of M_1 are strictly positive, so M_1 is irreducible and the Perron–Frobenius theorem applies.

The diagonal of the matrix contains the sum of the probabilities that a path starts and ends in the same state. Initially we had the same variable set for e_{1a} and e_{1b},

so if we normalize M_1 (by dividing it by its trace in the case of probabilities), the elements on the diagonal of the matrix are the values of the objective function θ_1. The off–diagonal elements of M_1 only have a meaning in the non–cyclic graph of Figure 3.3b as a variable in e_1 can only be assigned a single value at a time. The values of the off–diagonal elements are determined by the structure of the local kernels at each vertex on the cycle and can affect the performance of the SPA for that cycle.

Generally in coding we are not concerned with the actual values of the state computed by the SPA, or even the values of the objective function. We are only concerned with making a decision based on these values. So we only care that the objective function and SPA state both result in the same decision. In the sections that follow we will show that when $|A_{E_i}| = 2$, so the hidden variable associated with e_i is binary valued, the SPA decision based on the final state is always a MAP decision. However, for a hidden variable set which takes on three or more values, the off–diagonal elements of M_1 tend to act like noise and can cause the SPA's decision to be different from the objective function's MAP decision.

3.5 Binary Valued Hidden Variables

Let the edge e have $|A_E| = 2$, so e contains only a single binary valued variable in its edge domain, say $x \in \mathbf{x}$. The matrix M is defined as

$$M = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix}, \tag{3.7}$$

where M has been normalized such that $m_{11} + m_{22} = 1$. Since we have normalized the trace of M to 1, the off–diagonal elements are no longer constrained to be less than 1. Let m_{11} and m_{22} correspond to the values of the objective function computed for $\theta(x = 0)$ and $\theta(x = 1)$ respectively. The largest eigenvalue of M is

$$\lambda = \frac{m_{11} + m_{22}}{2} + \sqrt{\left(\frac{m_{11} - m_{22}}{2}\right)^2 + m_{12}m_{21}} \tag{3.8}$$

and its right and left eigenvectors are

$$\begin{pmatrix} m_{12} \\ \lambda - m_{11} \end{pmatrix} \text{ and } \begin{pmatrix} m_{21} \\ \lambda - m_{11} \end{pmatrix} \tag{3.9}$$

respectively. Without loss of generality we assume $m_{11} > m_{22}$, so we would decide $x = 0$, if we were to make a MAP decision based on the objective function $\theta(x)$. For the component–wise product of the two eigenvectors to make the same decision, we must satisfy the inequality $(\lambda - m_{11})^2 \leq m_{12}m_{21}$, which is easily verified from (3.8).

Define

$$\Delta = m_{11} - \frac{m_{12}m_{21}}{m_{12}m_{21} + (\lambda - m_{11})^2} \tag{3.10}$$

to be the difference between the value computed by the objective function and the SPA state for $x = 0$. We observe the following about Δ.

- $\Delta = 0$ iff $m_{12}m_{21} = m_{11}m_{22}$, i.e., M is singular. Thus M has rank 1 and is equal to the outer product of the left and right eigenvectors associated with λ.

- The difference is only dependent on the product of the off–diagonal elements m_{12} and m_{21} and not on their individual values.

- We have

  $$\lim_{m_{12}m_{21} \to \infty} \Delta = \frac{1}{2}.$$

  So when the off–diagonal elements are large with respect to the diagonal, the decision of the SPA is less certain.

- We have

  $$\lim_{m_{12}m_{21} \to 0} \Delta = \begin{cases} 0 & m_{11} < \frac{1}{2} \\ \frac{1}{2} & m_{11} = \frac{1}{2} \\ 1 & m_{11} > \frac{1}{2} \end{cases}$$

  So when the off–diagonal elements are small with respect to the diagonal, the SPA tends to make a "hard decision."

Thus for a binary valued hidden variable the SPA will always make the correct

decision; however, the certainty of the decision is dependent on the product of the off–diagonal elements. The state of the edge will only be the same as its objective function when the matrix M is singular.

3.6 Non–Binary Valued Hidden Variables

For $|A_E| > 2$, it is much harder to characterize the performance of the SPA in terms of the matrix M. In this section we will present the results of two experiments and one theorem which highlight some of the characteristics of the SPA found for non–binary valued random variables.

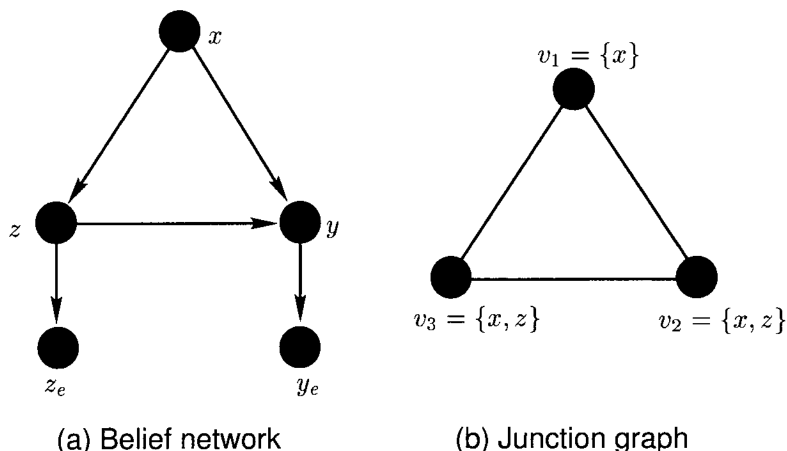
In our experiments the state and objective functions we deal with are always probabilities so we refer to the values computed by the state function as *SPA probabilities* and those computed from the objective function as *optimal* or *exact probabilities.* We count an error when the decision based on the SPA probabilities differs from the decision based on the exact probabilities.

3.6.1 Experiment 1

The first experiment demonstrates a simple junction graph for which the SPA estimates the values of the objective function very well and typically only makes an incorrect decision when the decision based on the exact probabilities is a close call.

Consider the belief network and its junction graph after the first stage of message passing, shown in Figures 3.4a and 3.4b respectively. The hidden variables x, y and z are all ternary $\{0, 1, 2\}$ valued. The dependencies for the belief network are defined as follows: x has a uniform prior; there is a symmetric channel with crossover probability p between x and z; y is the mod 3 sum of x and z; and z_e and y_e are noisy versions of z and y respectively, where the noise is Gaussian with mean 0 and variance σ^2.

If we fix the crossover probability, $p = 0.25$, and the variance of the noise, $\sigma^2 = 0.2$, then we would like to compute the objective function for the hidden variable x for a given observation of the variables z_e and y_e. Since x has a uniform prior, the local kernel $\alpha_1(x)$ at v_1 is a constant so the final state $\sigma_1 = \rho_1 = \rho_3$. In making a



(a) Belief network (b) Junction graph

Figure 3.4: (a) A simple belief network consisting of 5 nodes and 1 loop and a corresponding (b) junction graph shown after the first stage of message passing has completed.

decision using the SPA, we form the matrix M_1 at e_1 for a particular evidence set and then find the component–wise product of the left and right eigenvectors of its largest eigenvalue. The decision is then the value of x corresponding to the largest value in the component–wise product. This is equivalent to performing the SPA algorithm on the network for a large enough number of iterations for the decision to converge to its final state.

Figure 3.5a shows the SPA's decision regions for various observed values of z_e and y_e while Figure 3.5b shows the decision regions that are determined by computing the objective function for x. Figure 3.5c shows the regions where the decisions made by the SPA state function and the objective function differ. In this experiment the SPA only makes an incorrect decision when the objective function decision is at, or near, a decision region boundary, i.e., when the exact probabilities for 2 values of x are near, so the objective function decision is a close call.

For any fixed p, we can calculate the probability $P(E)$ of the SPA making an incorrect decision, with respect to the objective function decision, by integrating the joint distribution of z_e and y_e over the error region such as that in Figure 3.5c. The results are shown for various p in Figure 3.5d.

Alternatively we can look at how well the SPA estimates the objective function

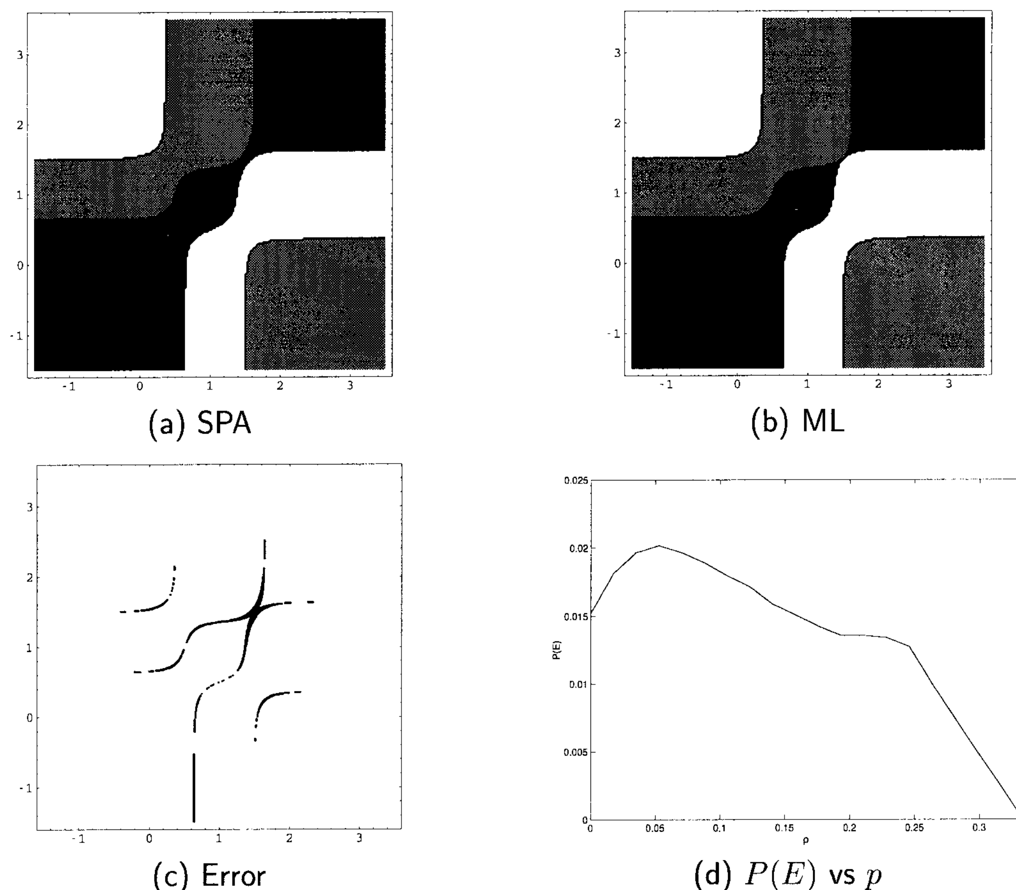


Figure 3.5: The decision regions for the hidden variable x where $p = 0.25$ and $\sigma^2 = 0.2$ using (a) the SPA and (b) the exact probabilities from the objective function. The black, grey and white regions correspond to deciding $x = 0$, 1 and 2 respectively. Plot (c) shows the regions where the SPA and the objective function make different decisions and plot (d) the $P(E)$ vs channel crossover probability p.

computed for x. Figure 3.6 shows the SPA probability vs the optimal probability, computed for $x = 0$, for 5000 trials. As we will see in the next experiment, on this particular network the SPA estimates the objective function quite well.

3.6.2 Experiment 2

The second experiment demonstrates the effect the off-diagonal elements can have on the convergence of the SPA to a correct decision.

Consider the following system. Define S to be the *signal matrix* consisting of a

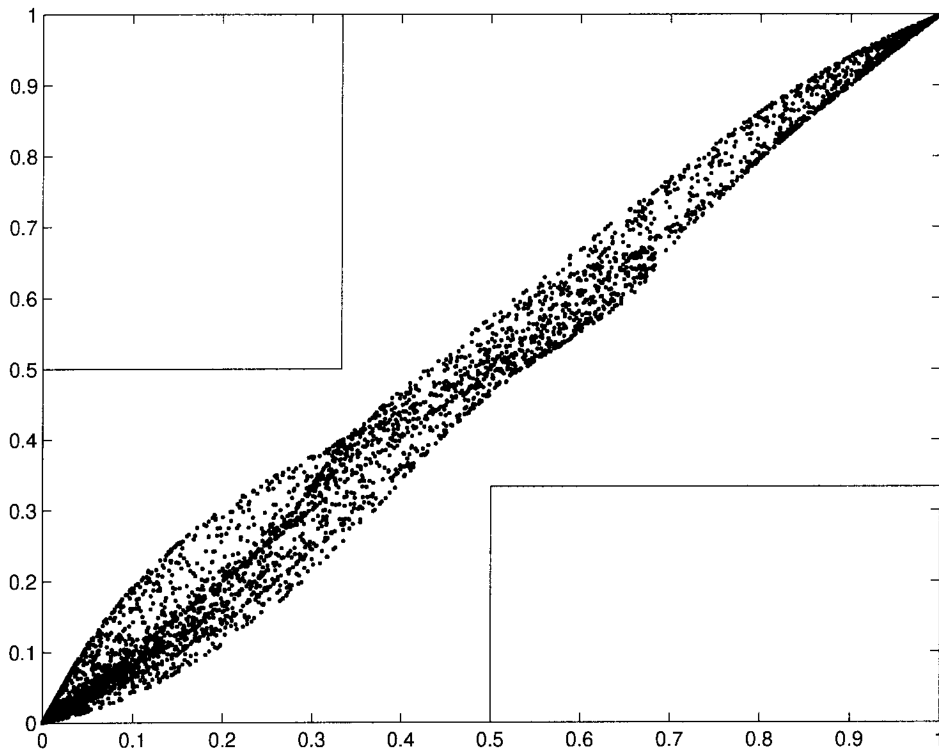


Figure 3.6: Plot of the state of x vs the objective function of x for $x = 0$ for 5000 trials for the network in Figure 3.4.

non–negative diagonal matrix with trace equal to 1. Define N to be the *noise matrix* consisting of a non–negative matrix with zero on its diagonal and all the off–diagonal elements less than 1. Let a and b be real non–negative constants such that $a + b = 1$ and define the matrix

$$M = aS + bN. \tag{3.11}$$

Now let M be the matrix associated with edge e and hidden variable x for a junction graph with a single cycle. We have already shown that we can create a junction graph to produce such an M. Since the state of e is the product of its messages, the values on the diagonal of M, or the signal matrix, are those calculated by the objective function for that edge, and the off–diagonal, noise matrix, values are due to the cycle in the junction graph. We can make the following observations about M.

- We have

$$\lim_{b \to 0} M = S,$$

so when the off–diagonal elements are 0, the SPA will make a hard decision and correctly decide on the largest element on the diagonal.

- We have

$$\lim_{a \to 0} M = N,$$

so when the diagonal elements are 0, the SPA decision is based entirely on the left and right eigenvectors associated with the largest eigenvalue of N. Thus the SPA decision is independent of the actual objective function and can be correct or incorrect depending on the nature of the noise.

- For intermediate values of a and b, the SPA may or may not make a correct decision depending on the mean diagonal–off–diagonal ratio (DODR) of the elements of M. Generally for a large DODR, the SPA makes a correct decision. The likelihood of a correct decision decreases with the DODR in a similar manner to the way codes perform more poorly as the signal to noise ratio is decreased.

Given the 3×3 matrices S and N, Figure 3.7a shows a plot of how the state probabilities calculated by the SPA vary in the simplex $m_{11} + m_{22} + m_{33} = 1$, as a is varied from 1 to 0. In each case, the state probability estimate starts in the corner corresponding to the largest element on the diagonal of S and moves towards the estimate given by performing the SPA on just the noise matrix, N. The four 3×3 matrices were chosen so that the SPA acting on the noise matrix alone would result in an incorrect decision. The "+" in the figure indicates the position of the diagonal of S in the simplex for each M. We can see that for large enough b, or small enough mean DODR, the SPA will make an incorrect decision.

It is possible for the noise matrix alone to result in the SPA making a correct decision, but the matrix M makes an incorrect decision [44]. However, we have run numerous experiments and found that this occurs extremely rarely.

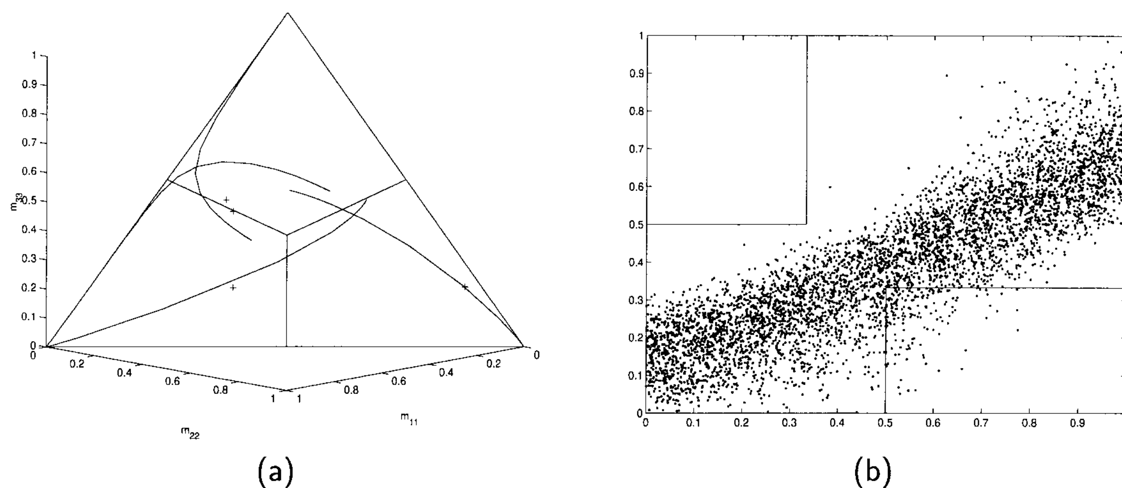


Figure 3.7: Plots of (a) how for a given S and N, the state probability vector varies as a is varied from 1 to 0 and (b) the SPA probability vs the optimal probability, computed for $x = 0$, for 5000 trials in which we have picked the off–diagonal elements uniformly between 0 and 1.

The plot in Figure 3.7b shows the SPA probability vs the optimal probability, computed for $x = 0$, for 5000 trials in which we have picked the off–diagonal elements uniformly between 0 and 1. As we can see when the elements are chosen at random, instead of based on an actual junction graph, as in Figure 3.6, the SPA does not perform as well.

The plot in Figure 3.8 shows the probability of making a correct decision vs the mean DODR for different distributions of the off–diagonal elements. We ran experiments where we selected the off–diagonal elements as the absolute value of a Gaussian random variable, as well as from an exponential and an uniform distribution, and we had similar results in each case. Thus it seems that the mean DODR is more important in determining the probability of a correct decision by the SPA than the actual distribution of the off–diagonal elements. The same results occurred in our experiments with 4×4 and 5×5 matrices.

To further confirm this we used the same distributions but fixed the mean DODR by multiplying the matrix by a constant and we plotted the $P(C)$ of the SPA vs the variance of the off–diagonal elements for various mean DODR's. The results are

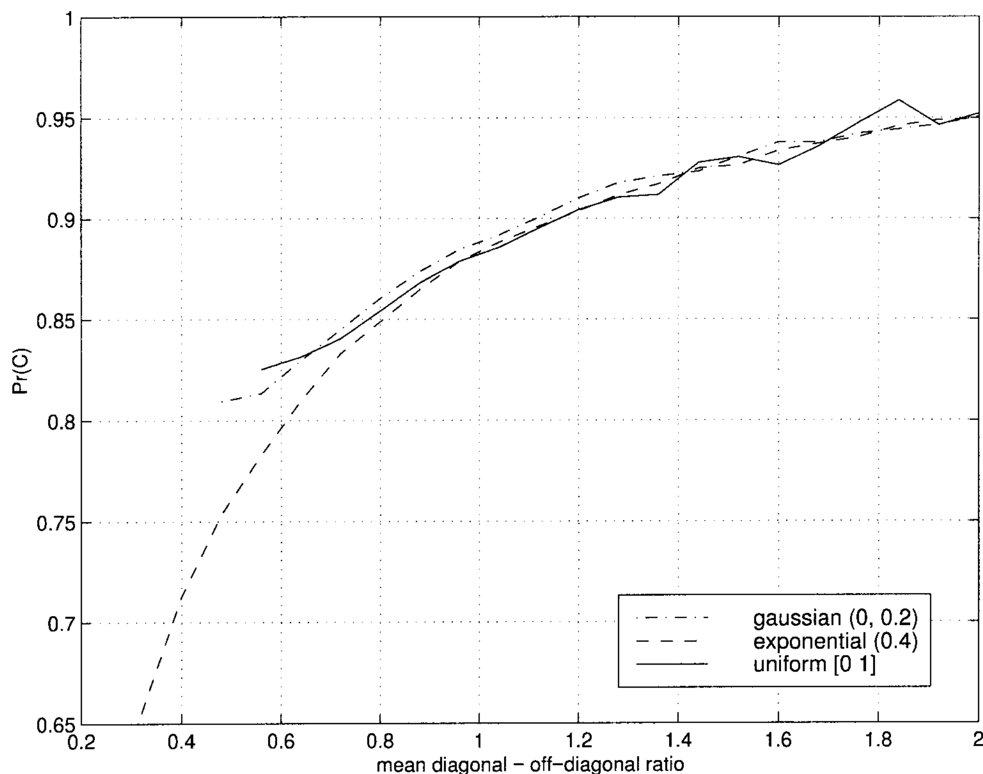


Figure 3.8: Plot of P(C) for the SPA vs the mean DODR for different distributions of the off–diagonal elements for various distributions of the off–diagonal elements.

shown in Figures 3.9a–d and surprisingly the performance does not seem to depend very much on the variance.

3.6.3 A Theorem

The theorem serves to illustrate the effect of a noise matrix where all the off–diagonal elements are the same. In this case the noise matrix N can be written as $N = J - I$, where J is the all ones matrix and I is the identity matrix. We shall show that in this case the SPA always converges to the correct decision regardless of the mean DODR.

Theorem 3.6.1 *Consider the matrix M from (3.11) for which $N = J - I$. Let λ be the largest eigenvalue of M with associated eigenvector $u = [u_1, \ldots, u_n]' \geq 0$. If $m_{11} > m_{ii}$, for $i = 2, \ldots, n$, then $u_1 > u_i$, for $i = 2, \ldots, n$.*

(The proof can be found in Appendix A.1.)

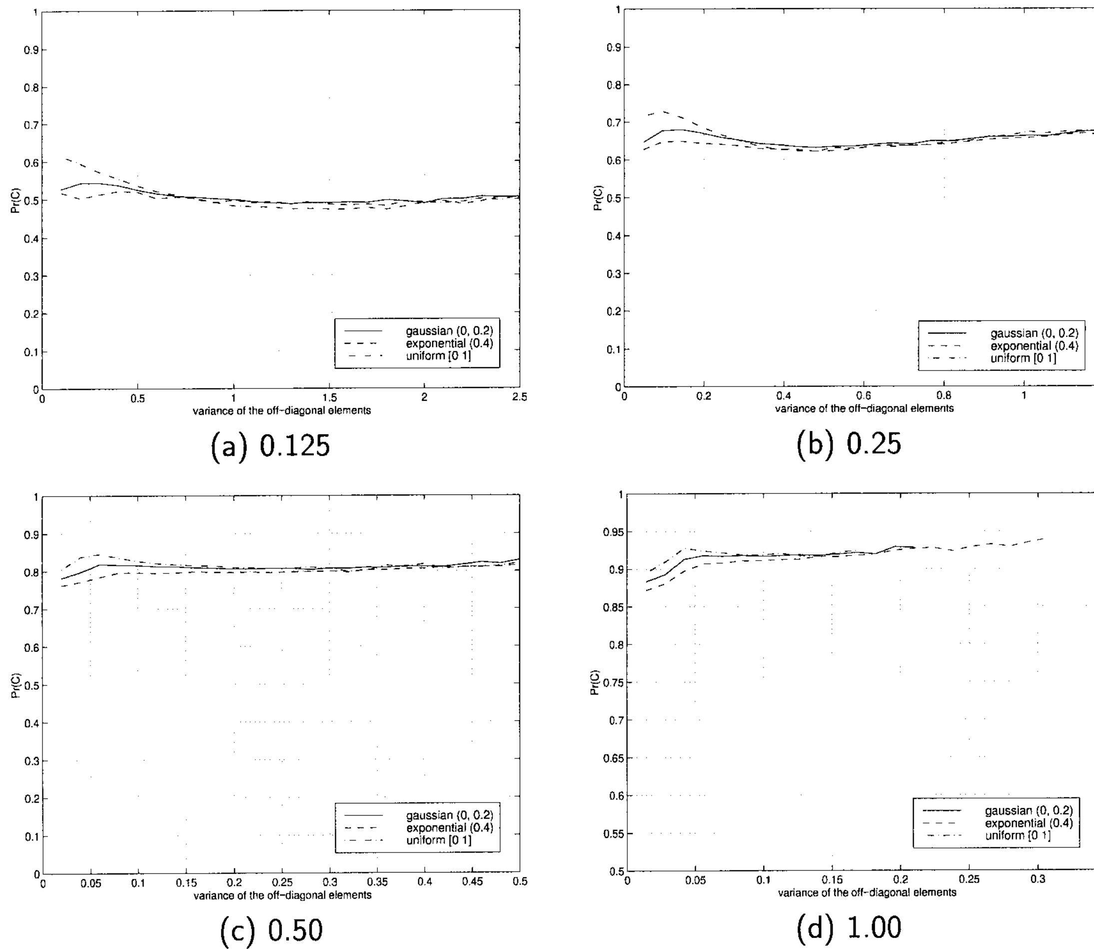


Figure 3.9: Plot of the P(C) vs the variance of the off–diagonal elements where the mean DODR is held constant.

Figure 3.10a shows a plot of how for a given S and $N = J - I$, the state probabilities move in the simplex as a is varied from 1 to 0. In each case, the state probability estimate, given by the SPA, starts in the corner corresponding to the largest element on the diagonal of M and moves towards the center, since the principal eigenvector of N is the all 1's vector. The SPA never leaves the correct decision region regardless of the value of a.

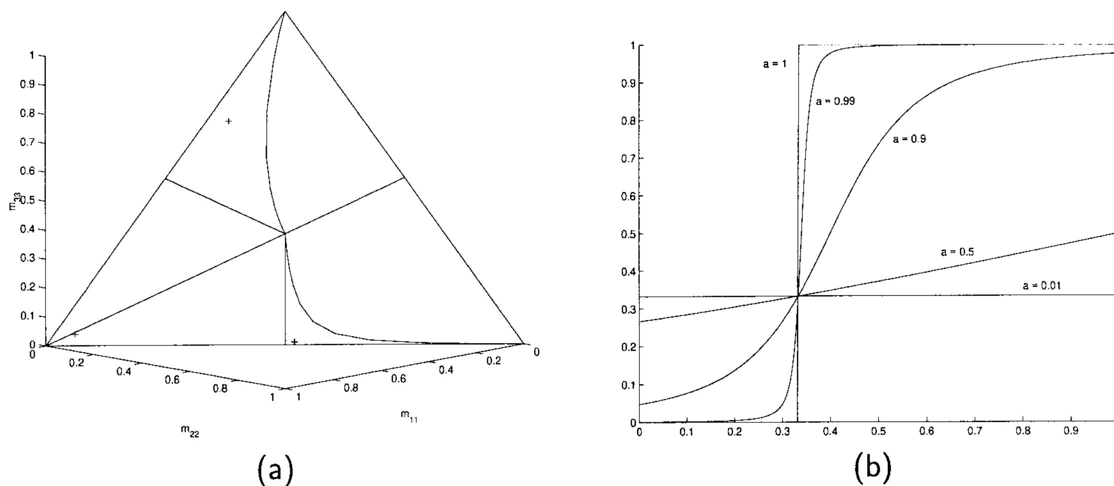


Figure 3.10: Plot of (a) how for a given S and $N = J - I$, the state probability vector varies as a is varied from 0 to 1 and (b) the SPA probability vs the optimal probability m, computed for for the matrix M in (3.12) for various values of a.

The plot in Figure 3.10b shows the SPA probability vs the optimal probability m, computed for for the matrix

$$M = \begin{bmatrix} am & b & b \\ b & a\left(\frac{1-m}{2}\right) & b \\ b & b & a\left(\frac{1-m}{2}\right) \end{bmatrix} \tag{3.12}$$

for various values of a. As a goes from 1 to 0, the certainty in the decision decreases, but the decision remains correct.

3.7 Conclusions

Message passing algorithms can be used to reduce the complexity of decoding tail–biting codes. For a single cycle graph the message passing converges for strictly positive local kernels and can be interpreted as a matrix multiplication at each vertex. For binary hidden variables the SPA will always make a correct decision but the actual probability can be anywhere on the correct side of 1/2. In the non–binary case, the performance of message passing with respect to an optimal decoder is dependent on

the off–diagonal elements of the matrix M. These off–diagonal elements are a function of the structure of the code and the noise in the channel. The SPA should have very good performance for a tail–biting code if we can design the code to minimize these off–diagonal elements.

Chapter 4

The Iterative Min–Sum Decoding of Tail–Biting Codes

In this chapter we turn to the behavior of the iterative min–sum algorithm (MSA) on graphs with a single cycle, the underlying graph for a tail–biting code. We will first develop the equivalent set of results for the min–sum algorithm as we did for the sum–product algorithm on a junction graph in Chapter 3. We will then turn our attention to the conventional tail–biting trellis description of the code to provide a deeper insight into the behavior of the MSA on single cycle graphs.

The MSA has been discussed in [36, 38, 40, 45]. Closely related results were discovered independently by Wiberg in his thesis [13], Weiss [38] and Forney et al. in [45].

The outline of the chapter is as follows. In Section 4.1, we discuss the behavior of the MSA on a single cycle junction graph. Specifically, we show that message passing on the junction graph is equivalent to a matrix multiplication, but that unlike the SPA, the MSA may converge to a periodic solution. In Section 4.1.3 we show that for binary hidden variables, the MSA will always make the correct decision, if it converges.

In Section 4.2, we turn to the tail–biting trellis description of the code. Using some basic concepts from graph theory, we prove a Perron–Frobenius theorem for the min–sum semiring. We then review the state space approach to convolutional codes in order to present some examples. We define a pseudo–codeword for a tail–biting trellis and obtain an estimate of the MSA decoder's performance using a union bound argument. Finally we present two examples of the MSA on a tail–biting code. In Section 4.3, we present our conclusions.

4.1 The Iterative Min–Sum Algorithm on Single Cycle Junction Graphs

Consider a junction graph that contains exactly one cycle. If we use the same message passing schedule as in Section 3.1, we need only consider the graph $G_J = (V_J, E_J)$, consisting of a single cycle of length l. Let $V_J = \{v_1, \ldots, v_l\}$ and $E_J = \{e_1, \ldots, e_l\}$, where $e_i = (v_i, v_{i+1})$ for $i \in [l-1]$ and $e_l = (v_l, v_1)$. For each $v_i \in V_J$ we have a local kernel $\alpha_i(x_{S_i})$.

From the update rule in (2.8), if we send a message in a clockwise direction from vertex v_i to v_{i+1} along e_i, we have

$$\mu_{i,i+1} = \Phi_i \; \mu_{i-1,i}, \tag{4.1}$$

where the matrix multiplication now occurs in the min–sum semiring. If we send messages between these vertices in an anti–clockwise direction, then we have

$$\mu_{i,i-1} = \Phi_i^T \; \mu_{i+1,i}. \tag{4.2}$$

So, like the SPA, messages are multiplied by a matrix for each iteration of the MSA. Since the message is multiplied by a matrix at each vertex on the cycle, we can rewrite the updated message $\mu'_{i,i+1}$, in terms of the old message as

$$\mu'_{i,i+1} = M_i \; \mu_{i,i+1}, \tag{4.3}$$

where $M_i = \Phi_i \cdots \Phi_1 \Phi_l \cdots \Phi_{i+1}$, is the ordered product of the matrices associated with each vertex visited in the cycle. Similarly,

$$\mu'_{i+1,i} = M_i^T \; \mu_{i+1,i}. \tag{4.4}$$

In the sum–product semiring, if M_i is strictly positive, then by the Perron–Frobenius theorem, the messages converge to a unique fixed point. We now present

an equivalent result for the min–sum semiring.

4.1.1 Message passing convergence

For the sum–product semiring the Perron–Frobenius theorem states [42]:

Theorem 4.1.1 (The Perron–Frobenius theorem) *Let M be an $s \times s$ irreducible and non–negative matrix. Then M has a positive eigenvector $\mathbf{x}$ with corresponding eigenvalue $\lambda > 0$. Moreover λ is algebraically simple and all other eigenvalues μ of M have $|\mu| \leq \lambda$.*

Furthermore, if M is primitive, then

$$\lim_{n \to \infty} \left[\frac{1}{\lambda} M \right]^n = \mathsf{P} > 0,$$

where $\mathsf{P} = \mathbf{x}\mathbf{y}^T$ has rank 1 and $\mathbf{x}$ and $\mathbf{y}$ are the positive right and left eigenvectors of λ respectively.

For the min–sum semiring we have the following theorem

Theorem 4.1.2 *Let M be the $s \times s$ irreducible matrix with entries from $\mathbb{R} \cup \infty$. Let M have a "critical cycle" of degree $l \in [s]$. Then M^l has a unique eigenvalue λ^l.*

Furthermore, if $l = 1$, then with min–sum arithmetic,

$$\left[\frac{1}{\lambda} M \right]^n = \mathsf{P} \textit{ for } n \geq n_0,$$

where n_0 is a finite constant and $\mathsf{P} = \mathbf{x}\mathbf{y}^T$ is a rank 1 matrix with $\mathbf{x}$ and $\mathbf{y}$, the unique right and left eigenvectors of λ respectively.

We will present a more general theorem and define a critical cycle in Section 4.2.2. We can see that unlike the SPA, the MSA may converge to a periodic solution. However, as we shall see in Section 4.2.6, when it does converge, it makes a maximum likelihood decision.

4.1.2 The objective function vs the final state of an edge

We would like to assign the set of values to the variables on the cycle such that the a posteriori probability of the unobserved variables given the observed variables is maximized. Since we are using the MSA, we will have negative log–likelihoods for our local kernels [16, 38], so we would like to compute the lowest cost assignment for the junction graph.

Consider some set of variables $x_{E_i} \subset \mathbf{x}$ found on the edge e_i of the cycle. The objective function for e_i will be the lowest cost junction graph for each possible assignment to the variables in x_{E_i}. The state of the edge e_i is the sum of the messages along the edge. If the iterates of the matrix M_i converge to a rank 1 matrix, then the final state is simply the component–wise sum of the left and right eigenvectors associated with the unique eigenvalue of the matrix M_i.

Since the local kernels are log–likelihood matrices, we can interpret M_i as an adjacency matrix for a graph with a cost associated with each edge. The (j, k)–entry of M_i is proportional to the lowest cost path starting in state j and ending in state k. If we assume that there is a path from every starting state to every ending state, then the entries of M_i are strictly positive, so M_i is irreducible and the min–sum Perron–Frobenius theorem applies. An ML decoder will compute $\min\{[M_i]_{j,j}\}$, i.e., the lowest cost path that starts and ends in the same state, since that corresponds to the most likely assignment for the junction graph.

In the next section we will show that when $|A_{E_i}| = 2$, i.e., the hidden variable associated with e_i is binary valued, an MSA decision based on the final state is always an ML decision. However, for a hidden variable set which takes on 3 or more values, the MSA's decision can be different from the objective function's decision.

4.1.3 Binary valued hidden variables

Let the edge e have $|A_E| = 2$, so e contains only a single binary valued variable in its edge domain, say $x \in \mathbf{x}$. The matrix M is defined as

$$M = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix}, \tag{4.5}$$

where M has been normalized such that $\min(m_{11}, m_{12}, m_{21}, m_{22}) = 0$. Let m_{11} and m_{22} correspond to the values of the objective function computed for $\theta(x = 0)$ and $\theta(x = 1)$ respectively. Without loss of generality we assume $m_{11} < m_{22}$, so we would decide $x = 0$, if we were to make an ML decision based on the objective function $\theta(x)$.

Define

$$\Delta = (m_{12} + m_{21}) - 2m_{11} \tag{4.6}$$

to be the difference between the sum of the off-diagonal elements and twice the smallest diagonal element of M. We note the following about Δ.

- Δ is only dependent on the sum of the off–diagonal elements m_{12} and m_{21} and not on their individual values.

- For $\Delta > 0$, the unique eigenvalue of M is

  $$\lambda = m_{11}, \tag{4.7}$$

  and its right and left eigenvectors are

  $$\begin{pmatrix} m_{11} \\ m_{21} \end{pmatrix} \text{ and } \begin{pmatrix} m_{11} \\ m_{12} \end{pmatrix} \tag{4.8}$$

  respectively. The component–wise sum of the two eigenvectors always makes the correct decision.

- For $\Delta < 0$, M^2 converges to a matrix of rank 2 where the eigenvectors have

eigenvalue

$$\lambda = m_{12} + m_{21}, \tag{4.9}$$

and

$$M^n = \begin{pmatrix} m_{12} + m_{21} & m_{12} + m_{11} \\ m_{11} + m_{21} & m_{12} + m_{21} \end{pmatrix} + \frac{n-2}{2}(m_{12} + m_{21})J \tag{4.10}$$

for n even and

$$M^n = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{11} \end{pmatrix} + \frac{n-1}{2}(m_{12} + m_{21})J \tag{4.11}$$

for n odd, where J is the 2×2, all ones matrix.

Thus for a binary valued hidden variable the MSA will always make the correct decision when it converges. When the sum of the off–diagonal elements is smaller than twice the value of the smallest diagonal element, the MSA solution will oscillate with period 2. Either solution will determine that both values of x are equally likely, so the MSA provides no information.

4.2 The Iterative Min–Sum Algorithm on a Tail–Biting Trellis

We now turn to a different graphical model to provide greater insight into the behavior of the MSA on a graph for a tail–biting code. The model is the tail–biting trellis for the code [35, 36].

We will proceed as follows. In Section 4.2.1 we will introduce the basic notation and definitions of graph theory that we require. In Section 4.2.2 we prove a Perron–Frobenius theorem for the min–sum semiring. In Section 4.2.3 we will introduce the tail–biting trellis and in Section 4.2.4 the state–space approach to convolutional codes. In Section 4.2.5 we present some examples of the Perron–Frobenius theorem for the

min–sum semiring on a tail–biting trellis for a convolutional code. In Section 4.2.6 we present a theorem showing the MSA converges to the dominant pseudo–codeword and in Section 4.2.7 we use this theorem to estimate the decoding performance. Finally in Section 4.2.8 we present two examples of the MSA decoder performance for a tail–biting code.

4.2.1 Basic notation and definitions

In this section, we introduce the necessary algebraic and combinatorial tools required to prove the Perron–Frobenius theorem for the min–sum semiring. We will make use of the notation of Stanley [46, Section 4.7].

Let $D = (V, E)$ be a finite *directed graph* or *digraph*, where V is a set of vertices and E is a set of edges, with $E \subseteq V \times V$. Let $|V| = m$ and $|E| = r$. If $e = (u, v)$, then e is said to be a *directed edge* from u to v with *initial vertex* u and *final vertex* v, denoted $u = \text{init}(e)$ and $v = \text{fin}(e)$ respectively. Figure 4.1 shows a digraph with vertex set $V = \{1, 2, 3, 4\}$ and edge set $E = \{(1,2), (2,3), (3,4), (4,1), (4,2), (3,2)\}$.

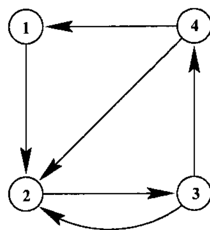


Figure 4.1: A digraph.

A *path* of length n, from u to v in D is a sequence of edges $e_1 e_2 \dots e_n$ such that $u = \text{init}(e_1)$, $v = \text{fin}(e_n)$ and $\text{fin}(e_i) = \text{init}(e_{i+1})$ for $1 \le i < n$. A path from u to v is called *closed* if $u = v$. A path from u to v is called *simple* if each vertex on the path is distinct. A simple closed path is called a *cycle*. Figure 4.1 has a cycle ((2,3),(3,4),(4,2)) of length 3. If D has no pair of edges, e and $f \in E$ such that $\text{init}(e) = \text{init}(f)$ and $\text{fin}(e) = \text{fin}(f)$, then we can write a path of length n as a sequence of $n+1$ vertices without any ambiguity. The above cycle can be written as

(2,3,4,2).

Now let $w : E \to R$ be a *weight function* on edge set E taking values in some commutative semiring R. In this chapter, we only consider the min–sum and the sum–product semirings. If $\mathcal{P} = e_1 e_2 \dots e_n$ is a path, then the weight of $\mathcal{P}$ is defined as $w(\mathcal{P}) = w(e_1)w(e_2)\cdots w(e_n)$. By convention, if $e \notin E$, then $w(e) = 0$, where 0 is the additive identity in R.

We can represent a digraph and its weight function in matrix form. Define the *incidence matrix* A to be the square matrix with entries

$$[A]_{ij} = \sum_{e} w(e),$$

where e are the edges with $i = \text{init}(e)$ and $j = \text{fin}(e)$. If all the edges in D have weight 1, then Figure 4.1 will have the incidence matrix

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\!\!\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix} \end{array}. \tag{4.12}$$

Similarly we can define

$$A_{ij}(n) = \sum_{\mathcal{P}} w(\mathcal{P})$$

to be the sum of the weights over all paths $\mathcal{P}$ in D, of length n from vertex i to vertex j. Note that $A_{ij}(1)$ is simply the (i, j)-entry of A. We now have the following simple theorem.

Theorem 4.2.1 *The (i, j)-entry of A^n is equal to $A_{ij}(n)$. (By convention we define $A^0 = I$.)*

(The proof is given in Appendix A.2.1.)

If all the edges in D have weight 1, then in the sum–product semiring, the (i, j)-

entry of A^n is equal to the number of paths of length n between vertex i and vertex j. For example, from Figure 4.1 and (4.12), we have

$$A^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 2 & 1 & 1 \\ 1 & 2 & 2 & 1 \\ 1 & 2 & 1 & 1 \end{pmatrix} \end{array} \tag{4.13}$$

which shows, for example, that there are 2 paths from vertex 4 to vertex 2 of length 4, namely $(4, 2, 3, 4, 2)$ and $(4, 1, 2, 3, 2)$. We can also see that there is no path of length 4 from vertex 1 to vertex 4.

If D now has incidence matrix

$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} \infty & 0 & \infty & \infty \\ \infty & \infty & 3 & \infty \\ \infty & 4 & \infty & 2 \\ 1 & 2 & \infty & \infty \end{pmatrix} \end{array}, \tag{4.14}$$

then in the min–sum semiring, the (i, j)-entry of A^n is equal to the lowest weight path of length n between vertex i and vertex j. For example,

$$A^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 6 & 7 & 10 & \infty \\ \infty & 6 & 8 & 12 \\ 10 & 11 & 6 & 9 \\ 6 & 7 & 10 & 6 \end{pmatrix} \end{array} \tag{4.15}$$

so the least weight path from vertex 4 to vertex 2 of length 4, has weight 7 which corresponds to the path $(4, 2, 3, 4, 2)$.

We now present the Perron–Frobenius theorem for the min–sum semiring.

4.2.2 The Perron–Frobenius theorem for the min–sum semiring

Let $D = (V, E)$ be a strongly connected digraph, i.e., for any two vertices u and $v \in V$, there is a path from u to v. Let w be the weight function for D and A be the corresponding incidence matrix. Since D is strongly connected, A is an irreducible matrix [47, Problem 31E].

Assume that among all cycles in D, there is a unique cycle with minimum average edge weight λ. We call this cycle the *critical cycle* in D. Let C be the set of vertices in the critical cycle, with $|C| = l$. Note that since $C \subseteq V$, we have $1 \leq l \leq m$.

The Perron–Frobenius theorem for the min–sum semiring is now:

Theorem 4.2.2 *For n sufficiently large, with min–sum arithmetic,*

$$\left[\frac{1}{\lambda}A\right]^{ln} = \mathsf{P}, \tag{4.16}$$

where P is a fixed matrix and λ^l is the unique eigenvalue of A^l.

For the special case when the critical cycle is a self loop, $\mathsf{P} = \mathbf{x}\mathbf{y}^T$ has rank 1, with $\mathbf{x} = [x_1, \ldots, x_m]$ and $\mathbf{y} = [y_1, \ldots, y_m]$, the unique right and left eigenvectors of λ respectively.

(The proof is given in Appendix A.2.2.)

The entry $x_i \in \mathbf{x}$ corresponds to the least weight path in D from vertex i to the critical cycle, and $y_j \in \mathbf{y}$ corresponds to the least weight path in D from the critical cycle to vertex j. Note that in the min–sum semiring, an eigenvalue, if it exists, is unique.

We now give a brief introduction to tail–biting trellises and convolutional codes in order to present some examples of the Perron–Frobenius theorem for the min–sum semiring.

4.2.3 Tail–biting trellises

In this section, we will introduce tail–biting trellises and convolutional codes. We will use the notation of McEliece [48] to describe the basic structure of a trellis. For further details we refer the reader to [35].

A tail–biting trellis $T = (V, E)$ of length N is a finite digraph where V is partitioned into N classes $V_0, V_1, \ldots, V_{N-1}$. The edge set is the disjoint union of N sets of edges $E_0, E_1, \ldots, E_{N-1}$ where $E_i \subseteq V_i \times V_{i+1}$ for $0 \leq i \leq N-1$. By convention we will perform index arithmetic modulo N, for a tail–biting trellis.

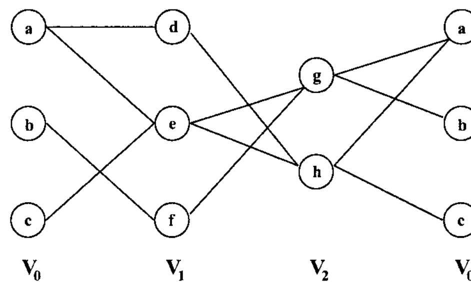


Figure 4.2: A tail–biting trellis of length 3.

Figure 4.2 shows an example of a tail–biting trellis of length 3. Note that for convenience we have repeated the vertices of V_0 after V_2 instead of showing the trellis wrapping around on itself. The trellis has vertex sets

$$V_0 = \{a, b, c\},\ V_1 = \{d, e, f\},\ V_2 = \{g, h\}$$

and edge sets

$$\begin{aligned} E_0 &= \{(a, d), (a, e), (b, f), (c, e)\}, \\ E_1 &= \{(d, h), (e, g), (e, h), (f, g)\}, \\ E_2 &= \{(g, a), (g, b), (h, a), (h, c)\}. \end{aligned}$$

Now let $w : E \to R$ once again be a weight function on edge set E taking values in R. We can represent each edge set E_i, for $0 \leq i \leq N-1$, by the matrix A_i. Define

the $|V_i| \times |V_{i+1}|$ incidence matrix A_i, to be the matrix with entries

$$[A_i]_{jk} = \sum_e w(e)$$

where e are the edges with $\text{init}(e) = j \in V_i$ and $\text{fin}(e) = k \in V_{i+1}$. For example, the tail–biting trellis in Figure 4.2, with all edge weights being 1, can be represented by the matrices

$$A_0 = \begin{array}{c} \\ a \\ b \\ c \end{array}\!\!\begin{array}{c} \begin{array}{ccc} d & e & f \end{array} \\ \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \end{array}, \quad A_1 = \begin{array}{c} \\ d \\ e \\ f \end{array}\!\!\begin{array}{c} \begin{array}{cc} g & h \end{array} \\ \begin{pmatrix} 0 & 1 \\ 1 & 1 \\ 1 & 0 \end{pmatrix} \end{array}, \quad A_2 = \begin{array}{c} \\ g \\ h \end{array}\!\!\begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \end{array}.$$

We call $A_0, A_1, \ldots, A_{N-1}$ the *matrix representation* of the tail–biting trellis. The *matrix product* $\mathbf{A} = A_0 A_1 \cdots A_{N-1}$ is defined, since the number of columns of A_i is equal to the number of rows of A_{i+1} is equal to $|V_{i+1}|$ for $0 \leq i < N-1$.

A *path* of length n, from u to v in T, is a sequence of edges $e_1 e_2 \ldots e_n$ such that if $e_1 \in E_j$, then $e_i \in E_{j+i-1}$ for $2 \leq i \leq n$. Of course we still must have $u = \text{init}(e_1)$, $v = \text{fin}(e_n)$ and $\text{fin}(e_i) = \text{init}(e_{i+1})$ for $1 \leq i < n$.

Once again we will define $A_{ij}(n)$ to be the sum of the weights of all paths of length n in T. If we restrict i and $j \in V_0$, then $A_{ij}(n)$ is only defined for $n = lN$, for some $l \in \mathbb{Z}$. By the same proof as Theorem 4.2.1, it can be shown that the (i,j)-entry of $\mathbf{A}^l$ is equal to $A_{ij}(lN)$. We summarize this as a corollary to Theorem 4.2.1.

Corollary 4.2.3 *The (i,j)-entry of the matrix product $\mathbf{A}^l = (A_0 A_1 \ldots A_{N-1})^l$ for $l = 0, 1, 2, \ldots$ is equal to $A_{ij}(lN)$.*

If all the weights on the edge set E are 1, then in the sum–product semiring $A_{ii}(N)$ represents the number of closed paths of length N, from vertex i in the trellis. For

example, for Figure 4.2,

$$\mathbf{A} = \begin{array}{c} \\ a \\ b \\ c \end{array}\!\!\begin{array}{c} \begin{array}{ccc} a & b & c \end{array} \\ \begin{pmatrix} 3 & 1 & 2 \\ 1 & 1 & 0 \\ 2 & 1 & 1 \end{pmatrix} \end{array},$$

so there are 3 closed paths of length 3 from vertex a, namely (a, d, h, a), (a, e, g, a) and (a, e, h, a), while there is only 1 from vertices b or c.

We define the *minimal state complexity* of a trellis, denoted $\mathcal{S}_{min}$, to be

$$\mathcal{S}_{min} = \min_{0 \leq i \leq N-1} |V_i|,$$

so $\mathcal{S}_{min}$ is the minimum cardinality of a class $V_i \in V$. The trellis in Figure 4.2 has minimal state complexity, $\mathcal{S}_{min} = 2$, since $|V_1| = 2$ and $|V_0| = |V_2| = 3$.

For a trellis, we define a *cycle* to be a closed path of length N and a *pseudo-cycle* to be a simple closed path of length lN. We will call l the *degree* of the pseudo-cycle. Since we can visit each vertex only once in a pseudo–cycle, $l \leq \mathcal{S}_{min}$. For fixed N, since $\mathcal{S}_{min}$ is finite, the number of pseudo–cycles in the trellis is also finite.

We now introduce the tail–biting trellis for convolutional codes.

4.2.4 Convolutional codes

In this section we will briefly review the state–space approach to convolutional codes in order to derive the finite state machine and the tail–biting trellis for the code.

An (n, k, m) *convolutional encoder* is a finite state machine with m memory elements which determine a mapping between the k-dimensional input block and the n-dimensional output block. We denote the input blocks $\mathbf{u_0}, \mathbf{u_1}, \ldots$, the output blocks $\mathbf{x_0}, \mathbf{x_1}, \ldots$ and the states $\mathbf{s_0}, \mathbf{s_1}, \ldots$. The ith output block and $i+1$th state are linear functions of the ith input block and state, which we can write as

$$\mathbf{s_{i+1}} = \mathbf{s_i}\mathcal{A} + \mathbf{u_i}\mathcal{B} \quad (4.17)$$

$$\mathbf{x_i} = \mathbf{s_i}\mathcal{C} + \mathbf{u_i}\mathcal{D} \quad (4.18)$$

(4.19)

where $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$ are matrices with entries from $GF(2)$. The initial state $\mathbf{s_0}$ is arbitrary.

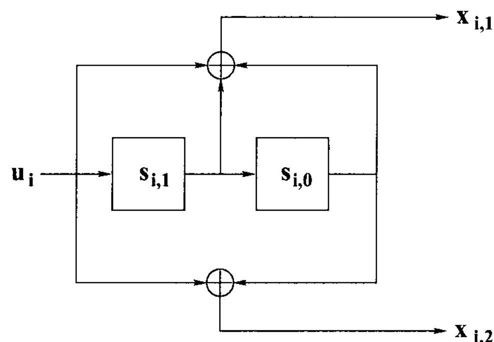


Figure 4.3: A rate 1/2, constraint length 2 convolutional encoder.

For example, the $(2,1,2)$ encoder shown in Figure 4.3 has input block $\mathbf{u_i}$, output block $\mathbf{x_i} = (x_{i,1}, x_{i,2})$ and state $\mathbf{s_i} = (s_{i,0}, s_{i,1})$ at time index i. The future output and states can be determined from the matrices,

$$\mathcal{A} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}, \quad \mathcal{B} = \begin{pmatrix} 1 & 0 \end{pmatrix}, \quad \mathcal{C} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \quad \mathcal{D} = \begin{pmatrix} 1 & 1 \end{pmatrix}.$$

For each convolutional encoder, we can construct a *state diagram* with 2^m vertices. The state diagram is a weighted digraph where each vertex corresponds to a state of the encoder and each edge corresponds to a transition from one state to another. Figure 4.4a shows the state diagram for the $(2,1,2)$ convolutional encoder. We have labelled the edges with the corresponding input/outputs for each transition.

If we define an ordered time axis, or *index set* $I = \{0, 1, 2, \ldots\}$, then we can construct a trellis for the convolutional encoder where we associate each vertex set V_i, with the set of possible states at time i and each edge set E_i, with the set of possible transitions between time i and $i+1$. The trellis is simply a time indexed version of the state diagram.

Figure 4.4b shows a trellis section for the $(2,1,2)$ convolutional encoder where

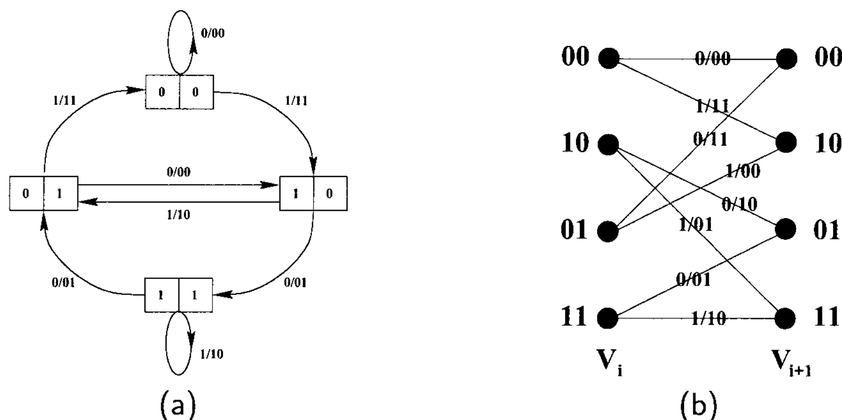


Figure 4.4: The (a) state diagram and (b) trellis section for the convolutional encoder with input and output weights on the edges.

once again we have labelled the edges with the corresponding input/outputs for each transition. If we force the convolutional encoder to start in a certain state, or subset of states, then some states will not occur for the first few time indices, so the trellis will initially contain fewer states and edges. We will call a section of the trellis *full*, if all 2^m states of the convolutional encoder occur in V_i and V_{i+1}. For example, if we restrict the convolutional encoder to start in the all zeros state, then for an encoder with memory m, the first full trellis section occurs at time index m.

We can construct a tail–biting trellis of length N for a convolutional code, by "pasting together" N full sections of the trellis, defined over the index set $I = \{0, 1, \ldots N-1\}$, the vertex set $V_0, V_1, \ldots, V_{N-1}$ and edge set $E_0, E_1, \ldots, E_{N-1}$. Since every section of the tail–biting trellis will be the same, we say the trellis is *time invariant.* A 5–section tail–biting trellis for the encoder in Figure 4.3 is shown in Figure 4.5, where we have repeated the vertices of V_0 after V_4 instead of showing the trellis wrapping around on itself.

Since the state diagram is a digraph, we can construct its incidence matrix A. If each edge has weight 1, then the state diagram in Figure 4.4a will have incidence

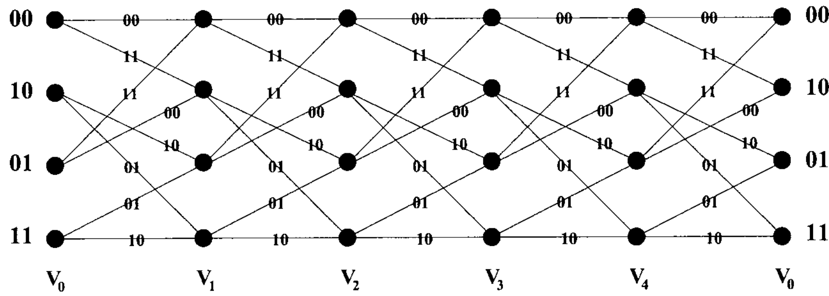


Figure 4.5: A 5–section tail–biting trellis for the encoder in Figure 4.3.

matrix

$$A = \begin{array}{c} \\ 00 \\ 10 \\ 01 \\ 11 \end{array} \begin{array}{c} \begin{array}{cccc} 00 & 10 & 01 & 11 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \end{array}. \tag{4.20}$$

Note that the corresponding tail–biting trellis will have $A_i = A$, for $0 \leq i \leq N-1$, where A_i is the incidence matrix of section i of the trellis.

If the weight function w is the Hamming weight of the output for each edge, then the state diagram in Figure 4.4a will have incidence matrix

$$A = \begin{array}{c} \\ 00 \\ 10 \\ 01 \\ 11 \end{array} \begin{array}{c} \begin{array}{cccc} 00 & 10 & 01 & 11 \end{array} \\ \begin{pmatrix} 0 & 2 & \infty & \infty \\ \infty & \infty & 1 & 1 \\ 2 & 0 & \infty & \infty \\ \infty & \infty & 1 & 1 \end{pmatrix} \end{array}, \tag{4.21}$$

where a Hamming weight of ∞ is assigned if no edge exists.

A convolutional code is defined as the set of all possible output sequences corresponding to infinite paths in the trellis or state diagram, which start in the all zeros state. A tail–biting code of dimension N is defined as the set of all possible output sequences corresponding to cycles in the tail–biting trellis. We call the output se-

quence associated with a cycle a *codeword* and the output sequence associated with a pseudo-cycle a *pseudo–codeword.*

4.2.5 Examples of the Perron–Frobenius theorem

In this section we will present two examples of the Perron–Frobenius theorem for the min–sum semiring on a tail–biting trellis for a convolutional code. The first example will be of a matrix A^n that converges as $n \to \infty$ to a matrix of rank 1. The second example will be of a matrix A^n for which A has no eigenvalues and A^{2n} converges as $n \to \infty$ to a matrix of rank 2.

Example 4.2.1 Consider the matrix product $\mathbf{A} = A_0 A_1 \cdots A_{N-1}$ for a tail–biting trellis of length N, where we define A_i, for $0 \leq i \leq N-1$, to be the incidence matrix of section i of the trellis. If the weight function w is the Hamming weight of the output for each edge, then in the min–sum semiring, the matrix $\mathbf{A}$ represents the lowest weight paths of length N in the trellis. For instance, for the 5–section tail–biting trellis in Figure 4.5 with incidence matrix A in (4.21), we have

$$\mathbf{A} = A^5 = \begin{array}{c} \\ 00 \\ 10 \\ 01 \\ 11 \end{array} \begin{array}{c} \begin{array}{cccc} 00 & 10 & 01 & 11 \end{array} \\ \begin{pmatrix} 0 & 2 & 3 & 3 \\ 3 & 3 & 3 & 3 \\ 2 & 2 & 3 & 3 \\ 3 & 3 & 3 & 3 \end{pmatrix} \end{array},$$

so the lowest non-zero weight codeword in the trellis is weight 3 and starts in the 10, 01 or 11 state.

For $\mathbf{A} = A^N$ as $N \to \infty$, we have

$$\mathbf{A} = A^N = \begin{pmatrix} 0 & 2 & 3 & 3 \\ 3 & 5 & 6 & 6 \\ 2 & 4 & 5 & 5 \\ 3 & 5 & 6 & 6 \end{pmatrix} = \begin{pmatrix} 0 \\ 3 \\ 2 \\ 3 \end{pmatrix} \begin{pmatrix} 0 & 2 & 3 & 3 \end{pmatrix} \text{ for } N > 9.$$

So the encoder in Figure 4.3 will define a tail–biting trellis with a code of minimum distance 5 for all $N > 9$, and every lowest weight codeword goes through the 00 state. The critical cycle is a self loop of weight 0 at the 00 state.

Example 4.2.2 (Anderson and Hladik [40]) Let the weight function w now be defined as the Hamming distance between the edge and the received sequence for that edge. For the 5–section tail–biting trellis in Figure 4.5, consider the received sequence

$$00\ 10\ 10\ 00\ 00,$$

where the ith pair of outputs determines the adjacency matrix A_i. For the matrix product $\mathbf{A}$, we have

$$\mathbf{A} = \begin{array}{c} \\ 00 \\ 10 \\ 01 \\ 11 \end{array} \begin{array}{c} \begin{array}{cccc} 00 & 10 & 01 & 11 \end{array} \\ \begin{pmatrix} 2 & 3 & 2 & 2 \\ 3 & 2 & 3 & 3 \\ 1 & 2 & 3 & 3 \\ 3 & 2 & 3 & 3 \end{pmatrix} \end{array}.$$

Now consider $\mathbf{A}^n$ as $n \to \infty$. We have

$$\mathbf{A}^n = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 2 & 2 & 2 \\ 0 & 1 & 0 & 0 \\ 1 & 2 & 2 & 2 \end{pmatrix} + n\frac{3}{2}J \qquad \text{for } n \text{ even, } n \geq 4$$

and

$$\mathbf{A}^n = \frac{1}{2}\begin{pmatrix} 1 & 3 & 1 & 1 \\ 3 & 5 & 3 & 3 \\ -1 & 1 & 1 & 1 \\ 3 & 5 & 3 & 3 \end{pmatrix} + n\frac{3}{2}J \qquad \text{for } n \text{ odd, } n \geq 5.$$

The two most likely codewords and the most likely pseudo–codeword associated with the received sequence are

1. the all–zero codeword 00 00 00 00 00

2. the nonzero codeword 01 10 10 01 00

3. the pseudo–codeword 00 11 10 00 10 00 10 11 00 00

Codewords 1 and 2 are Hamming distance 2 from the received sequence, while the third pseudo–codeword has a Hamming distance of 3 over 2 cycles. Thus the critical cycle is of degree 2 and $\mathbf{A}^n$ oscillates with period 2. $\mathbf{A}^{2n}$ converges, with eigenvalue 3, although not to the outer product of two eigenvectors but to a matrix of rank 2. It is easy to show that the matrix $\mathbf{A}$ has no eigenvectors or eigenvalues in the min–sum semiring.

4.2.6 Tail–biting codes and pseudo–codewords

In this section we will prove that the iterative min–sum decoder converges to the pseudo–codeword with the minimum average weight per cycle, called the *dominant pseudo–codeword* in [45]. Since the dominant pseudo-codeword is not necessarily a codeword, the MSA may converge to a periodic solution.

Let $\mathbf{y}$ be the received noisy codeword. Assume the dominant pseudo–codeword is unique. We now have the following theorem:

Theorem 4.2.4 *After a finite number of iterations, the decoder locks on to the pseudo-codeword nearest to* $\mathbf{y}$.

(The proof follows almost directly from the min-sum Perron–Frobenius theorem and is given in Appendix A.2.3.)

For an alternative proof using the computation trellis, see [13] or [38, 45]. Since there are a finite number of pseudo–codewords for a given tail–biting trellis, we can use Theorem 4.2.4 to estimate the decoding performance using the familiar union bound argument.

4.2.7 Estimating the error probability with the union bound

Let $\mathbf{x}^0$ be the all zeros codeword. Let $\mathcal{P}$ be the finite set of pseudo-codewords for the trellis, not including $\mathbf{x}^0$. If $\mathbf{x} \in \mathcal{P}$ is an l segment pseudo–codeword, we write $\mathbf{x}$ as

$$\mathbf{x} = \begin{pmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & & \vdots \\ x_{l,1} & x_{l,2} & \cdots & x_{l,n} \end{pmatrix} = \begin{pmatrix} \mathbf{x}^1 \\ \mathbf{x}^2 \\ \vdots \\ \mathbf{x}^l \end{pmatrix},$$

where $\mathbf{x}^i$ denotes the output vector associated with the ith segment of the pseudo–codeword. Without loss of generality we assume the all zeros codeword was transmitted and received as the noisy codeword $\mathbf{y}$.

For a pseudo–codeword $\mathbf{x} \in \mathcal{P}$, define $c(\mathbf{x}^i)$ to be the weight of $\mathbf{x}^i$, for $i \in [l]$, and

$$c(\mathbf{x}) = \frac{1}{l}\sum_{i=1}^{l} c(\mathbf{x}^i) \tag{4.22}$$

to be the average weight of $\mathbf{x}$. From Theorem 4.2.4, we know that a necessary condition for a decoding error is for there to be a minimum average weight path through the trellis, which is closer to $\mathbf{y}$ than $\mathbf{x}^0$, that is, a pseudo–codeword $\mathbf{x} \in \mathcal{P}$, such that $c(\mathbf{x}) \le c(\mathbf{x}^0)$.

Define the error event

$$\varepsilon = \{c(\mathbf{x}) \le c(\mathbf{x}^0)\}$$

for every $\mathbf{x} \in \mathcal{P}$, and the set of error events

$$\mathcal{E} = \bigcup_{\mathbf{x}_j \in \mathcal{P}} \varepsilon_j. \tag{4.23}$$

If we define P_E^{MSA} to be the decoder word error probability, then by the standard union bound, we have

$$P_E^{MSA} = Pr(\mathcal{E}) \le \sum_{\mathbf{x}_j \in \mathcal{P}} Pr(\varepsilon_j). \tag{4.24}$$

For the special case of BPSK modulation on an AWGN channel, with signal–to–noise ratio E_b/N_0, we have

$$\varepsilon = \left\{ \frac{1}{l} \sum_{i=1}^{l} - \log p(\mathbf{y}|\mathbf{x}^i) < - \log p(\mathbf{y}|\mathbf{x}^0) \right\}$$

for the pseudo–codeword $x \in \mathcal{P}$. Note that the weight associated with the ith segment $\mathbf{x}^i$ is the negative log–likelihood of the received vector given $\mathbf{x}^i$.

Let $\mathbf{s}^i$ be the signal vector sent across the channel if we were to transmit segment $\mathbf{x}^i$. For BPSK modulation, a 0 is mapped into $-\sqrt{RE_b}$ and a 1 is mapped into $\sqrt{RE_b}$ where $R = k/n$ is the rate of the code. For the AWGN channel we have

$$p(\mathbf{y}|\mathbf{x}^i) = \frac{1}{(\pi N_o)^{n/2}} \exp\left\{ \frac{1}{N_o} |\mathbf{y} - \mathbf{s}^i|^2 \right\}, \tag{4.25}$$

so

$$\begin{aligned} \varepsilon &= \left\{ \frac{1}{l} \sum_{i=1}^{l} |\mathbf{y} - \mathbf{s}^i|^2 < |\mathbf{y} - \mathbf{s}^0|^2 \right\} \\ &= \left\{ \frac{1}{l} \left(\sum_i \mathbf{s}^i - \mathbf{s}^0 \right)^T \mathbf{y} > \frac{1}{2} \frac{1}{l} \left(\sum_i |\mathbf{s}^i|^2 - |\mathbf{s}^0|^2 \right) \right\}. \end{aligned}$$

Hence the probability of the error event ε is

$$Pr(\varepsilon) = Q\left(\sqrt{\frac{\sum_i |\mathbf{s}^i - \mathbf{s}^0|^2}{2N_o |\sum_i (\mathbf{s}^i - \mathbf{s}^0)|^2}} \right) \tag{4.26}$$

where $Q(t) = 1/\sqrt{2\pi} \int_t^\infty e^{-v^2/2} dv$. We can further simplify (4.26) by noting that $(\mathbf{s}^i - \mathbf{s}^0) = \sqrt{RE_b \sum_j x_{i,j}}$. If we define c_j to be the jth column sum, $c_j = \sum_{i=1}^{l} x_{i,j}$, for $j \in [n]$, then (4.26) becomes

$$Pr(\varepsilon) = Q\left(\sqrt{2R\mathcal{W}(\mathbf{x}) E_b/N_o} \right), \tag{4.27}$$

where we define $\mathcal{W}(\mathbf{x})$ to be the pseudo–weight of the pseudo–codeword $\mathbf{x}$,

$$\mathcal{W}(\mathbf{x}) = \frac{(\sum_j c_j)^2}{\sum_j c_j^2}. \tag{4.28}$$

In [13], Wiberg derives the same expression for the pseudo–weight of a pseudo–codeword, except in the more general context of the computation tree. We shall use this expression in Chapter 6 to determine the performance of the MSA for cycle codes. Forney et al. [49] have also discovered the same expression independently.

Example 4.2.3 The 2–segment pseudo–codeword in Example 4.2.2 has

c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}
0	0	2	1	2	1	0	0	1	0

so that its pseudo–weight is 49/11. Note that a codeword will have a pseudo–weight that is the same as its Hamming weight.

For the iterative min–sum decoding of a tail-biting code, we can apply the union bound from (4.24) to get

$$P_E^{MSA} \leq \sum_{\mathbf{x}_j \in \mathcal{P}} Q\left(\sqrt{2R\mathcal{W}(\mathbf{x}_j)E_b/N_o}\right). \tag{4.29}$$

For an ML decoder for the tail-biting code, we can ignore all non–codeword pseudo-codewords in computing the ML word error probability, P_E^{ML}, so we have the ordinary union bound determined by the code $\mathbb{C}$,

$$P_E^{ML} \leq \sum_{\mathbf{x}_j \in \mathbb{C}} Q\left(\sqrt{2R\mathcal{W}(\mathbf{x}_j)E_b/N_o}\right). \tag{4.30}$$

If the minimum pseudo–weight of a pseudo–codeword in the trellis is greater than the minimum distance of the code, then these two bounds are asymptotically equal.

In the next section we will give two examples, a code for which the minimum pseudo–weight is greater than the minimum distance and one for which the minimum

pseudo–weight is less than the minimum distance.

4.2.8 The (2,1,2) convolutional code and the (24,12,8) Golay code

In general it is not easy to compute the pseudo–weight enumerator for a tail–biting trellis for a given code. In fact in Chapter 5 we can see it is not easy to even count the number of pseudo-codewords of each degree except in special cases.

Example 4.2.4 Consider again the $(2,1,2)$ encoder shown in Figure 4.3. We can form a $(2N, N)$ tail–biting code by pasting together N trellis sections from Figure 4.4b.

We ran the MSA decoder for 8 iterations around the trellis with no stopping rule to check if we converged. Figure 4.6a shows the bit error rate (BER) of the MSA on the AWGN for various N. The improvement in performance from $N = 5$ to 8 to 16 is mainly due to the increase in the minimum distance of the code and not the improved performance of the MSA. For $N > 9$, we showed in Example 4.2.1, that the $(2N, N)$ tail–biting code has a minimum distance of 5, so the performance of the MSA decoder or the ML decoder is the same at high E_b/N_o for all N.

We have computed the pseudo–weight enumerator for $N = 5$ in Table 4.1. The first column of the table is the degree l of the pseudo-codeword, the second column contains the pseudo–weight $\mathcal{W}$ and the third column, the number of pseudo–codeword of weight $\mathcal{W}$, P.W.E. We can see from the table that for $N = 5$ the minimum weight for a codeword is 3, while the minimum pseudo–weight of a non–codeword pseudo–codeword is 4.455. In Figure 4.6b, we have plotted the performance of the MSA decoder and ML decoder for the (10,5) tail–biting code as well as the bit error probability union bound with and without pseudo–codewords. Since the minimum pseudo–weight is greater than the minimum distance of the code, the union bounds are asymptotically the same and the performance of the MSA and ML decoders is too.

The difference in performance between the MSA and ML decoders is in fact very

l	$\mathcal{W}$	P.W.E.	l	$\mathcal{W}$	P.W.E.
1:	0.000	1	3:	4.765	5
	3.000	5		5.000	31
	4.000	5		5.261	40
	5.000	6		5.538	30
	6.000	10		5.828	30
	7.000	5		6.125	20
2:	4.455	25		6.429	35
	4.500	5		6.737	50
	5.000	1		7.049	30
	5.333	25		7.364	10
	5.400	20		7.681	25
	6.231	10		8.000	30
	6.250	30		8.643	5
	6.368	10		8.966	10
	7.200	10		9.615	1
	7.348	20	4:	5.444	55
	8.048	5		6.231	80
	8.167	5		7.118	70
	8.333	5		8.048	40
	9.143	5		9.000	11

Table 4.1: The pseudo-weights for the (10,5,3) tail–biting code with $m = 2$.

small for this entire family of codes and is shown separately in Figure 4.6c. Here we have counted an error when the MSA decoder makes a different decision to the ML decoder, which we denote the MSA error rate (MSAER). Even though there is not a visible difference in BER performance for $N = 16$ and 32, we can see that for $N = 32$, the MSA performs much more like an ML decoder. Figure 4.6d shows a plot of the MSAER for the (10,5) tail–biting code and the MSA union bound where we now considered only non–codeword pseudo–codewords, i.e., $\mathcal{P} - \mathbb{C}$. We can see this bound is not as tight as the union bound in (4.29).

We can show that any tail–biting code consisting of N concatenated trellis sections from Figure 4.4b will have a minimum pseudo-weight that is greater than or equal to the minimum distance of the code. In order to do this, we need the following two results, which are proved in Appendix A.2.4.

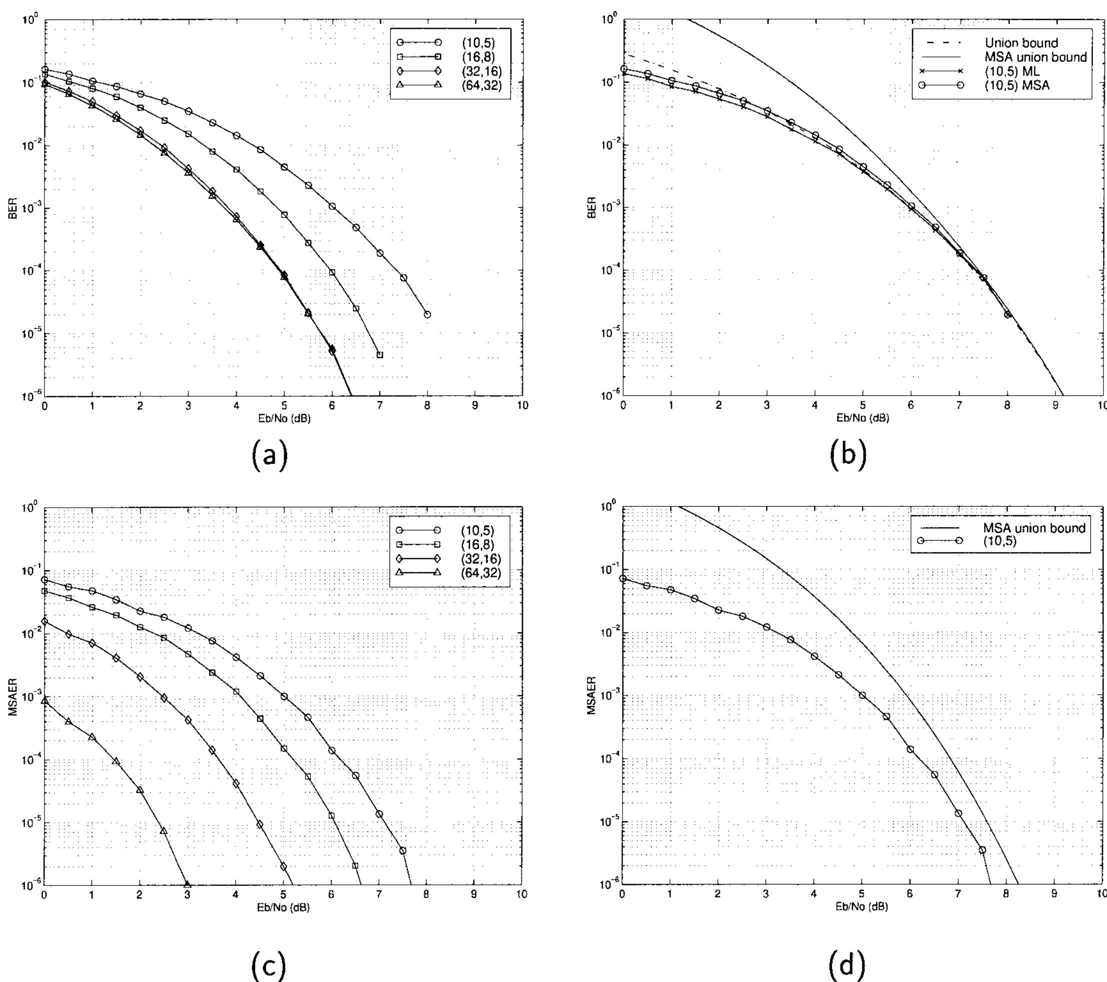


Figure 4.6: Plots of the performance of the $(2N, N)$ tail–biting code based on the convolutional encoder in Figure 4.3 for various N, including the union bounds from (4.29) and (4.30).

Lemma 4.2.5 *The pseudo–weight on any pseudo–codeword of degree 2 is at least the minimum distance of the code.*

Lemma 4.2.6 *Consider a time invariant trellis of length N. If there are α edges of weight 0 in a section of the trellis, then for sufficiently large N, every pseudo–codeword of degree $l > \alpha$ will have a pseudo–weight at least the minimum distance of the code.*

For the tail-biting code consisting of N concatenated trellis sections from Figure 4.4b, we have $\alpha = 2$. By Lemma 4.2.6, for $N \geq 15$ and $N \geq 10$, there are no pseudo–

codewords of degree 3 and 4 respectively, with pseudo–weight less than 5. By brute force enumeration we have found that for smaller N, there are no low weight pseudo–codewords of degree 3 or 4 either. Since $S_{min} = 4$, by Lemma 4.2.5, the minimum pseudo–weight for a tail–biting code, based on the (2,1,2) convolutional code in Figure 4.3, is then at least the minimum distance of the code. Thus for this family of codes the MSA and ML decoders will have the same asymptotic performance.

There are many tail–biting codes for which the minimum weight pseudo–codeword is at least the minimum distance of the code, for example the tail–biting trellis for the (8,4,4) Hamming code in [21, 22]. However, Kötter and Vardy have constructed a class of tail–biting trellises whose minimum pseudo–weight is strictly less than the minimum distance of the code for a pseudo–codeword of degree 3 or more [50]. In the next example we present a tail–biting trellis representation, for the (24,12,8) extended Golay code, for which there is a pseudo-codeword of degree 4, whose pseudo–weight is less than 8. The poor performance of the MSA decoder for this tail–biting code was first noticed by Anderson and Tepe [51].

Example 4.2.5 Consider the time invariant tail–biting trellis of the (24,12,8) extended Golay code with generators (414,730)[1]. There is a pseudo-codeword of degree 4 whose pseudo–weight of 7.36. We can write the pseudo–codeword as the input sequence

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

where if we write the input as a sequence $u_0, u_1, \ldots u_{47}$, then the state of the encoder at time i is the input string $u_{i-1}, u_{i-2}, \ldots, u_{i-6}$ (where we perform index arithmetic

[1]The encoder is defined by two sets of shift register taps which we express as left–justified octals. The tap set will be of length $m + 1$ for a memory m encoder. For further details we refer the reader to [52].

modulo 48). The input sequence results in the corresponding output sequence

$$\begin{array}{cccccccccccc} 10 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & 11 \\ 10 & 00 & 10 & 00 & 00 & 00 & 01 & 01 & 00 & 00 & 00 & 10 \\ 10 & 10 & 00 & 00 & 00 & 00 & 01 & 00 & 00 & 00 & 00 & 01 \\ 10 & 01 & 01 & 00 & 00 & 00 & 01 & 00 & 01 & 00 & 00 & 01 \end{array}$$

which has pseudo–weight 324/44.

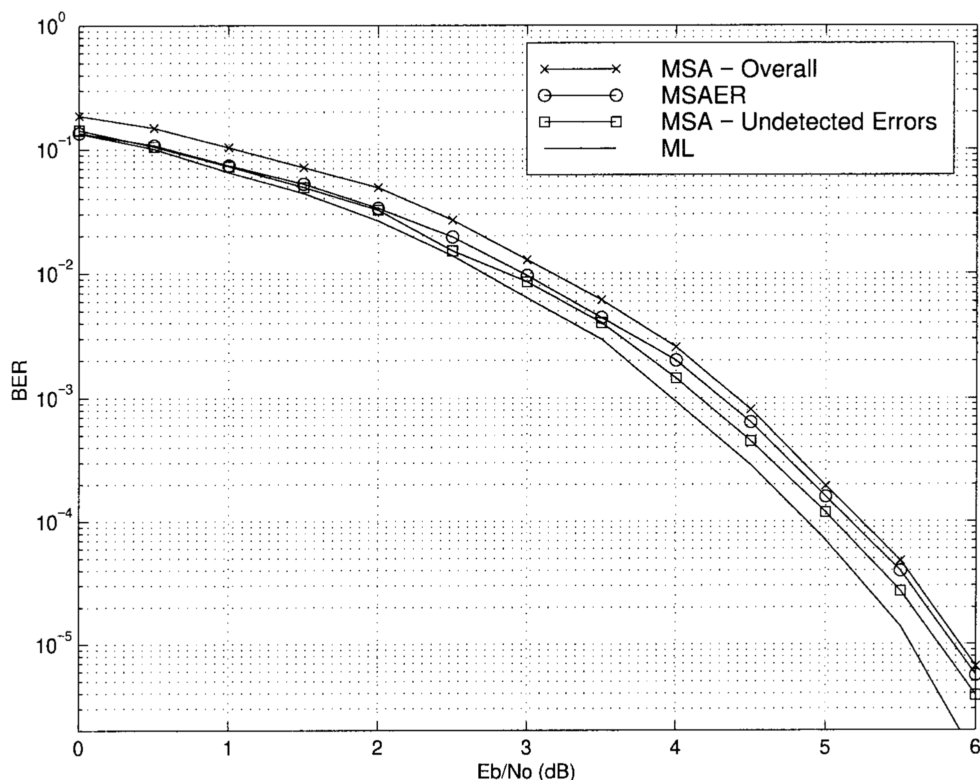


Figure 4.7: The performance of the MSA decoder for the (24,12,8) tail–biting extended Golay code on an AWGN channel with a maximum of 30 iterations.

We ran the MSA decoder for up to 30 iterations around the trellis where we stopped after 3 or more iterations if we decided on the same state twice in a row. In Figure 4.7, the top curve shows the performance of the MSA decoder. The middle two curves show the undetected error rate for the MSA, i.e., when the MSA converges to the incorrect codeword, and the error rate with respect to an ML decoder, i.e.,

when the MSA makes a non-maximum likelihood decision. The bottom curve shows the performance of an ML decoder. The difference between the performance of the MSA and ML decoders is about 0.5 dB. We can see that unlike in Example 4.2.4, the MSA decoder makes a significant number of non–ML decisions. This is due to the existence of low weight pseudo–codewords. The optimal, time varying 16–state trellis for the extended Golay code described in [35] has MSA decoder performance that is almost as good as maximum likelihood, as shown in [21].

4.3 Conclusions

The iterative min–sum algorithm either converges to the maximum likelihood decision or to a periodic solution on the single cycle graph. Like the iterative sum–product algorithm, the MSA will always make an ML decision for binary valued hidden variables and may make an incorrect decision in the non–binary case. We can gain a much better understanding of the behavior of the MSA on a single cycle graph if we look at a tail–biting trellis. We used the trellis to develop a union bound argument for the decoding performance of the MSA so we can determine when an MSA decoder will perform asymptotically as well as an ML decoder. We can also select the appropriate tail–biting trellis representation of the code to improve the performance of the MSA decoder.

Chapter 5

Counting Pseudo–Codewords for Tail–Biting Convolutional Codes

In this chapter, the problem we would like to solve is to count the number of pseudo–codewords of degree l that can be found in a tail–biting trellis for a convolutional code.

In Section 5.1, we introduce the transfer matrix method, which we will use to count pseudo–codewords. In Section 5.2, we introduce the class of convolutional codes for which we will count the number of pseudo–codewords. In Section 5.3, we will construct the pseudo–codeword state diagram for the trellis and present two techniques to reduce its complexity. In Section 5.4, we will present some examples of how we count the number of pseudo–codewords in the trellis for some tractable cases and in Section 5.5 we present our conclusions. In this chapter all multiplication and addition is performed in the sum–product semiring.

5.1 The Transfer Matrix Method

In this section, we continue with the definitions and notation of Section 4.2.1 and introduce the rest of the necessary algebraic and combinatorial tools required to count pseudo–codewords. Much of this material is taken directly from Stanley [46, Section 4.7] and McEliece [48].

Let D be a weighted digraph with incidence matrix A, and let $A_{ij}(n)$ denote the sum of all paths of length n in the trellis from vertex i to vertex j. We define the

generating function $F_{i,j}$ to be

$$F_{i,j}(z) = \sum_{n \geq 0} A_{ij}(n) z^n$$

and we have the following result from Stanley [46, Theorem 4.7.2].

Theorem 5.1.1 *The generating function $F_{i,j}(z)$ is given by*

$$F_{i,j}(z) = [(I - zA)^{-1}]_{i,j}. \tag{5.1}$$

(The proof is given in Appendix A.3.1.)

Now define the generating function

$$W(z) = \sum_{n \geq 1} w_n z^n, \tag{5.2}$$

where w_n is the sum of the weights of all closed walks of length n in D. For example, $w_1 = \mathrm{tr}(A)$, where tr denotes the trace function. We conclude this section with one final result from Stanley [46, Corollary 4.7.3] and an example.

Theorem 5.1.2 (The transfer-function theorem) *The generating function $W(z)$ is given by*

$$W(z) = -\frac{zQ'(z)}{Q(z)}, \tag{5.3}$$

where $Q(z) = det(I - zA)$.

(The proof is given in Appendix A.3.2.)

For example, for Figure 4.1, with incidence matrix A in (4.12), we have

$$Q(z) = 1 - z^2 - z^3 - z^4$$

and

$$W(z) = \frac{z^2(2 + 3z + 4z^2)}{1 - z^2 - z^3 - z^4}.$$

We can also write $W(z)$ as the series,

$$W(z) = 2z^2 + 3z^3 + 6z^4 + 5z^5 + 11z^6 + \text{ higher order terms.}$$

Note that the number of closed paths w_N satisfies the recursion

$$w_N = w_{N-2} + w_{N-3} + w_{N-4},$$

with initial conditions

$$w_1 = 0,\ w_2 = 2,\ w_3 = 3,\ w_4 = 4.$$

According to $W(z)$ there are 6 closed paths of length 4 in Figure 4.1 which agrees with $\mathrm{tr}(A^4)$ in (4.13).

5.2 Convolutional Codes, Tail-biting Trellises and Pseudo–Cycles

Since we are only interested in counting the number of pseudo–codewords in the tail–biting trellis, and not interested in computing the actual pseudo–weights, we will ignore the output matrices $\mathcal{C}$ and $\mathcal{D}$ in (4.18) for a moment and restrict ourselves to convolutional encoders of the form shown in Figure 5.1. The encoder will now have matrices $\mathcal{A}$ and $\mathcal{B}$ of the form

$$\mathcal{A} = \begin{pmatrix} 0 & \cdots & 0 & c_0 \\ & I_{m-1} & & \vdots \\ & & & c_{m-1} \end{pmatrix}, \quad \mathcal{B} = \begin{pmatrix} 1 & 0 & \cdots & 0 \end{pmatrix},$$

where I_{m-1} is the $m-1 \times m-1$ identity matrix and c_j is 0 (or 1) depending on whether the feedback switch for $s_{i,j}$ is open (or closed) for $0 \leq j \leq m-1$.

For a convolutional encoder of this form, if we write the state at time i as the

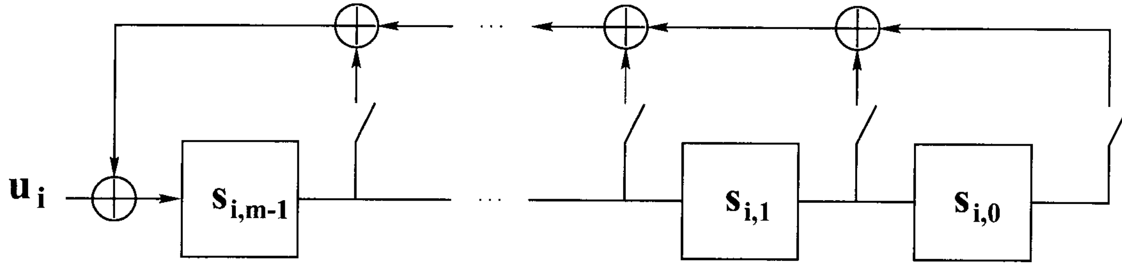


Figure 5.1: A constraint length m convolutional encoder with feedback (shown without any output taps).

vector

$$\mathbf{s_i} = (s_{i,0}, s_{i,1}, \ldots, s_{i,m-1}), \tag{5.4}$$

then the state at time $i+1$ will be

$$\begin{aligned} \mathbf{s_{i+1}} &= (s_{i+1,0}, s_{i+1,1}, \ldots, s_{i+1,m-2}, s_{i+1,m-1}) \\ &= (s_{i,1}, s_{i,2}, \ldots, s_{i,m-1}, s_{i+1,m-1}). \end{aligned}$$

Thus if the initial state is written as

$$\mathbf{s_0} = (s_{-m}, s_{-m+1}, \ldots, s_{-1}), \tag{5.5}$$

we can represent the sequence of states for N time steps as a string of length $N+m$,

$$\mathbf{s} = s_{-m} \ldots s_{-1} s_0 s_1 \ldots s_{N-1}, \tag{5.6}$$

where the state at time i is

$$\mathbf{s_i} = (s_{i-m}, s_{i-m+1}, \ldots, s_{i-1}), \tag{5.7}$$

for $0 \leq i \leq N-1$.

If the sequence is a cycle on the tail-biting trellis of length N, then the initial

state $\mathbf{s_0}$ must be

$$\mathbf{s_0} = (s_{N-m}, \ldots, s_{N-1}) \tag{5.8}$$

and we can represent the sequence of states, simply as the string

$$\mathbf{s} = s_0 s_1 \ldots s_{N-1}. \tag{5.9}$$

The state of the encoder at time index i will be the same as in (5.7) where we take indices modulo N.

We will consider two sequences $\mathbf{s}_1$ and $\mathbf{s}_2$ of length lN to be the same pseudo–codeword, if $s_{1,i} = s_{2,i+jN}$ for $0 \leq i \leq lN - 1$ and some $j \in \mathbb{Z}$, since it is the same sequence shifted in time by a multiple of N. A pseudo–codeword will thus be independent of shifts of multiples of N in the time axis.

Let $w_{N,l}$ denote the number of pseudo–codewords of degree l on a tail–biting trellis of length N. We would like to find a closed form expression for the generating function

$$W_l(z) = \sum_{N \geq 1} w_{N,l} z^N.$$

We will do this using the transfer-function theorem from Section 5.1 and the pseudo–codeword state diagrams which we will develop in the next section.

For $l = 1$, we can also find the weight enumerator for the code. Since the weight is a meaningless metric for pseudo–codewords of degree greater than 1, we refer the reader to [48] for further details.

5.3 Pseudo–Codeword State Diagrams

Consider an (n, k, m) convolutional code with a tail–biting trellis of length N and a pseudo–codeword of degree l. We can represent the pseudo–codeword as a sequence of lN states $\mathbf{s_0}, \ldots, \mathbf{s_{lN-1}}$. If we partition the states into N sets

$$\mathbf{S_0} = (\mathbf{s_0}, \mathbf{s_N}, \ldots, \mathbf{s_{(l-1)N}})$$

$$\mathbf{S_1} = (\mathbf{s_1}, \mathbf{s_{N+1}}, \ldots, \mathbf{s_{(l-1)N+1}})$$
$$\vdots$$
$$\mathbf{S_{N-1}} = (\mathbf{s_{N-1}}, \mathbf{s_{2N-1}}, \ldots, \mathbf{s_{lN-1}}),$$

then the l states in $\mathbf{S_i}$, for $0 \leq i \leq N$, must be different for the sequence to be a pseudo–codeword. We call the set $\mathbf{S_i}$ the *pseudo–codeword state* at time i.

We can construct a *pseudo–codeword state diagram* consisting of $(2^m)!/(2^m - l)!$ states, corresponding to all the possible pseudo–codeword states. There is an edge from pseudo–codeword state $\mathbf{S_i}$ to pseudo–codeword state $\mathbf{S_j}$, if for each state in $\mathbf{S_i}$, there is a transition in the state diagram for the encoder to the corresponding state in $\mathbf{S_j}$. A pseudo–codeword of length lN can now be written as a path $\mathbf{S_0}, \ldots, \mathbf{S_N}$, where

$$\mathbf{S_N} = (\mathbf{s_N}, \ldots, \mathbf{s_{(l-1)N}}, \mathbf{s_0}),$$

, i.e., $\mathbf{S_N}$ is left cyclic shift of $\mathbf{S_0}$.

If we construct an incidence matrix for the pseudo–codeword state diagram, then we can use Theorem 5.1.1, to count the number of pseudo–codewords of length N for each possible initial state $\mathbf{S_0}$. The sum divided by l (since shifts of N are the same pseudo–codeword) will then be the generating function $W_l(z)$. Unfortunately, this method is impractical except when l and m are very small. In the following sections we present two techniques to reduce the complexity of the pseudo–codeword state diagram.

5.3.1 Pseudo–codewords where the first N bits are zero

If we restrict ourselves to the convolutional encoders of the form in Figure 5.1, then we can write the pseudo–codeword as a string of lN bits $\mathbf{s} = s_0 s_1 \ldots s_{lN-1}$ where the string of m bits $s_{i-m} \ldots s_{i-1}$ is the state of the memory at time index i. By convention we will always take the indices modulo lN for a pseudo–codeword of degree l.

We can write the bits of $\mathbf{s}$ in a ring. If we take l windows of width m and place them on the ring, at intervals N bits apart, then the l windows should each contain

a different sequence of length m for every position of the windows on the ring. In fact, we only need to rotate the windows along any N bit interval to ensure that $\mathbf{s}$ is a pseudo–codeword.

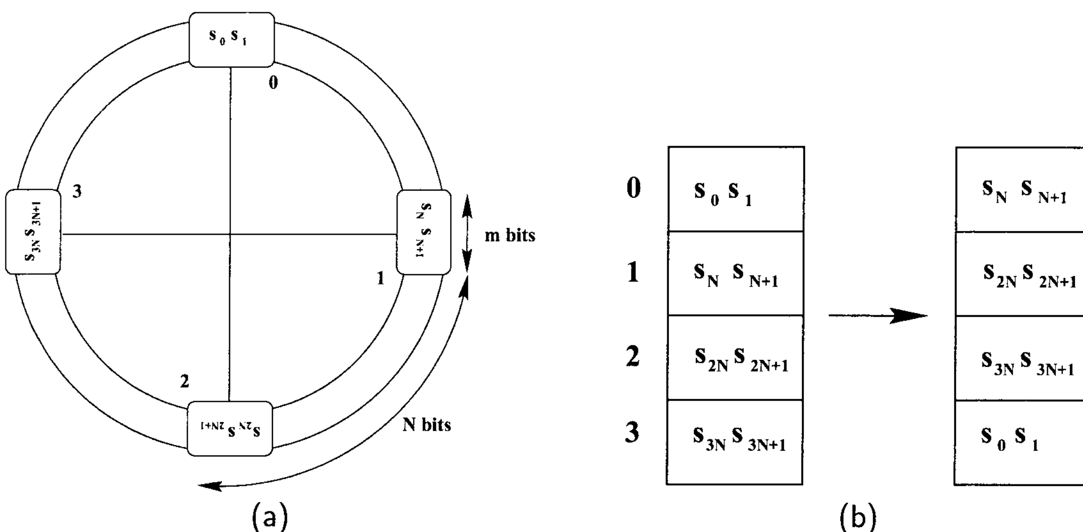


Figure 5.2: (a) The pseudo–codeword s with $l = 4$ and $m = 2$ written in a ring. (b) The initial contents of the 4 windows and the contents after the windows have been rotated clockwise by N bits.

For example, Figure 5.2a shows a pseudo–codeword of degree 4, written on a ring for a convolutional code with memory $m = 2$. Since $m = 2$, the encoder only has four states, namely $\{00, 01, 10, 11\}$, so each state must appear in one of the windows at each shift, for $\mathbf{s}$ to be a pseudo–codeword.

If we label the windows 0 to $l-1$, then after shifting N bits in a clockwise direction, the bit string in window i will be the initial contents of window $i + 1$. For example, Figure 5.2b shows the contents of the windows from Figure 5.2a before and after a clockwise rotation of N bits. If we let the contents of the l windows represent the state of the pseudo–codeword, then we can construct a pseudo–codeword state diagram in the same manner described above.

A pseudo–codeword is represented by a path of length N in this state diagram. Since the contents of window i will be the previous contents of window $i + 1$ after N shifts and the windows are of size m, the last m transitions in the path will be

predetermined by the starting state of the pseudo–codeword.

Define the sequence $\mathbf{y}$ with elements, $y_i = s_{i\,mod\,N}$ for $0 \leq i \leq lN - 1$, so $\mathbf{y}$ is l copies of the first N bits of $\mathbf{s}$ pasted together. Now define the sequence $\mathbf{t} = \mathbf{s} \oplus \mathbf{y}$, where $\oplus$ is addition modulo 2. Clearly $\mathbf{t}$ is also a pseudo–codeword, where the first N bits are now equal to zero.

Let $\mathbf{S}$ be the set of pseudo–codewords and $\mathbf{T}$ be the set of pseudo–codewords whose first N bits are zero, so $\mathbf{T} \subset \mathbf{S}$. From the previous section we defined $w_{N,l}$ as the number of sequences that are pseudo–codewords, i.e., $w_{N,l} = |\mathbf{S}|$. Now we let $v_{N,l} = |\mathbf{T}|$ and define the generating function

$$V_l(z) = \sum_{N \geq 1} v_{N,l} z^N.$$

We then have the following theorem.

Theorem 5.3.1 *We have*

$$W_l(z) = \frac{1}{l} V_l(2z).$$

(The proof is given in Appendix A.3.3.)

We can construct the pseudo–codeword state diagram for pseudo–codewords that start with N zeros in much the same way as we do for ordinary pseudo–codewords except we now only have $(2^m - 1)!/(2^m - l)!$ states. Thus we have reduced the number of states by a factor of 2^m.

However, we have to be careful when counting the number of paths of length N. If we start each path at $\mathbf{s_m}$, (so that we are guaranteed all zeros in the 0 window), then the last $N - m$ steps in the pseudo–codeword path may not be represented by states in the reduced pseudo–codeword state diagram. This is because we now can have nonzero entries in the 0 window. But the last $N - m$ steps in a pseudo–codeword path are predetermined for a given initial state, so we only need to count the paths of length $N - m$, that will lead to a state from which the final m fixed steps are possible. We can do this for each starting state using Theorem 5.1.1 and sum the results to get $V_l(z)/z^m$.

The complexity of the pseudo–codeword state diagram is still quite prohibitive so we will introduce one more technique to reduce the number of states and edges further before we present some examples.

5.3.2 Reducing the size of the windows to $m-1$

Given a convolutional encoder of the form in Figure 5.1, the set of transitions between state $\mathbf{s_i}$ and $\mathbf{s_{i+1}}$ is the same for $s_{i,0}=0$, or 1. Therefore, two pseudo–codeword states $\mathbf{S_i}$ and $\mathbf{S_j}$, which share the same last $m-1$ bits in every window will have the same set of transitions to the next state, i.e.,

$$\{\text{fin}(e) : e \in E, \text{init}(e) = \mathbf{S_i}\} = \{\text{fin}(e) : e \in E, \text{init}(e) = \mathbf{S_j}\},$$

where E is the set of edges in the state diagram.

Therefore, we can construct a new pseudo–codeword state diagram from the initial pseudo–codeword state diagram by combining all states which share the same last $m-1$ bits in every window and combining all multiple edges between two states. The new pseudo–codeword state diagram will thus have 2^l fewer states and edges than the original.

Let $u_{N,l}$ to be the number of paths of length N in the new pseudo–codeword state diagram and define the generating function

$$U_l(z) = \sum_{N \geq 1} u_{N,l} z^N.$$

We then have the following theorem.

Theorem 5.3.2 *We have*

$$W_l(z) = U_l(z).$$

(The proof is given in Appendix A.3.4.)

We can combine the technique of this section with the technique of Section 5.3.1 to produce a pseudo–codeword state diagram of even lower complexity. We call this

pseudo–codeword state diagram the *reduced pseudo–codeword state diagram.* The reduced pseudo–codeword state diagram will contain

$$(2^{m-1})^{l-1} - \binom{l-1}{2}(2^{m-1})^{l-3} - \sum_{i=3}^{l-1}\binom{l-1}{i}(2^{m-1})^{l-1-i}$$

states and

$$\frac{(2^m-1)!}{(2^m-l)!}$$

edges.

Note that the number of states is not $(2^m-1)!/(2^m-l)!(2^{l-1})$ since there are not necessarily 2^{l-1} states which share the same last $m-1$ bits in every window in the pseudo–codeword state diagram for pseudo–codewords where the first N bits are zero. The number of states is therefore the number of sets of $l-1$ windows of size $m-1$, in which no two windows contain the all-zeros state and no three or more windows contain the same state.

In the next section we will present some examples of counting pseudo–codewords for various m and l.

5.4 Examples of Counting Pseudo–Codewords for Small m and l

In this section, we will use the techniques of Section 5.3 to count pseudo–codewords. In Section 5.4.1, we will set the first N bits of the pseudo–codeword to zero, as in Section 5.3.1, in order to count the number of codewords and degree 2 pseudo–codewords. In Section 5.4.2, we will use the reduced pseudo–codeword state diagram of Section 5.3.2, to count the number of pseudo–codewords of degree 3, for $m = 2$ and 3, and the number of pseudo–codewords of degree 4, for $m = 2$.

5.4.1 Codewords and pseudo–codewords of degree 2

We will first demonstrate how to count the number of codewords and pseudo–codewords with zero for the first N bits.

We start by counting the number of codewords in a tail–biting trellis. Since a codeword is N bits, the only sequence in $\mathbf{T}$ is the all zeros sequence. Therefore, $v_{N,l} = 1$ and is independent of the memory m of the convolutional encoder, or the length of the trellis N. By Theorem 5.3.1 we have

$$\begin{aligned} W_1(z) &= \sum_{N\geq 1} 2^N z^N \\ &= \frac{2z}{1-2z}, \end{aligned}$$

so there are 2^N possible codewords for a tail–biting trellis of length N.

A more interesting example is counting the pseudo–codewords of degree 2. If we write a string of length $2N$ on a ring, then we need 2 windows to check if it is a pseudo–codeword. If the first window is positioned somewhere over the first N bits, then the second window can contain any string except a string of m zeros. We can simply delete the first N bits from the string and only consider strings of N bits on a ring, which don't contain a substring of m or more zeros. For $m = 2$ we will show that the number of such strings satisfies the same recursion as that of the Fibonacci numbers, although will different initial conditions.

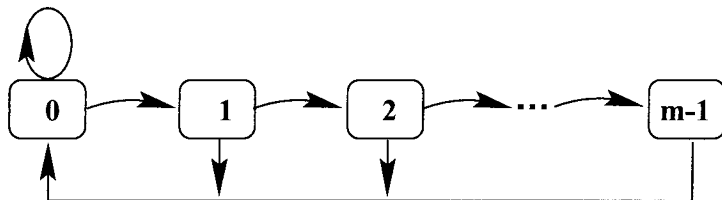


Figure 5.3: The state diagram for a device which generates any string that contains no substring of m or more zeros.

Figure 5.3 shows the state diagram for a finite state machine that can generate any string that contains no substring of m or more zeros. State i indicates that the

last i bits that have occurred in the sequence are 0. Thus a 1 causes the machine to return to state 0 and a 0 causes the machine to go from state i to state $i+1$. If the machine is in state $m-1$, then the next bit must be a 1 so the machine can only go to state 0. The number of strings of length N, which wrap around and have no string of m consecutive zeros, is then the number of closed paths of length N in the state diagram.

The incidence matrix A, for the state diagram, will be

$$A = \begin{array}{c} \\ 0 \\ 1 \\ \vdots \\ m-2 \\ m-1 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & \cdots & m-1 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & \cdots & 0 \\ 1 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & 0 & \cdots & 1 \\ 1 & 0 & 0 & \cdots & 0 \end{pmatrix} \end{array}.$$

So by Theorem 5.1.2, we have

$$\begin{aligned} V_2(z) &= \frac{-zQ'(z)}{Q(z)} \\ &= \frac{z + 2z^2 + \cdots + mz^m}{1 - z - z^2 \cdots - z^m}. \end{aligned}$$

For $m = 2$, we have

$$v_{N,2} = v_{N-1,2} + v_{N-2,2},$$

which is the Fibonacci recursion.

By Theorem 5.3.1, we have

$$W_2(z) = \frac{1}{2} \frac{2z + 2(2z)^2 + \cdots + m(2z)^m}{1 - 2z - (2z)^2 - \cdots - (2z)^m}. \tag{5.10}$$

So the number of pseudo–codewords of degree 2, $w_{N,2}$ satisfies the recursion

$$w_{N,2} = 2w_{N-1,2} + \cdots + 2^m w_{N-m,2}. \tag{5.11}$$

$m\backslash N$	1	2	3	4	5	6	7	8	9
2	1	6	16	56	176	576	1856	6016	19456
3	1	6	28	88	336	1248	4544	16768	61696
4	1	6	28	120	416	1632	6336	24448	93952
5	1	6	28	120	496	1824	7232	28544	112384
6	1	6	28	120	496	2016	7680	30592	121600

Table 5.1: The number of pseudo–codewords $w_{N,2}$ for various m.

For example, for the $(2,1,2)$ convolutional encoder in Figure 4.3, we have

$$w_{N,2} = 2w_{N-1,2} + 4w_{N-2,2},$$

with initial conditions

$$w_{1,2} = 1, \; w_{2,2} = 6.$$

It is interesting to note that for a convolutional encoder with memory m, the number of pseudo–codewords of degree 2 can be expressed by a recursion of order m. Table 5.1 shows the number of pseudo–codewords of degree 2, for $m = 2, 3, \ldots, 6$ for various N.

5.4.2 Pseudo–codewords of degree 3 or more

We use the reduced pseudo–codeword state diagram to count the number of pseudo–codewords of degree 3 or more. We will start with the simplest case, namely $l = 3$ and $m = 2$.

The pseudo–codeword state diagram for pseudo–codewords of degree 3 and a convolutional encoder of memory 2 will have 24 states. If we only count pseudo–codewords with the first N bits zero, then we reduce the number of states and edges to 6 and 12 respectively, as shown in Figure 5.4a. By convention we label each state with the contents of windows 1 thru $l-1$ since the 0 window always contains the all-zeros string. Note that for the states a and b, the last $m-1$ bits in windows 2 and 3 are the same. This is also true for states c and d and states e and f.

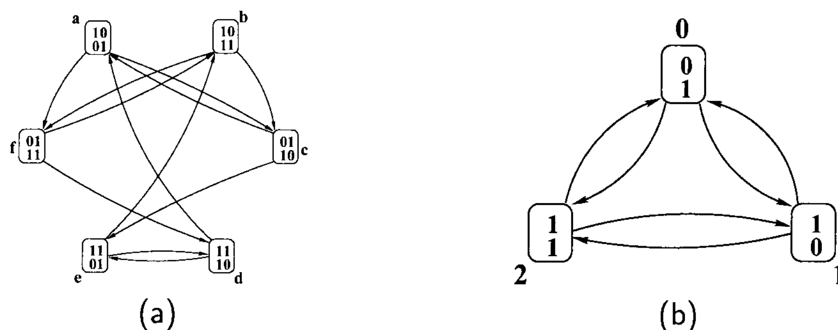


Figure 5.4: (a) The pseudo–codeword state diagram for pseudo–codewords with the first N bits zero and (b) the reduced pseudo–codeword state diagram, for $m = 2$ and $l = 3$.

The incidence matrix A for Figure 5.4a is

$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array} \begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array} \tag{5.12}$$

and

$$Q(z) = 1 - 3z^2 - 2z^3. \tag{5.13}$$

Since the transitions from state a are the same as those from state b, the first and second rows of A are the same. If we want to count the number of pseudo–codewords using (5.12), then for each state we must find the set of states from which the final state can be reached in m steps. For example, state a has final state

$$\begin{pmatrix} 01 \\ 10 \\ 00 \end{pmatrix}$$

which can be reached in 2 steps only from states a, b, c and d. This is because, if we

were in states e or f, the determined transitions force windows 2 and 3 to have the same contents after 1 step and we no longer have a pseudo–codeword.

We define R to be the matrix where the (i, j)-entry is a 1 if the final state of a pseudo–codeword with initial state i can be reached from state j in m steps without violating the property that no two windows can contain the same sequence of m bits. Otherwise the (i, j)-entry is a 0. For Figure 5.4a, we have

$$R = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array} \begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array}. \tag{5.14}$$

From Theorem 5.1.1, we can sum over all the entries of $[I - zA]^{-1}$ for which the corresponding entry of R is a 1. The result is

$$V_3(z) = \frac{3z^2 + 6z^3}{1 - z - 2z^2}, \tag{5.15}$$

where we have multiplied $V_3(z)$ by z^2 since a path of length $N - 2$ in the pseudo–codeword state diagram, corresponds to pseudo–codeword of length N. From Theorem 5.3.1 we have

$$W_3(z) = \frac{4z^2 + 16z^3}{1 - 2z - 8z^2}. \tag{5.16}$$

We now demonstrate the same result using the reduced pseudo–codeword state diagram for $l = 3$ and $m = 2$ shown in Figure 5.4b. As we can see states a and b, c and d, and e and f have been combined into states 0, 1 and 2 respectively. So the state diagram now has 3 states and 6 edges. The incidence matrix A for Figure 5.4b

is now

$$A = \begin{array}{c} \\ 0 \\ 1 \\ 2 \end{array} \begin{array}{c} \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \end{array} \tag{5.17}$$

and

$$Q(z) = 1 - 3z^2 - 2z^3, \tag{5.18}$$

which is the same as in (5.13) and

$$R = \begin{array}{c} \\ 0 \\ 1 \\ 2 \end{array} \begin{array}{c} \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \end{array}. \tag{5.19}$$

From Theorem 5.1.1, we have

$$V_3(z) = \frac{3z}{1 - z - 2z^2} \tag{5.20}$$

and

$$W_3(z) = \frac{2z}{1 - 2z - 8z^2}, \tag{5.21}$$

where we have multiplied $V_3(z)$ by z since a path of length $N-1$ in the pseudo–codeword state diagram, now corresponds to a pseudo–codeword of length N. The coefficients of the power series defined by $W_3(z)$ are the same for (5.16) and (5.21), except that (5.16) does not include the coefficient of z. This is because the pseudo–codeword state diagram for pseudo–codewords that start with N zeros, can only count the number of pseudo–codewords for $N \geq m$ whereas the reduced pseudo–codeword state diagram counts the number of pseudo–codewords for $N \geq m-1$. It is simple enough to compute the number of pseudo–codewords for $N < m$ using brute force enumeration.

It follows from (5.16) and (5.21) that the number of pseudo–codewords of degree

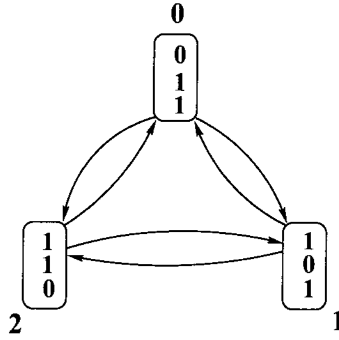


Figure 5.5: The reduced pseudo–codeword state diagram for $m = 2$ and $l = 4$.

3, $w_{N,3}$, satisfies the recursion

$$w_{N,3} = 2w_{N-1,3} + 8w_{N-2,3}, \tag{5.22}$$

with initial conditions

$$w_{1,3} = 2, \; w_{2,3} = 4.$$

For $m = 2$ and $l = 4$, we have the reduced pseudo–codeword state diagram shown in Figure 5.5. The regular pseudo–codeword state diagram contains 24 states whereas Figure 5.5 contains only 3 states and 6 edges.

The reduced pseudo–codeword state diagram has incidence matrix,

$$A = \begin{array}{c} \\ 0 \\ 1 \\ 2 \end{array} \begin{array}{c} \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \end{array} \tag{5.23}$$

and

$$R = \begin{array}{c} \\ 0 \\ 1 \\ 2 \end{array} \begin{array}{c} \begin{array}{ccc} 0 & 1 & 2 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \end{array}. \tag{5.24}$$

So we have

$$V_4(z) = \frac{2z}{1-2z} \tag{5.25}$$

and

$$W_4(z) = \frac{z}{1-4z}, \tag{5.26}$$

where once again we have multiplied $V_4(z)$ by z. It follows from (5.26) that the number of pseudo–codewords of degree 4, $w_{N,4}$ satisfies the recursion

$$w_{N,4} = 4w_{N-1,4}, \tag{5.27}$$

with initial condition

$$w_{1,4} = 1.$$

Table 5.2 shows the number of pseudo–codewords of degree 1, 2, 3 and 4 for $m = 2$.

$l \backslash N$	1	2	3	4	5	6	7	8	9
1	2	4	8	16	32	64	128	256	512
2	1	6	16	56	176	576	1856	6016	19456
3	2	4	24	80	352	1344	5504	21760	87552
4	1	4	16	64	256	1024	4096	16384	65536

Table 5.2: The number of pseudo–codewords $w_{N,l}$ for $m = 2$.

Note that for $m = 2$, $w_{N,l}$ always satisfies a recursion of order m or less. Therefore, even though we have a large number of states in the reduced pseudo–codeword state diagram, we had hoped that we could always find a simple recursion of size at most m to count the number of pseudo–codewords of degree $l > 2$. In our final example we show that unfortunately this is not always the case.

For $l = 3$ and $m = 3$ the pseudo–codeword state diagram will contain 336 states while the reduced pseudo–codeword state diagram shown in Figure 5.6 has 15 states and 42 edges. The incidence matrix A, for the reduced pseudo–codeword state dia-

gram, will be

$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \\ i \\ j \\ k \\ l \\ m \\ n \\ o \end{array} \begin{array}{c} \begin{array}{ccccccccccccccc} a & b & c & d & e & f & g & h & i & j & k & l & m & n & o \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}. \qquad (5.28)$$

and

$$R = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \\ i \\ j \\ k \\ l \\ m \\ n \\ o \end{array} \begin{array}{c} \begin{array}{ccccccccccccccc} a & b & c & d & e & f & g & h & i & j & k & l & m & n & o \end{array} \\ \left(\begin{array}{ccccccccccccccc}
0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0
\end{array}\right) \end{array}. \qquad (5.29)$$

So we have

$$V_3(z) = \frac{3z^2(3 + 6z + 13z^2 + 6z^3 + 2z^4 - 8z^5)}{(1 + z + z^2 - z^3)(1 - 2z - 2z^2 - 4z^3)} \qquad (5.30)$$

and

$$W_3(z) = \frac{4z^2(3 + 12z + 52z^2 + 48z^3 + 32z^4 - 128z^5)}{(1 + 2z + 4z^2 - 8z^3)(1 - 4z - 8z^2 - 32z^3)}, \qquad (5.31)$$

where once again we have multiplied $V_3(z)$ by z^2.

It follows from (5.31) that the number of pseudo–codewords of degree 3, $w_{N,3}$ satisfies the recursion

$$\begin{aligned} w_{N,3} &= 2w_{N-1,3} + 12w_{N-2,3} + 72w_{N-3,3} + \\ &\quad 64w_{N-4,3} + 64w_{N-5,3} - 256w_{N-6,3}, \end{aligned} \qquad (5.32)$$

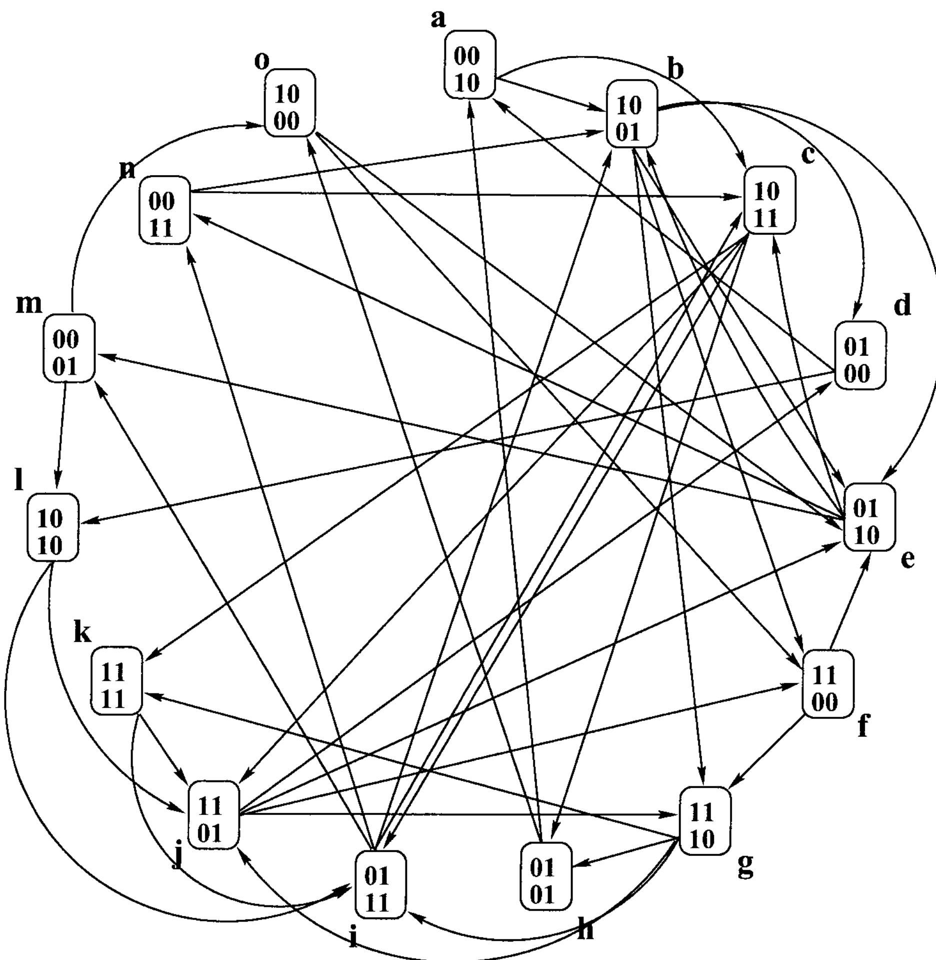


Figure 5.6: The reduced pseudo–codeword state diagram for $m = 3$ and $l = 3$.

for $m = 3$ with initial conditions

$$w_{1,3} = 2,\ w_{2,3} = 12,\ w_{3,3} = 72,\ w_{4,3} = 496,\ w_{5,3} = 2912,\ w_{6,3} = 17856.$$

Thus the recursion is of order 6, while the memory of the code is only 3.

5.5 Conclusions

In this chapter we introduced some techniques to count the number of pseudo–codewords in the trellis of a convolutional code with memory m. We showed that for $m = 2$, there is a simple recursion of order at most m, to count the number of

pseudo–codewords of each degree. We also found that for $l = 2$, there is a simple recursion of order m that counts the pseudo–codewords. In each case we had a simple expression even though we started with a large matrix, which could have resulted in a recursion of much higher degree. Unfortunately, for general l and m there does not seem to be a recursion of order m that counts the pseudo–codewords. However, the order of the recursion seems to be much less than the number of states in the pseudo–codeword state diagram.

For large l and m the techniques introduced in this chapter become intractable very quickly. It appears that the problem of counting pseudo–codewords for general l and m may be hard.

Chapter 6

The Iterative Decoding of Cycle Codes

Cycle codes were first introduced by Gallager [24] as a special case of LDPC codes, in which each codeword bit is checked by exactly two parity checks. However, not much attention was paid to them as they had poor distance properties. They were first studied as graph theoretic codes by Hakimi and Bredeson [53]. While there has been some interest in developing bounded distance decoders for these codes [53, 54], not much attention has been paid to their iterative decoding performance.

In this chapter we will study the behavior of iterative decoding on the Tanner graphs for cycle codes. In Gallager's original analysis of regular LDPC codes, he demonstrated that one can achieve an arbitrarily small error probability for a rate 1/2, (3,6) regular LDPC code, on a BSC with crossover probability 0.04 using his decoding algorithm A[1]. In this chapter we will show that for cycle codes, algorithm A produces a decoded bit error rate of 1/2. We will also present two other message passing algorithms for the BSC which have better performance. The first is the algorithm in example 6 of Richardson and Urbanke [12]. While this algorithm's performance is better than that of algorithm A, it still cannot achieve an arbitrarily small error probability for any $p > 0$. On the other hand, we will demonstrate experimentally that one can achieve quite good performance for the BSC using the iterative sum–product algorithm to decode.

For an AWGN channel we will extend Wiberg's analysis [13, 34] of the iterative min–sum algorithm for cycle codes, to classify the different behaviors of the iterative min–sum decoder and demonstrate that it performs asymptoticly as well as an ML decoder.

[1] Gallager's original analysis applied to the family of LDPC codes whose graphs have infinite girth. Recently Luby et al. [11] and Richardson and Urbanke [12] both extended Gallager's work to include LDPC codes whose underlying graphs have some short cycles.

Here is an outline of the chapter. In Section 6.1, we will introduce cycle codes as graph theoretic codes and describe their connection to the Tanner graph presented in Section 2.2.1. In Section 6.2 we will present three examples of message passing algorithms for the BSC and analyze their behavior. In Section 6.3 we will show that the min–sum algorithm performs asymptotically as well as an ML decoder for cycle codes and introduce the notion of good and bad pseudo–cycles in determining the algorithm's performance. Finally, in Section 6.4, we present some conclusions.

6.1 Basic Notation and Definitions

In this section we shall introduce a graph theoretical framework to describe cycle codes. We shall also present a class of message passing algorithms, that will be used to perform iterative decoding on these graphs. Unlike in previous chapters, we will consider message passing algorithms other than the MSA and SPA.

6.1.1 Graph theoretic codes and Tanner graphs

Let $G = (V, E)$ be a finite undirected graph, where V is the set of vertices and E is the set of edges. Let $|V| = m$ and $|E| = n$. If T is a spanning tree of G, then every edge in $G - T$ will form a unique cycle when it is added to T [55]. We call these cycles the *fundamental cycles* of G. Since there are n edges in G and $m - 1$ edges in T, the number of such cycles is $k = n - m + 1$.

If we label the edges of G, $\{1, \ldots, n\}$, then we can represent every fundamental cycle by a binary n–tuple whose ith component is 1 (or 0) depending on whether the ith edge is (or is not) a part of the cycle. Any cycle or disjoint union of cycles in G can then be expressed as the mod 2 sum of fundamental cycles. The $k \times n$ matrix Φ, whose rows consist of the fundamental cycle vectors, forms a basis for a linear vector space in G. If the edges not in T are labelled $\{1, \ldots, k\}$, then the matrix will be of the form

$$\Phi = [I|\Phi_{12}] \tag{6.1}$$

and is called the *fundamental cycle matrix* of G [55]. The minimum weight of a vector in the span of Φ is simply the cardinality of the shortest cycle in G, i.e., the girth, d of G.

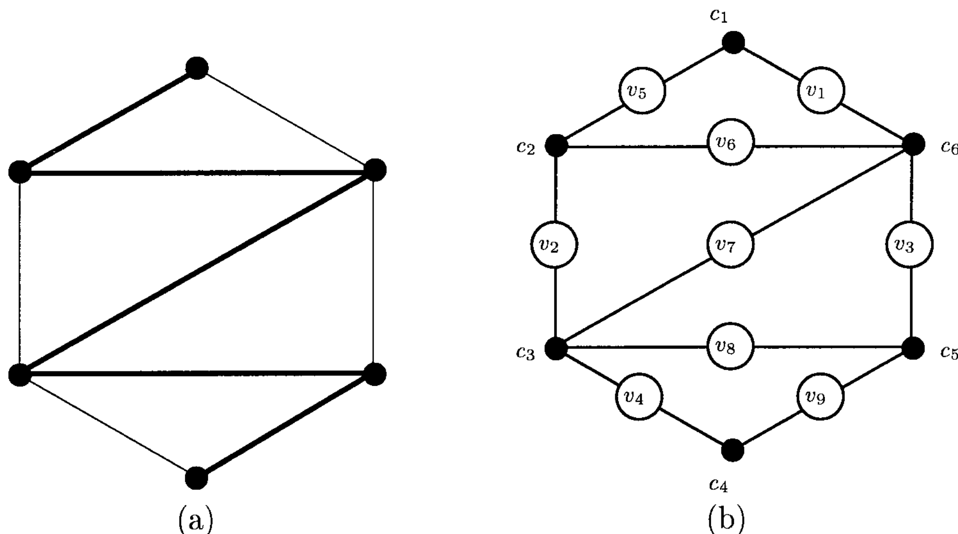


Figure 6.1: (a) The graph G and (b) the Tanner graph G_T, for a (9,4,3) cycle code.

Consider the graph G shown in Figure 6.1a. A possible spanning tree T, of G is shown in bold. For this tree we can label the edges as in Figure 6.1b, to get a fundamental cycle matrix

$$\Phi = \begin{array}{c} \begin{array}{ccccccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & v_9 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array} \tag{6.2}$$

as a generator matrix for the code. The code has dimension $k = 4$, and minimum distance $d = 3$.

Note that a different choice of spanning tree results in a different set of fundamental cycles and thus a different Φ. However, the number of fundamental cycles, k, and the girth d of G, remain the same for every choice of T.

We can represent the cut–set for each vertex by a binary n–tuple whose ith component is 1 (or 0) depending on whether the ith edge is (or is not) connected to that vertex. The $m \times n$ matrix H, whose rows consist of the m cut–set vectors, forms a basis for the null space of Φ. Note that for each connected component of G, we have a redundant cut–set vector in H, so if G is connected, then H will have rank $m-1$. Since every edge is incident on exactly two vertices, every column of H contains two 1's.

The fundamental cycle matrix of G, thus generates an (n,k,d) cycle code $\mathbb{C}$ [53, 54], with H as its parity check matrix. If the graph G is r–regular, then every row of H will have r 1's and $\mathbb{C}$ will be a $(2,r)$–regular LDPC code.

We would like to represent the code $\mathbb{C}$, not by the graph G, but by its Tanner graph G_T as we defined in Section 2.2. We can construct the Tanner graph from H by the method described in Section 2.2, or we can simply set the nodes of G to be the check nodes V_p and place a codeword node V_c on each edge in E. Figure 6.1 shows (a) the graph and (b) the Tanner graph for a (9,4,3) cycle code with parity check matrix

$$H = \begin{array}{c} \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \end{array} \begin{array}{c} \begin{array}{ccccccccc} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & v_9 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix} \end{array}. \tag{6.3}$$

6.1.2 Message passing algorithms

Let $\mathbf{x} = (x_1, \ldots, x_n)$ be a binary vector with x_i, corresponding to codeword node v_i, in the Tanner graph, for $i \in [n]$. The vector $\mathbf{x}$ is a codeword, if and only if $H\mathbf{x} = 0$, where H is the parity check matrix for the graph. Equivalently, we can say that $\mathbf{x}$ is a codeword if and only if the set of codeword nodes attached to each check node has even parity.

The codeword $\mathbf{x}$ is transmitted across the channel and received as a noisy vector $\mathbf{y} = (y_1, \ldots, y_n)$ at the receiver. In this chapter we will consider 2 channel models:

1. The Binary Symmetric Channel (BSC) with crossover probability p.

2. The Additive White Gaussian Noise Channel (AWGN) with signal–to–noise ratio (SNR) E_b/N_o.

Note that both of these channels are memoryless.

There are a number of decoding algorithms for cycle codes based on techniques from combinatorial optimization. In [53], Hakimi and Bredeson propose a decoding algorithm based on majority logic. Jungnickel and Vanstone present a more efficient algorithm in [54], based on the Chinese postman problem [55]. However, both of these algorithms are bounded distance decoding algorithms and can only correct up to t errors where $d \geq 2t + 1$.

In this chapter we will analyze decoding algorithms that use message passing on the Tanner graph for a cycle code. The message passing schedule we will use is described in Section 2.4, while the computation of the messages is more general than for the iterative min–sum or sum–product algorithms. A message from codeword node v_i to check node c_j, along edge (v_i, c_j) is based on the received y_i and the incoming message from the other edge connected to v_i. The message from a check node c_j, to codeword node v_i, is based on all the incoming messages to the check node c_j, except for the message from the edge (v_i, c_j). The message sent out from a node on an edge is therefore not a function of the message coming into that node from that edge. The reason for this is to avoid the "double counting" of information on an edge [29, 30, 32]. Note that the messages in the MSA and the SPA are computed this way.

It is also possible to let the messages depend on the current iteration number, but we will not consider algorithms of this type here but refer the reader to [12] for further details. In the next section we will extend Gallager's, and Richardson and Urbanke's analysis to message passing algorithms for cycle codes on a BSC.

6.2 Iterative Decoding on the BSC

In this section we will analyze the behavior of three different message passing algorithms for a BSC, with crossover probability p. The algorithms we will consider are

- Gallager's decoding algorithm A [24],
- Richardson and Urbanke's example 6 [12] and
- the iterative sum–product algorithm.

In each case we will assume that the all 0's codeword is transmitted across the channel.

In the first two examples, the decoder sends messages over a finite alphabet $\mathcal{A}$, while for the SPA, the message alphabet is the $[0,1]$ interval. Let $p_i^{(l)}$ be the probability that a codeword node sends out message $i \in \mathcal{A}$, at the lth iteration. We will extend the work of Gallager, and Richardson and Urbanke, to find a recursion for $p_i^{(l)}$ for cycle codes.

6.2.1 Gallager's decoding algorithm A

Consider the codeword node v_i, connected to check nodes c_j and c_k by the edges (v_i, c_j) and (v_i, c_k) respectively. The node v_i sends its received value y_i to both check nodes in the first iteration. From the second iteration on, v_i will send the message it receives on edge (v_i, c_j) to c_k and vice–versa. The check node c_j, sends v_i the exclusive–or of all its received messages, except for the one received from the (v_i, c_j) edge.

In Gallager's decoding algorithm B, the codeword node v_i will send its received value y_i along edge (v_i, c_j), after the first iteration, unless a sufficient number of the messages it receives from its other edges are not y_i. Since every codeword node has degree 2, Gallager's decoding algorithms A and B are the same for cycle codes.

Since the algorithm only passes binary valued messages, we only need consider the values of $p_0^{(l)}$ and $p_1^{(l)}$. We would like to find the probability that the node v_i

sends out the incorrect message, $p_1^{(l)}$, at the lth iteration along the edge (v_i, c_k). This is equivalent to finding the probability that the message received on edge (v_i, c_j) is incorrect.

We assume the all zeros codeword is transmitted, so we have $p_0^{(0)} = 1 - p$ and $p_1^{(0)} = p$. We would like to find the probability that we send the incorrect message, $p_1^{(l)}$, at the lth iteration. Gallager [24, p. 48] showed that for an $(r+1)$–regular cycle code, $p_1^{(l)}$ obeys the recursion

$$\begin{aligned} p_1^{(l)} &= p_1^{(0)} - p_1^{(0)} \left[\frac{1 + (1 - 2p_1^{(l-1)})^r}{2} \right] \\ &\quad + (1 - p_1^{(0)}) \left[\frac{1 - (1 - 2p_1^{(l-1)})^r}{2} \right]. \end{aligned}$$

This simplifies to

$$\begin{aligned} p_1^{(l)} &= \frac{1}{2} - \frac{1}{2}(1 - 2p_1^{(l-1)})^r && (6.4) \\ &= \frac{1}{2} - \frac{1}{2}(1 - 2p)^{r^l} && (6.5) \\ &\to \frac{1}{2} \quad \text{as} \quad l \to \infty && (6.6) \end{aligned}$$

for $0 < p < 1/2$. Thus we see that the probability of sending out an incorrect message approaches $1/2$ if we use Gallager's decoding algorithm A, regardless of the channel crossover probability. This makes intuitive sense if we look at what is happening at an edge connected to the root of the computation tree. The message passed to the root on that edge, consists of the parity of all the leaves on the edge's side of the tree. As the tree grows to infinity, the probability of the leaves having even or odd parity is the same as the probability of an infinite Bernoulli(p) sequence having an even or odd number of ones, which is $1/2$.

Thus Gallager's decoding algorithm A is a bad idea for decoding cycle codes on a BSC in fact, it is worse than having no decoding at all.

6.2.2 Richardson and Urbanke's example 6

Richardson and Urbanke [12, Ex. 6] add a third possible message to the decoder to see the effect on its performance. Thus, instead of using a binary message passing alphabet, $\mathcal{A} = \{0, 1\}$, as in Gallager's decoding algorithm A or B, they use the ternary alphabet $\mathcal{A}' = \{-1, 0, 1\}$, where a 0 in $\mathcal{A}$ is mapped to a 1 in $\mathcal{A}'$, and a 1 in $\mathcal{A}$ is mapped to a -1 in $\mathcal{A}'$. The received codeword $\mathbf{y}$ is thus mapped to the vector $\mathbf{y_t} = (y_{t,1}, \ldots, y_{t,n})$ where $\mathbf{y_t} = 1 - 2\mathbf{y}$.

Consider the codeword node v_i, connected to check nodes c_j and c_k by the edges (v_i, c_j) and (v_i, c_k) respectively. The node v_i, sends its received value $y_{t,i}$ to both check nodes in the first iteration. From the second iteration on, v_i sends the sign of the weighted sum of $y_{t,i}$ and the message it receives on edge (v_i, c_j) to c_k and vice–versa. So if m_{ij} is the message v_i receives on edge (v_i, c_j), then the message sent to c_k is equal to $\mathrm{sgn}(w_l y_{t,i} + m_{ij})$, where w_l is a weight that depends on the iteration number l. For cycle codes, one can see that if $w_l > 1$, we always pass, $y_{t,i}$ and if $w_l < 1$, we have Gallager's decoding algorithm A again, so we will assume $w_l = 1$ for all l. The check node c_j, sends v_i, the product of all its received messages, except for the one received from the (v_i, c_j) edge.

If we define $q_i^{(l)}$ to be the fraction of messages i, sent out from the check nodes in round l, then Richardson and Urbanke [12, Ex. 6] showed that for an $(r+1)$–regular cycle code

$$q_1^{(l)} = \frac{1}{2}\left[(p_1^{(l)} + p_{-1}^{(l)})^r + (p_{-1}^{(l)} - p_1^{(l)})^r\right] \tag{6.7}$$

$$q_{-1}^{(l)} = \frac{1}{2}\left[(p_1^{(l)} + p_{-1}^{(l)})^r - (p_{-1}^{(l)} - p_1^{(l)})^r\right] \tag{6.8}$$

$$q_0^{(l)} = 1 - (1 - p_0^{(l)})^r \tag{6.9}$$

and so we have

$$p_1^{(l)} = p_1^{(0)}(q_0^{(l-1)} + q_1^{(l-1)}) \tag{6.10}$$

$$p_{-1}^{(l)} = p_{-1}^{(0)}(q_0^{(l-1)} + q_{-1}^{(l-1)}) \tag{6.11}$$

$$p_0^{(l)} = p_1^{(0)} q_{-1}^{(l-1)} + p_{-1}^{(0)} q_1^{(l-1)}. \tag{6.12}$$

If we want to achieve an arbitrarily small decoding error probability, then we must find a $p_{-1}^{(0)}$ such that $p_{-1}^{(l)} + p_0^{(l)} \to 0$ as $l \to \infty$. Unfortunately, this is not possible, since (6.11) + (6.12) $\leq p_{-1}^{(0)} = p$. In fact, if we look at the algorithm carefully we see that if two variable nodes with a common check node are in error, then the algorithm will always fail to correct them.

Richardson and Urbanke's decoding algorithm in example 6 improves on the performance of Gallager's decoding algorithm A for $j \geq 3$, but it is still a bad idea for decoding cycle codes on a BSC.

6.2.3 The iterative sum–product algorithm

Finally we present some experimental results of the SPA on the BSC to show that we can achieve low bit error rates using message passing to decode cycle codes. Figure 6.2 shows the relative performance of Gallager's algorithm A, Richardson and Urbanke's example 6 and the SPA. We randomly generated graphs with 10000 codeword nodes and 5000 check nodes, resulting in codes of rate approximately 1/2. For Gallager's decoding algorithm A and Richardson and Urbanke's example 6, we performed message passing on both completely random and 4–regular cycle code graphs, while for the SPA we only used 4–regular cycle code graphs. In each case we assumed the all zeros codeword was transmitted and we randomly flipped $10000p$ bits to simulate a channel crossover probability p. For Gallager's decoding algorithm A and Richardson and Urbanke's example 6, our decision for each codeword node was based on the sum of the incoming messages and the received bit, while for the SPA our bit decisions were based on the final state of each node. For each point on the top 4 curves and bottom 2 curves we performed message passing on 1000 and 10000 graphs respectively.

In Figure 6.2, from top to bottom we have the following performance curves. The top two curves, which are indistinguishable, are the performance of Gallager's decoding algorithm A for both random and 4–regular graphs. The probability of error is 1/4 regardless of the channel crossover probability as predicted by equation (6.6),

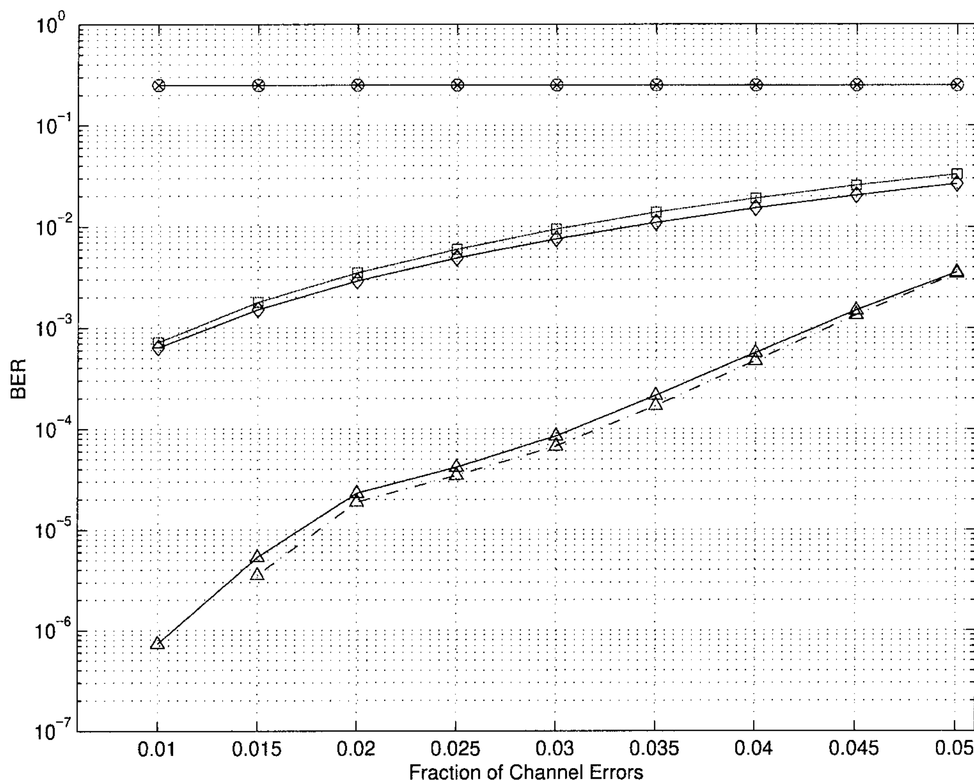


Figure 6.2: The performance of the 3 message passing algorithms on both regular and random graphs for a code rate of 1/2 and a maximum of 150 iterations.

since each codeword node has degree 2. The next two curves show the performance of Richardson and Urbanke's example 6 for random graphs and 4–regular graphs respectively. Message passing performs slightly better on the 4–regular graphs, but still not very well. Finally the bottom 2 curves show the performance of the SPA on 4–regular graphs. The lower, dashed line shows the detected bit error rate while the upper curve is the overall bit error rate. The SPA makes more undetected than detected errors for cycle codes, unlike other LDPC codes [11, 10]. A partial explanation is presented in the next section for the AWGN channel.

We also performed some experiments to see how often the SPA converges to a valid codeword and how many iterations are required. Figure 6.3a shows the median number of iterations before the SPA successfully converges to a codeword. The bars show the 25th and 75th percentiles. Figure 6.3b shows the probability of convergence to a valid codeword. For a crossover probability of 0.045, the SPA finds a codeword

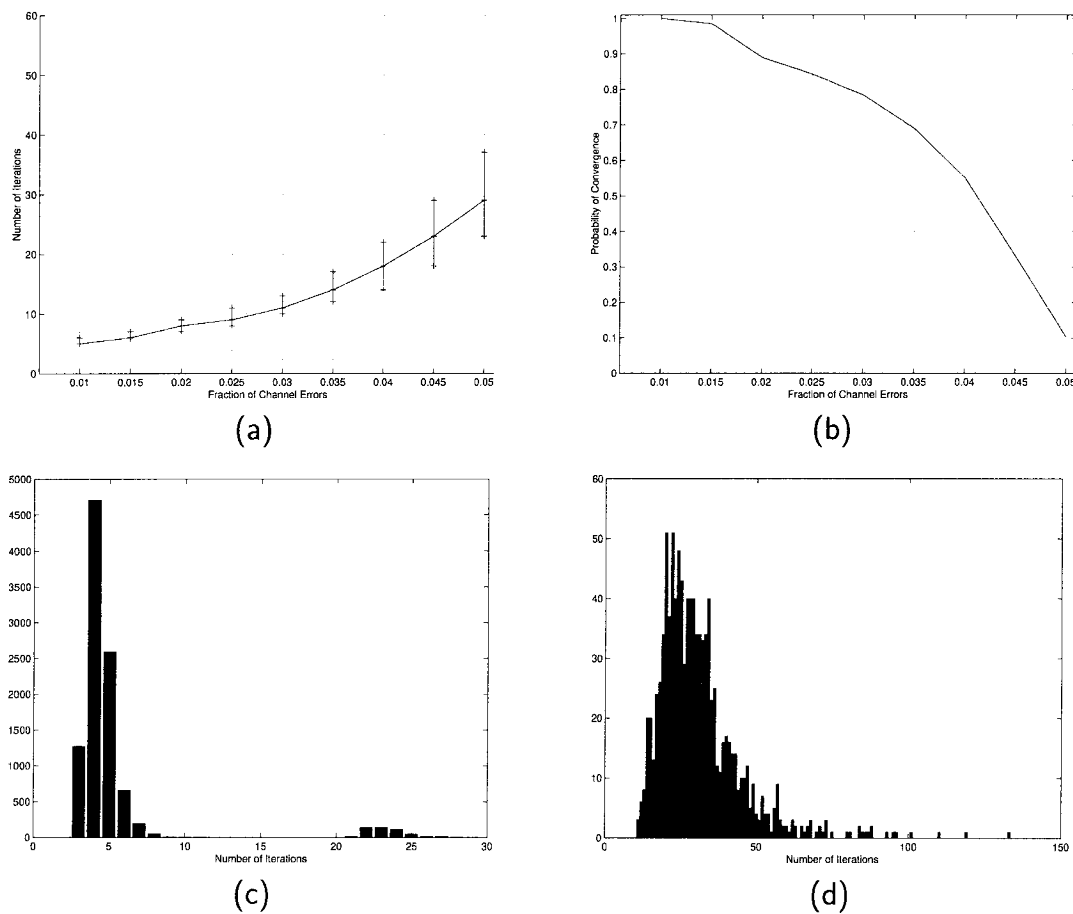


Figure 6.3: Plot (a) shows the median number of iterations for the SPA to successfully converge to a codeword. The bars show the 25th and 75th percentiles. Plot (b) shows the probability of convergence to a valid codeword. Plots (c) and (d) show an empirical distribution of the number of iterations before convergence for a channel crossover probability of 0.01 and 0.045 respectively.

only about 9.8% of the time yet the error probability is 0.00355. Thus only a small fraction of the graph has not converged correctly. Figures 6.3c and 6.3d show an empirical distribution of the number of iterations before convergence for a channel crossover probability of 0.01 and 0.045 respectively.

6.3 Min–Sum Decoding over an AWGN Channel

In this section we will consider cycle codes being used with BPSK modulation on an AWGN channel. In Section 6.3.1, we will define a pseudo–cycle. In Section 6.3.2, we present Algorithm 1 which computes the minimum weight computation tree for which all the parity checks are satisfied, the same minimum weight tree that we get when we trace back the computation of the messages for the MSA. In Section 6.3.3, we use Algorithm 1 to derive sufficient conditions for the MSA to converge to the maximum likelihood codeword and present examples of good and bad pseudo–cycles. In Section 6.3.4, we use linear programming to classify a pseudo–cycle as good or bad. In Section 6.3.5, we define the pseudo–weight of a pseudo–cycle and show that the minimum pseudo–weight of a bad pseudo–cycle is greater than $2d$, i.e., twice the minimum distance of the code. Finally in Section 6.3.6, we present a union bound argument that shows that the iterative min–sum algorithm has the same asymptotic performance as an ML decoder.

6.3.1 Non–cycle irreducible closed walks

In this section we will we will use the definitions and notation from Wiberg [13] to define a non–cycle irreducible closed walk, or pseudo–cycle in the computation tree of the MSA decoder for the Tanner graph.

If we assume that the all zeros codeword is transmitted, then we defined a deviation in Section 2.4, as a locally consistent configuration of the codeword nodes in the computation tree, for which there is a one in the root node. Wiberg [13] showed that such a deviation must have a path between 2 leaves that contains the root node, and for which every codeword node on the path has a bit assignment of one.

We shall call a path, between any two nodes in the computation tree, a *walk* in the underlying Tanner graph, which we write as an alternating sequence of codeword and check nodes $v_{i_1}, c_{i_2}, v_{i_3} \ldots$ such that the same edge is never used twice in a row. Any path in the computation tree corresponds to some walk in the Tanner graph for the code. When the graph has girth 3, there is at most one check node connecting

any two codeword nodes, so we can write the walk as a sequence of only codeword nodes $v_{i_1}, v_{i_2}, v_{i_3} \ldots$, without any ambiguity.

We now have a few more definitions. A *closed walk*, $v_{i_1}, \ldots, v_{i_L}$ is a walk such that $v_{i_1}, v_{i_2}, \ldots, v_{i_L}, v_{i_1}, v_{i_2}$ is also a walk. An *irreducible* walk is one that cannot be written as the concatenation of a closed walk and another walk. A *cycle* is an irreducible closed walk in which every codeword node is distinct and a *pseudo–cycle* is any closed walk that is not a cycle.

We are now ready to examine the MSA in more detail for cycle codes.

6.3.2 Algorithm 1 and the min–sum algorithm

Consider a cycle code with Tanner graph $G_T = (V_T, E_T)$ and computation tree $T_i(l) = (V_l, E_l)$ after l iterations. (Note that the size of the computation tree is a function of the number of iterations l, as well as the message passing schedule.) Let s denote the number of codeword nodes in $T_i(l)$.

If we label the codeword nodes in the computation tree, $v_1, \ldots, v_s$, then for any locally consistent configuration, we define the binary s–tuple D_C, whose ith component denotes the bit assignment to the node v_i in the computation tree, for $i \in [s]$. Define $w(D_C)$ to be the weight of the computation tree given D_C.

Now define

$$D_{opt} = \arg \min_{D_C} w(D_C),$$

where we only consider locally consistent configurations for D_C. We now have the following algorithm and theorem:

```
Algorithm 1:
    D_C = {0,...,0}
    do
       initWeight = w(D_C)
       finWeight  = initWeight
       for (each pair of leaf nodes in T)
          D_temp = D_C w/ all the bits on the path
                          between the 2 leaves inverted
          tempWeight = w(D_temp)
          if (tempWeight < finWeight)
            D_C = D_temp
            finWeight = tempWeight
          end if
       end for
    while (initWeight > finWeight)
```

Theorem 6.3.1 *When Algorithm 1 terminates, D_C contains the minimum weight locally consistent configuration for the computation tree, i.e., $D_C = D_{opt}$.*

(The proof can be found in Appendix A.4.1.)

If we form the computation tree for the MSA on the graph G_T, we will compute the same minimum weight, locally consistent configuration for the computation tree at each iteration [13, 34]. So given a computation tree for l iterations of message passing, the MSA and Algorithm 1 both compute the same set of bit assignments, D_{opt}. In the next section we will use this result to derive a set of sufficient conditions for the MSA to converge to an ML decision.

6.3.3 Convergence of the min–sum algorithm

In this section we will present sufficient conditions for the MSA to converge to an ML decision. We shall say that the MSA has *converged* after l iterations, if for all $l' > l$,

the decision for the MSA based on the state at every codeword node $v \in V_c$, does not change. Thus the MSA has converged if the minimum weight locally consistent configuration has the same bit assignment at the root node of the computation tree, for every codeword node $v \in V_c$, for all $l' > l$.

Without loss of generality, let us assume that the all zeros codeword is the ML codeword. Let the weight of the all zeros path along every cycle be zero. Since the all zeros codeword is the ML codeword, the weight of the all ones path on any cycle must then be strictly positive.

We shall say a pseudo–cycle is *bad*, if there exists a set of weights assigned to each codeword node in G_T, such that the weight of the all ones path on any cycle is strictly positive, while the weight of the pseudo–cycle is negative. Otherwise we shall say that the pseudo–cycle is *good*.

We now have the following theorem.

Theorem 6.3.2 *A sufficient condition for the iterative min–sum algorithm to converge to the all zeros maximum likelihood codeword after sufficiently many iterations, is that the weight of the all ones path on any pseudo–cycle in G_T is positive.*

(The proof can be found in Appendix A.4.2.) Wiberg [13, Theorem 6.1] presented sufficient conditions for the MSA decoder to converge to the transmitted codeword. In the above theorem we consider the relative performance of the MSA decoder and an ML decoder. We are not concerned with whether or not the MSA decoder finds the transmitted bit, as that is more a function of the strength of the code. Here, we are more concerned with determining how well the MSA works relative to an optimal decoder. We now present some examples of good and bad pseudo–cycles.

Given an (n, k, d) cycle code with fundamental cycle matrix Φ, define the $2^k - 1 \times n$ codeword matrix C to be the matrix whose rows consist of all the non-zero codewords in the cycle code.

Example 6.3.1 Consider the Tanner graph for the (5,3,2) parity check code in Figure

6.4, with

$$\Phi = \begin{array}{c} \begin{matrix} v_1 & v_2 & v_3 & v_4 & v_5 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix} \end{array}. \tag{6.13}$$

We have

$$C = \begin{array}{c} \begin{matrix} v_1 & v_2 & v_3 & v_4 & v_5 \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} \end{array}. \tag{6.14}$$

Now let w_i be the cost of assigning a 1 to codeword node v_i, for each $v_i \in V_c$. Thus in order for the all zeros codeword to be the ML decision we must choose a set of weights, $\mathbf{w} = [w_1, w_2, \ldots, w_n]^T$, that satisfies

$$C\mathbf{w} > 0, \tag{6.15}$$

since each row of C corresponds to cycle, or a disjoint union of cycles in G_T.

In order for a pseudo-cycle to be bad, we must be able to pick a set of weights $\mathbf{w}$ that satisfy (6.15) but which favor the all ones assignment for the pseudo-cycle.

Example 6.3.1 (continued) Consider the pseudo-cycle $v_1v_5v_4v_2v_3v_4v_5v_2$, shown in Figure 6.4 by the dashed line. We can represent the pseudo–cycle with a cycle vector $\mathbf{b} = [b_1, b_2, \ldots, b_n]^T$, where b_i's value is the number of time the node v_i appears in the pseudo–cycle. For the pseudo–cycle above we have

$$\mathbf{b} = \begin{bmatrix} 1 & 2 & 1 & 2 & 2 \end{bmatrix}^T. \tag{6.16}$$

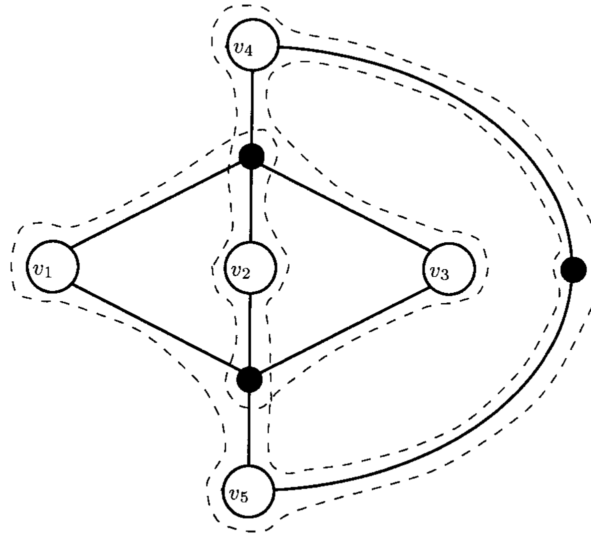


Figure 6.4: The graph of a (5,3,2) cycle code with a good pseudo-cycle illustrated by the dashed line.

For the pseudo–cycle to be bad, we must be able to choose a $\mathbf{w}$ that satisfies the following set of inequalities,

$$C\mathbf{w} > 0 \tag{6.17}$$

$$\text{and } \mathbf{b}^T\mathbf{w} < 0. \tag{6.18}$$

It is easy to see that no such $\mathbf{w}$ exists for this pseudo-cycle, since the sum of the third and sixth row of C is $\mathbf{b}$. Since no solution $\mathbf{w}$ exists, the pseudo-cycle must be good.

In fact for the (5,3,2) parity check code in Figure 6.4, every pseudo-cycle can be shown to be good. This is because every pseudo-cycle in the graph produces a cycle vector $\mathbf{b}$, that can be written as a linear combination of the rows in C with all coefficients positive. In the next section we shall show that this is a necessary and sufficient condition for a pseudo-cycle to be good. But before that we present an example of a bad pseudo-cycle.

Example 6.3.2 (Wiberg [13, Section 6.1]) Consider the (8,3,3) cycle code shown in

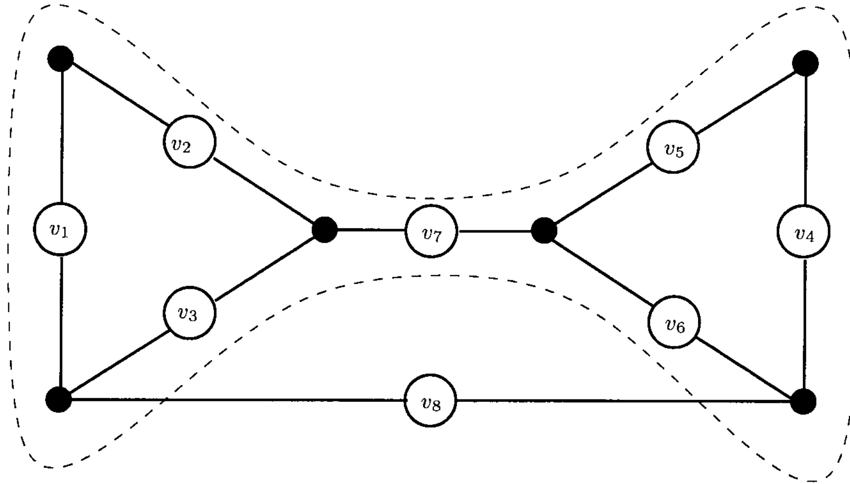


Figure 6.5: The Tanner graph for an (8,3,3) cycle code with a bad pseudo–cycle illustrated by the dashed line.

Figure 6.5, with

$$C = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{pmatrix}. \tag{6.19}$$

(columns labelled $v_1\ v_2\ v_3\ v_4\ v_5\ v_6\ v_7\ v_8$)

The pseudo–cycle, $v_1v_2v_7v_5v_4v_6v_7v_3$, is shown with a dashed line. The corresponding cycle vector is

$$\mathbf{b} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 2 & 0 \end{bmatrix}^T. \tag{6.20}$$

Let $\mathbf{w} = [1, 1, 1, 1, 1, 1, -4, 3]^T$, so we have

$$\begin{aligned} C\mathbf{w} &= [3\ 3\ 1\ 6\ 2\ 2\ 3] \\ \text{and } \mathbf{b}^T\mathbf{w} &= -2. \end{aligned}$$

Thus the pseudo-cycle is bad.

In the next section we use linear programming to devise a simple procedure for determining whether or not a given pseudo–cycle is good or bad.

6.3.4 Classifying pseudo-cycles using linear programming

In this section we shall formulate the pseudo-cycle problem as a linear program and classify all pseudo-cycles using the Farkas alternative.

For a given codeword matrix C and pseudo-cycle $\mathbf{b}$, we would like to solve the dual linear problem [56]:

$$C\mathbf{w} \geq 0 \tag{6.21}$$

$$\mathbf{b}^T\mathbf{w} = \text{minimum.} \tag{6.22}$$

A vector $\mathbf{w}$ that satisfies (6.21) is called a feasible solution. If it satisfies both (6.21) and (6.22), then it is called an optimal solution.

From linear programming we know that the optimal solution $\mathbf{w}$ has $\mathbf{b}^T\mathbf{w} = 0$ if and only if the primal problem

$$C^T\mathbf{x} = \mathbf{b} \tag{6.23}$$

$$\mathbf{x} \geq 0 \tag{6.24}$$

has a feasible solution $\mathbf{x}$, i.e., $\mathbf{b}$ can be written as the weighted sum of the cycles in G_T with all coefficients positive. If $\mathbf{b}^T\mathbf{w} = 0$, we have a good pseudo–cycle, otherwise we can select vectors $\mathbf{w}$ to drive the solution $\mathbf{b}^T\mathbf{w}$ to $-\infty$ and we have a bad pseudo–cycle. In order to examine the possible solutions to the linear program, we first need the following two lemmas.

Lemma 6.3.3 Wiberg [13, Lemma 6.1] *No codeword node occurs more than twice in a pseudo–cycle.*

(The proof can be found in Appendix A.4.3.)

The elements of $\mathbf{b}$ are therefore ternary valued $\{0, 1, 2\}$ and we have:

Lemma 6.3.4 *Let $S = v_{i_1}, v_{i_2}, \ldots, v_{i_L}$ be a pseudo-cycle of length L in G_T with cycle vector $\mathbf{b} = [b_1, b_2, \ldots, b_n]^T$. If the rows of the codeword matrix C are labelled $c^1, c^2, \ldots, c^{2^k-1}$, then we can say the following about b_i, for $i \in [n]$:*

1. *If $b_i = 1$, then there is a c^j such that $\mathbf{b} - c^j \geq 0$.*
2. *If $b_i = 2$, then either*
   - **i** *there is no c^j such that $\mathbf{b} - c^j \geq 0$,*
   - **ii** *there is a c^j such that $\mathbf{b} - 2c^j \geq 0$, or*
   - **iii** *there is a c^j and c^l, $l \neq j$ such that $\mathbf{b} - c^j \geq 0$ and $\mathbf{b} - c^l \geq 0$.*

(The proof can be found in Appendix A.4.4.) The second lemma states that if we consider the subgraph G', induced by the nodes in S, if a codeword node occurs only once in S, then it is contained in a cycle in G'. If a codeword node occurs twice in S, then it can either be in no cycles in G', in two distinct cycles, or a single cycle in which each codeword node appears twice in S.

We now present the main result of this section.

Theorem 6.3.5 *A pseudo-cycle S is bad if and only if the subgraph G' induced by the nodes in S contains a node that is not in any cycle.*

(The proof can be found in Appendix A.4.5.)

The theorem implies that any bad pseudo-cycle consists of two or more disjoint closed walks joined by a path. These disjoint closed walks may be pseudo-cycles themselves or they could just be cycles as in the pseudo-cycle in Figure 6.5. For the (5,3,2) cycle code in Figure 6.4, every pseudo-cycle induces a subgraph for which every node is contained in a cycle. Thus there are no bad pseudo-cycles for the (5,3,2) cycle code. In the next section we will calculate the minimum pseudo-weight of bad pseudo-cycles and use this to bound the performance of the MSA decoder.

6.3.5 The minimum pseudo–weight of bad pseudo–cycles

If $S = v_{i_1}, \ldots, v_{i_L}$ is a pseudo–cycle of length L with cycle vector $\mathbf{b} = [b_1, b_2, \ldots, b_n]^T$, then for an AWGN channel with BPSK modulation, Wiberg [13] defined the pseudo–weight of the pseudo–cycle to be

$$\mathcal{W}(S) = \frac{(\sum_{i=1}^{n} b_i)^2}{\sum_{i=1}^{n} b_i^2}. \tag{6.25}$$

Note that the pseudo–weight of an ordinary cycle, or a disjoint union of cycles, is the same as the Hamming weight of the corresponding codeword.

We would like to compute a lower bound on the minimum pseudo–weight of a bad pseudo–cycle. In order to do this we first need the following two lemmas.

Lemma 6.3.6 *Any pseudo–cycle contains at least two cycles which may be partially overlapping, or disjoint.*

(The proof can be found in Appendix A.4.6.)

Lemma 6.3.7 *If $G' = (V', E')$ is the subgraph induced by the pseudo–cycle S in G_T, and $|V'|$ is the number of codeword nodes in G', then $|V'| \geq \frac{3}{2}d$, where d is the minimum distance of the code.*

(The proof can be found in Appendix A.4.7.)

From Lemma 6.3.6 and Lemma 6.3.7, we can derive a lower bound on the minimum pseudo–weight of a good or bad pseudo–cycle.

Lemma 6.3.8 *Consider an (n, k, d) cycle code with Tanner graph $G_T = (V_T, E_T)$. Every pseudo–cycle S, in G_T has $\mathcal{W}(S) \geq \frac{4}{3}d$.*

(The proof can be found in Appendix A.4.8.)

Since a bad pseudo–cycle consists of two disjoint pseudo-cycles or cycles, joined by a path, we can use Theorem 6.3.5 and Lemma 6.3.8 to establish an even stronger lower bound on the pseudo–weight of bad pseudo–cycles.

Theorem 6.3.9 *Consider an (n, k, d) cycle code with Tanner graph $G_T = (V_T, E_T)$. Every bad pseudo–cycle S, in G_T has $\mathcal{W}(S) \geq 2d$.*

(The proof can be found in Appendix A.4.9.)

Since there are a finite number of bad pseudo–cycles, we can use Theorem 6.3.2 and Theorem 6.3.5 to estimate the decoding performance of the MSA decoder using the same union bound argument we used for tail–biting codes in Section 4.2.7.

6.3.6 Estimating the error probability with the union bound

Assume the all zeros codeword is transmitted across an AWGN channel with BPSK modulation and received as the vector $\mathbf{y}$. From Theorem 6.3.2, we know that the MSA decoder will converge to the ML codeword for $\mathbf{y}$, if the weight assignment to the codeword nodes in G^T corresponding to $\mathbf{y}$ does not result in any bad pseudo–cycles in G^T. Wiberg [13, Theorem 6.1] has shown that a necessary condition for the MSA decoder to make an error is for there to be a bad pseudo–cycle or cycle due to $\mathbf{y}$.

Let $\mathcal{P}$ denote the finite set of bad pseudo–cycles in G_T and $\mathcal{C}$ the set of cycles and disjoint union of cycles. Define the error event

$$\varepsilon = \{S \text{ is a bad pseudo–cycle or cycle}\}$$

for every $S \in \mathcal{P} \cup \mathcal{C}$. We have

$$Pr(\varepsilon) = Q\left(\sqrt{2R\mathcal{W}(S)E_b/N_o}\right), \tag{6.26}$$

where $Q(t) = (1/\sqrt{2\pi}) \int_t^{\infty} e^{-s^2/2} ds$ and R is the rate of the code.

If we define P_E^{MSA} to be the decoder word error probability, then by the standard union bound, we have

$$P_E^{MSA} \leq \sum_{S \in \mathcal{P} \cup \mathcal{C}} Q\left(\sqrt{2R\mathcal{W}(S)E_b/N_o}\right). \tag{6.27}$$

For an ML decoder, we can ignore all bad pseudo–cycles in computing the ML word error probability P_E^{ML}, so we have the standard union bound

$$P_E^{ML} \leq \sum_{S \in C} Q\left(\sqrt{2RW(S)E_b/N_o}\right). \tag{6.28}$$

Since the minimum pseudo–weight of a pseudo–cycle is greater than the minimum distance of the code, these two union bounds are asymptotically equal. For instance in Wiberg [13, Fig. 6.4], the (15,6,5) cycle code for the Peterson graph has almost identical block error rate performance for a MSA decoder as for an ML decoder for all E_b/N_o.

6.4 Conclusions

In this chapter, we have shown that message passing algorithms can approach the performance of an ML decoder in the case of an AWGN channel for a cycle code. We have also shown that message passing algorithms, which can achieve an arbitrarily small error probability above a certain threshold for LDPC codes with $j > 2$, can also have terrible performance on a BSC for a cycle code.

For the iterative min–sum decoder on the AWGN channel we have classified the pseudo–cycles in the Tanner graph of the code and shown that the minimum weight of a bad pseudo–cycle is greater than twice the minimum distance of the code. Thus cycle codes are the first class of codes for which iterative decoding can be shown to be asymptotically equivalent to ML decoding. For tail–biting codes we have already shown in Section 4.2.8 that a pseudo–codeword can have a pseudo–weight that is less than the minimum distance of the code.

The Tanner graph for a turbo code also has all codeword nodes and state nodes of degree 2. However, pseudo–cycles no longer govern the behavior of the iterative min–sum algorithm. The existence and pseudo-weight of a structure equivalent to pseudo–cycles, for turbo codes or for the general class of LDPC codes is a subject of future research.

Chapter 7

Conclusions

Iterative decoding of codes defined on graphs with cycles, appears to be an efficient means of achieving the performance Shannon predicted possible, some fifty years ago. By increasing our understanding of the behavior of the min–sum and sum–product algorithms on graphs with cycles, we can design our codes to perform better under iterative decoding and we can choose graphs which minimize the presence of low-weight pseudo–codewords.

Much of this thesis has been devoted to the analysis of the performance of the MSA and SPA on the graphs for tail–biting codes and cycle codes. We have given sufficient conditions for the MSA to converge to the maximum likelihood codeword after a finite number of iterations. We have also used the familiar union bound argument to characterize the performance of the MSA after many iterations. For a cycle code, we have shown that the performance of the MSA decoder is asymptotically as good as maximum likelihood. For tail–biting codes this depends on the trellis representation we choose. For instance, a tail-biting trellis with a minimal state complexity of two, can never have a pseudo–codeword whose pseudo–weight is less than the minimum distance of the code. For the (2,1,2) convolutional code with generators (7,5), there are also no pseudo–codewords with weight less than the minimum distance. However, this is not the case in general and specifically for the time invariant trellis with generators (414, 730), for the extended Golay code.

We would of course like to extend our analysis to the behavior of the MSA and SPA to the graphs for turbo codes and LDPC codes. For a generic turbo code with two component codes, all the information nodes and state nodes in the Tanner graph have degree 2 and all the check nodes have degree 3. Thus, the graph has a structure similar to a cycle code, except that some nodes are non–binary. Similarly, a LDPC

code is also similar to the graph for a cycle code, except the codeword nodes have a higher degree. Unfortunately, unlike cycle codes or tail–biting codes, Wiberg [13] has shown that the pseudo–codewords for these codes, form a subtree in the computation tree, so a significant portion of the pseudo–weight will lie in the leaves. Thus, we can no longer ignore the boundary conditions when analyzing the pseudo–codewords. This does not imply that the pseudo–codewords do not have a simple repetitive structure, equivalent to a pseudo–cycle, in the underlying Tanner graph, but we have not found one yet, if it exists.

We conclude with a simple experiment, which shows how the performance of the SPA can vary, for differing choices of graph. Consider the (15,5,6) code with parity check matrix

$$H = \begin{pmatrix} 1&0&0&0&1&1&0&0&0&0&0&0&0&0&0 \\ 1&1&0&0&0&0&1&0&0&0&0&0&0&0&0 \\ 0&1&1&0&0&0&0&1&0&0&0&0&0&0&0 \\ 0&0&1&1&0&0&0&0&1&0&0&0&0&0&0 \\ 0&0&0&1&1&0&0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0&0&1&0&1&0&0 \\ 0&0&0&0&0&0&1&0&0&0&0&1&0&1&0 \\ 0&0&0&0&0&0&0&1&0&0&0&0&1&0&1 \\ 0&0&0&0&0&0&0&0&1&0&1&0&0&1&0 \\ 0&0&0&0&0&0&0&0&0&1&0&1&0&0&1 \\ 0&1&0&0&0&0&0&0&0&1&1&0&0&0&0 \\ 0&0&1&0&0&1&0&0&0&0&0&1&0&0&0 \\ 0&0&0&1&0&0&1&0&0&0&0&0&1&0&0 \\ 0&0&0&0&1&0&0&1&0&0&0&0&0&1&0 \\ 1&0&0&0&0&0&0&0&1&0&0&0&0&0&1 \end{pmatrix}. \tag{7.1}$$

The matrix H defines a (3,3) LDPC code. The first 5 or middle 5 rows of H are redundant and can be removed without changing the rank of H. If we remove the last 5 rows, we get a parity check matrix for the (15,6,5) Peterson graph [37], which

is a cycle code that contains the (15,5,6) code as a sub-code.

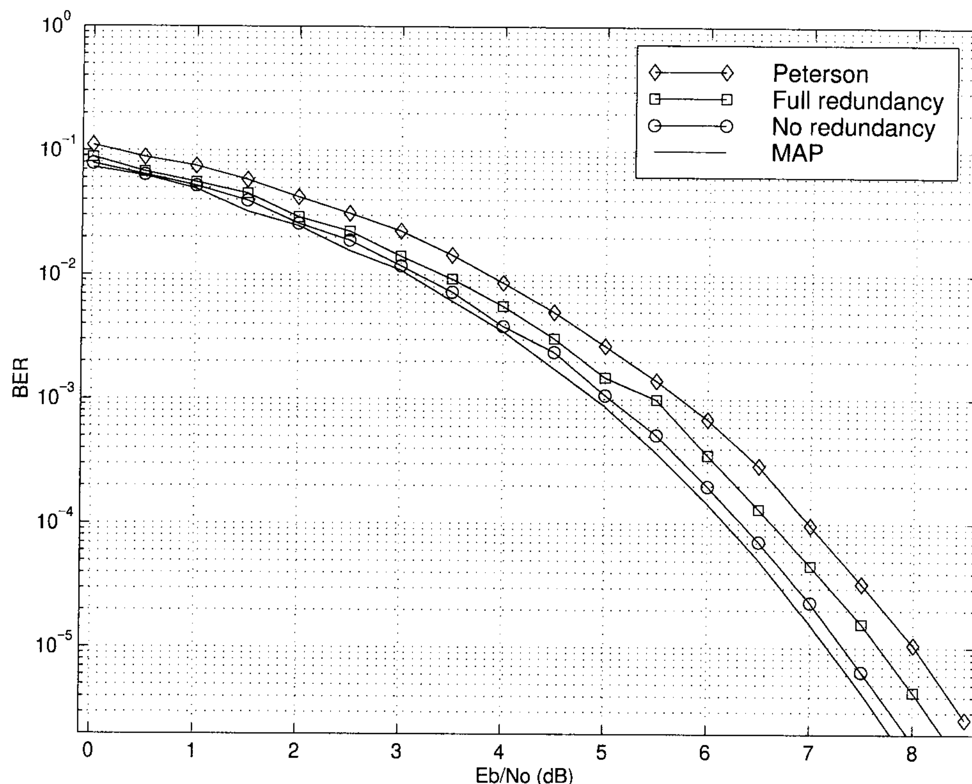


Figure 7.1: The performance of the SPA decoder for the (15,5,6) code on an AWGN channel with a maximum of 25 iterations.

Figure 7.1 shows the performance of the SPA decoder for the (15,5,6) code on an AWGN channel with a maximum of 25 iterations. After 5 or more iterations, we halted if the decision for each node resulted in a codeword. The top curve shows the performance of the SPA decoder on the Peterson graph. The second curve shows the performance for the full matrix H, while the third curve is the performance with the middle 5 rows of H removed. The bottom curve is the performance of the MAP decoder. From Chapter 6, we know that for the Peterson graph, the bad pseudo–cycles will either be of weight 5, or 10 or more, due to the additional codewords allowed in the graph and the bad pseudo–cycles respectively. The minimum pseudo–weight for the full matrix appears to be between 5 and 6.

For sufficiently large LDPC codes, the decoding errors, found to date, are always due to low weight pseudo–codewords, since the SPA decoder has never been observed

to converge to an incorrect codeword [10, 11]. This leads us to conjecture that for the general class of LDPC codes, the pseudo–weight is less than the minimum distance. The actual structure and pseudo–weight of these pseudo–codewords are subjects for future research.

Appendix A

Proofs and Derivations

A.1 Proof of Theorem 3.6.1

Consider the matrix M with $m_{11} > m_{ii}$, for $i = 2, \ldots, n$. We can write $Mu = \lambda u$ as

$$\begin{bmatrix} am_{11}u_1 + bu_2 + \cdots + bu_n \\ bu_1 + am_{22}u_2 + \cdots + bu_n \\ \vdots \\ bu_1 + bu_2 + \ldots + am_{nn}u_n \end{bmatrix} = \lambda \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix}.$$

Thus

$$am_{11}u_1 + bu_2 + \cdots + bu_n = \lambda u_1 \tag{A.1}$$

$$\text{and} \quad am_{ii}u_i + bu_1 + \ldots + bu_n - bu_i = \lambda u_i, \tag{A.2}$$

for $i = 2, \ldots, n$.

Subtracting (A.2) from (A.1) we get

$$m_{11}u_1 - m_{ii}u_i = \frac{\lambda + b}{a}(u_1 - u_i), \tag{A.3}$$

for $i = 2, \ldots, n$.

Define $d = \frac{\lambda+b}{a}$. Since M is Hermitian and λ is its largest eigenvalue, $\lambda \geq am_{11}$ and so $d \geq m_{11} + \frac{b}{a}$. Thus from (A.3) $u_1 > u_i$ for $i = 2, \ldots, n$. $\diamond$

A.2 Proofs for Section 4.2

A.2.1 Proof of Theorem 4.2.1

(We reproduce the proof of Stanley [46, Theorem 4.7.1] for completeness.)

The result is immediate by definition of matrix multiplication, i.e.,

$$[A^n]_{ij} = \sum A_{ii_1} A_{i_1 i_2} \dots A_{i_{n-1} j}$$

where the sum is over all sequences $i_1 i_2 \dots i_{n-1} \in m^{n-1}$. If $i i_1 i_2 \dots i_{n-1} j$ is a path, then its weight is equal to the product of the weights on the edges. Otherwise its weight is zero and the proof follows. $\diamond$

A.2.2 Proof of Theorem 4.2.2

Consider the matrix $A^* = A^l$. By Theorem 4.2.1, l of the entries on the diagonal of A^* will be equal to λ^l, corresponding to the l vertices in C. The rest of the entries on the diagonal will be greater than λ^l since λ is by assumption, the minimum average edge weight for a cycle in D.

Let D^* be the digraph associated with the matrix A^*. We begin by subtracting λ^l from the weight function for each edge in D^*, i.e., subtracting λ^l from each entry of A^*. We now have $\lambda^l = 0$. By assumption D^* has no negative weight cycles.

Define $p^*_{i,j}$ be the minimum weight path from vertex i to vertex j in D^*, and let $|p^*_{i,j}|$ be its length. Now by Theorem 4.2.1, the weight of the path $p_{i,j}$, of length n in D^* is the (i, j)–entry of $[A^*]^n$.

There are two possibilities for $p_{i,j}$:

Case i The path contains some vertex $c \in C$:

We have

$$w(p_{i,j}) \geq w(p^*_{i,c}) + w(p^*_{c,j}). \tag{A.4}$$

The lower bound is achievable for $n > |p^*_{i,c}| + |p^*_{c,j}|$, since $w(p^*_{c,c}) = 0$. It is finite since D is strongly connected.

Case ii The path does not contain a vertex in C:

We can factor the path into a collection of cycles, $P_1, \ldots P_s$ and a simple path P_0. We have

$$w(p_{i,j}) = w(P_0) + \sum_{k=1}^{s} w(P_k) \tag{A.5}$$

$$\geq w(P_\alpha) + s \cdot w(P_\beta), \tag{A.6}$$

where P_α is the lowest weight simple path in D^* and P_β is the second lowest weight cycle in D^*, so $w(P_\beta) > 0$.

For large enough s, we have

$$w(P_\alpha) + s \cdot w(P_\beta) \geq w(p^*_{i,c}) + w(p^*_{c,j}). \tag{A.7}$$

Thus for sufficiently large n, we are guaranteed a minimum of s cycles in the path, so $p_{i,j}$ must contain a vertex in C.

The matrix P will have entries

$$[\mathsf{P}]_{i,j} = w(p^*_{i,c}) + w(p^*_{c,j}) \tag{A.8}$$

for i, $j \in [m]$, where

$$c = \arg\min_{v \in C}(w(p^*_{i,v}) + w(p^*_{v,j})).$$

For $l = 1$, we have $A^* = A$, so the critical cycle is a self loop at vertex c. With min–sum arithmetic

$$\mathsf{P} = \mathbf{x}\mathbf{y}^T, \tag{A.9}$$

where

$$\mathbf{x} = [w(p^*_{1,c}) \; w(p^*_{2,c}) \cdots w(p^*_{m,c})]^T$$

is the right eigenvector of A and

$$\mathbf{y} = [w(p^*_{c,1}) \; w(p^*_{c,2}) \cdots w(p^*_{c,m})]^T$$

is the left eigenvector of A.

We complete the proof by showing that the eigenvalue of A^* is unique. Let v be a right eigenvector of A^* with eigenvalue μ, so

$$A^* v = \mu v.$$

Assume $\mu > \lambda$. We have shown above that

$$\left[\frac{1}{\lambda} A^*\right]^n = \mathsf{P},$$

so for sufficiently large n, we have

$$\mathsf{P} v = \left(\frac{\mu}{\lambda}\right)^n v. \tag{A.10}$$

Since all the entries in P are finite and $\mu > \lambda$, by (A.10), $v = [\infty \ \cdots \ \infty]^T$ since the right hand side of (A.10) is increasing in n. Thus any non-trivial eigenvector of A^* must have an eigenvalue equal to λ. $\diamond$

A.2.3 Proof of Theorem 4.2.4

We associate an incidence matrix A_i, with section i of the trellis, for $0 \le i \le N-1$. The entries of A_i are the negative log-likelihoods of the ith section of $\mathbf{y}$, given the output associated with each edge in the ith section of the trellis.

Now consider the matrix product $\mathbf{A} = (A_0 A_1 \ldots A_{N-1})$ for the trellis. Since we are in the min–sum semiring, the (i,j)–entry of $\mathbf{A}$ is given by the minimum weight path of length N, from state i to state j in the trellis. By the min–sum Perron–Frobenius theorem, the MSA decoder converges, after a finite number of iterations, to the path of minimum average weight in the trellis, i.e., the dominant pseudo–codeword. $\diamond$

A.2.4 Proof of Lemmas 4.2.5 and 4.2.6

We first prove Lemma 4.2.5. Consider a pseudo–codeword $\mathbf{x} = [\mathbf{x}^1 \mathbf{x}^2 \ldots \mathbf{x}^l]^T$, of degree l. The mod 2 sum $\mathbf{x}^1 + \mathbf{x}^2 + \cdots + \mathbf{x}^l$ defines a codeword, since the trellis is linear. For $l = 2$, this implies that $\mathbf{x}^1 \neq \mathbf{x}^2$, as otherwise we would have a non–zero state sequence producing the all–zero codeword.

Define $\hat{d}$ to be $|\{c_j : c_j = 1, j \in [n]\}|$, so for $l = 2$, $\hat{d}$ is at least the minimum distance of the code. Let $m = |\{c_j : c_j = 2, j \in [n]\}|$ count the remaining non–zero columns of $\mathbf{x}$. The pseudo–weight is then

$$\begin{aligned}\mathcal{W}(\mathbf{x}) &= \frac{(\sum_j c_j)^2}{\sum_j c_j^2} \\ &= \frac{(\hat{d} + 2m)^2}{\hat{d} + 4m} \\ &\geq \hat{d}. \diamond\end{aligned}$$

In order to prove Lemma 4.2.6, we first need to show that the average weight of a pseudo–codeword is less than its pseudo–weight. Define the average weight of a pseudo–codeword to be

$$\overline{\mathcal{W}} = \frac{1}{l}\sum_{j=1}^{N} c_j. \tag{A.11}$$

Now

$$\begin{aligned}\mathcal{W} &= \frac{(\sum_j c_j)^2}{\sum_j c_j^2} \\ &= \frac{\sum_j c_j}{l} \frac{l(\sum_j c_j)}{\sum_j c_j^2} \\ &\geq \overline{\mathcal{W}},\end{aligned}$$

since $c_j \leq l$, for $l \in [N]$.

By definition a pseudo–codeword can only visit a state or an edge once in the trellis. For an N–section trellis, if $l = \alpha + i$, then the average weight of the pseudo–codeword must be at least Ni/l. Since the minimum distance for the code is constant,

for large enough N the average weight will be greater than the minimum distance. ⋄

A.3 Proofs for Chapter 5

A.3.1 Proof of Theorem 5.1.1

(We reproduce the proof of McEliece [48, Theorem 1.2] for completeness.)

By Theorem 4.2.1, we have $A_{i,j}(n) = [A^n]_{i,j}$ so we get

$$\begin{aligned} F_{i,j}(z) &= \sum_{n\geq 0} A_{i,j}(n) z^n \\ &= \sum_{n\geq 0} [A^n]_{i,j} z^n \\ &= [(I - zA)^{-1}]_{i,j}, \end{aligned}$$

as this is the sum of a geometric series for matrices. ⋄

A.3.2 Proof of Theorem 5.1.2

(We reproduce the proof of Stanley [46, Corollary 4.7.3] for completeness.)

Let $\lambda_1, \ldots, \lambda_m$ be the eigenvalues of A. By Theorem 4.2.1, we have $w_n = \mathrm{tr}(A^n)$ so we get

$$\begin{aligned} W(z) &= \sum_{n\geq 1} \mathrm{tr}(A^n) z^n \\ &= \sum_{n\geq 1} \sum_{i=1}^{m} \lambda_i^n z^n \\ &= \sum_{i=1}^{m} \frac{\lambda_i z}{1 - \lambda_i z} \\ &= -\frac{zQ'(z)}{Q(z)}, \end{aligned}$$

since $\prod_{i=1}^{m}(1 - \lambda_i z) = Q(z)$ and $\mathrm{tr}(A^n) = \lambda_1^n + \cdots + \lambda_m^n$. ⋄

A.3.3 Proof of Theorem 5.3.1

We can take an arbitrary bit string of length N and append it to itself $l-1$ times to form a sequence of length lN, denoted $\mathbf{y}'$. There are 2^N such strings possible. For each $\mathbf{t}$ and $\mathbf{y}'$, the string $\mathbf{t} \oplus \mathbf{y}'$ is unique.

For a given $\mathbf{t}$, we can define the strings $\mathbf{y}'_\mathbf{0}, \ldots, \mathbf{y}'_\mathbf{l-1}$ as follows

$$(\mathbf{y}'_\mathbf{i})_j = t_{j \, mod \, N + iN}$$

for $j = 0, \ldots, lN-1$ and $i = 0, \ldots, l-1$. The set of strings $\mathbf{t} \oplus \mathbf{y}'_\mathbf{0}, \ldots, \mathbf{t} \oplus \mathbf{y}'_\mathbf{l-1}$ all form pseudo-codewords in $\mathbf{T}$ which are shifted by some multiple of N bits from each other. By definition these are all the same pseudo-codeword. Similarly every $\mathbf{s} \in \mathbf{S}$ will occur l times if we add all 2^N strings $\mathbf{y}'$ to every $\mathbf{t} \in \mathbf{T}$. Thus $w_{N,l} = v_{N,l} 2^N / l$ and the proof follows. $\diamond$

A.3.4 Proof of Theorem 5.3.2

A pseudo-codeword $\mathbf{s}$ corresponds to a unique path in the original pseudo-codeword state diagram. The last m transitions in the path are determined by the initial state of $\mathbf{s}$ and vice-versa. The path maps onto a unique path in the new pseudo-codeword state diagram. If we reverse the mapping, then the last m transitions will determine which of the 2^l initial states we start from in the original pseudo-codeword state diagram and the transitions are uniquely defined afterwards. Therefore the mapping between pseudo-codeword paths in the original pseudo-codeword state diagram and the new pseudo-codeword state diagram is one-to-one. $\diamond$

A.4 Proofs for Section 6.3

A.4.1 Proof of Theorem 6.3.1

Our proof consists of two parts. First we will show that D_C is always a locally consistent configuration for the computation tree. We then show that the algorithm

halts with $D_C = D_{opt}$.

Consider a set of binary variables that satisfy a parity check. If we invert an even number of those variables, then the parity check will still be satisfied. Now, given a locally consistent configuration D_C and a pair of leaves in V_l, since $T_i(l)$ is a tree, the path between any pair of leaves is unique. If we invert all the bits in D_C for the codeword nodes on the path between the 2 leaves, then we are inverting a pair of bits for each check node on the path. The check nodes not on the path are unaffected. Since we start with a globally consistent tree with $D_C = \{0, \ldots, 0\}$, and we invert all the variables on the path between 2 leaves each time we change D_C, we will have a locally consistent configuration for each iteration of the algorithm.

Now it is clear that a minimum weight computation tree must exist. Given s codeword nodes in $T_i(l)$, there are at most 2^s possible ways to assign a value to all of them. Of course only a fraction of these will result in a locally consistent configurations. The configuration D_{opt}, that minimizes the weight of the computation tree while satisfying all the parity checks will be among these. Let us further assume that D_{opt} is unique, i.e., if $D_C \neq D_{opt}$, and D_C corresponds to a locally consistent configuration, then $w(D_C) > w(D_{opt})$.

For each iteration of the do loop, $w(D_C)$ decreases, thus Algorithm 1 must eventually halt. Suppose the algorithm halts with $D_C \neq D_{opt}$. Then in order for both D_C and D_{opt} to be locally consistent, there must be a path between 2 leaves of $T_i(l)$ for which all the variables in D_C are the inverse of those in D_{opt}. By inverting those bits for D_C, we can obtain a lower cost tree since D_{opt} is the configuration for the minimum weight tree. Algorithm 1 would not halted if such a path exists, therefore when the algorithm halts $D_C = D_{opt}$. $\diamond$

A.4.2 Proof of Theorem 6.3.2

We initialize $D_C = \{0, \ldots, 0\}$ and run Algorithm 1 to find the minimum weight, locally consistent, computation tree. The bit assignment to the root node can change from 0 to 1, only if there is a path, including the root node, between two leaves in

the computation tree for which the all ones bit assignment has lower weight than the all zeros assignment.

Let $\mathcal{C}$ be the set of irreducible walks and let $\mathcal{P}$ be the set of pseudo-cycles and cycles in G_T. Now let $w(S)$ to be the weight of $S \in \mathcal{C} \cup \mathcal{P}$ with the all ones assignment and define

$$\delta = \min_{S \in \mathcal{P}} w(S),$$

and

$$\rho = \min_{S \in \mathcal{C}} w(S).$$

By assumption we have $\delta > 0$.

Every path between 2 leaves in the computation tree will consist of a series of closed walks and an irreducible walk. We label the walks $S_0, S_1, \ldots, S_l$ where S_0 is an irreducible walk and all the other walks $S_1, \ldots, S_l$ are either pseudo-cycles or cycles. For sufficiently many iterations, every path between two leaves, that includes the root node, must include at least l closed walks. The cost of the path is

$$\begin{aligned} w &= w(S_0) + \sum_{i=1}^{l} w(S_i) \\ &\geq \rho + l\delta \\ &> 0 \end{aligned}$$

for sufficiently large l. Thus for sufficiently many iterations, the value of the root node in the computation tree after Algorithm 1 terminates is the same as the ML value for all $v \in V$. Therefore, the MSA converges to the ML decision. ◇

A.4.3 Proof of Lemma 6.3.3

Since the degree of every codeword node is 2, any walk with the same node appearing more than twice must be reducible.◇

A.4.4 Proof of Lemma 6.3.4

If $b_i = 1$, the codeword node only occurs once in S so it must be in a cycle in G' since S is a closed walk. Thus we only need to consider the case when $b_i = 2$.

Figure 6.5 shows an example of a pseudo-cycle where $b_7 = 2$ and there is no cycle in G', that contains v_7. There is also no row c^j in the codeword matrix C in (6.19), such that $\mathbf{b} - c^j \geq 0$. All that we need to now show, is that if a codeword node appears twice in a pseudo–cycle, then G' cannot contain only one cycle with that node, unless every codeword node in that cycle appears twice in S.

Relabel the nodes in S, $v_1, v_2, \ldots, v_L$. Now consider a codeword node $v_i = v_j$ for $i < j$, that appears twice in S. If node v_i is in a cycle in G', then there must exist a node v_p such that $i < p < j$ and $v_p = v_q$ for $q < i$ or $q > j$. Let $p > i$ be the smallest p for which this is true. WLOG assume $q < i$. Thus the graph induced by the closed walk $v_{q+1}, \ldots, v_i, \ldots, v_{p-1}$ has a cycle containing v_i, since v_i appears only once in the walk. Similarly, if we take the smallest r, such that $v_{p+r} \neq v_{q-r}$, the graph induced by the closed walk $v_{p+r}, \ldots, v_L, v_1, \ldots, v_{q-r}$ contains v_i only once and thus also has a cycle containing v_i. If these two cycles are distinct, then there are a c^l and a c^j, $j \neq l$ such that $\mathbf{b} - c^l \geq 0$ and $\mathbf{b} - c^j \geq 0$. If they are the same, then there is an c^l such that $\mathbf{b} - 2c^l \geq 0$. $\diamond$

A.4.5 Proof of Theorem 6.3.5

We shall prove the theorem by making use of the Farkas Alternative which states:

Either

1. *$C^T\mathbf{x} = \mathbf{b}$ has a solution $\mathbf{x} \geq 0$*

   or (exclusive)

2. *$C\mathbf{w} \geq 0$, $\mathbf{b}^T\mathbf{w} < 0$ has a solution $\mathbf{w}$.*

Assume there is a codeword node $v_i \in S$ that is not contained in any cycle in G'. By Lemma 6.3.4, $b_i = 2$, so there is no $c^l \in C$ such that $\mathbf{b} - c^l \geq 0$. Thus for every

row $c^l \in C$ for which $c_{l,i} = 1$, there is an $c_{l,j} = 1$, $j \neq i$, where $b_j = 0$. Let

$$w_j = \begin{cases} -1 & j = i \\ 1 & b_j = 0 \\ 0 & \text{otherwise.} \end{cases}$$

The vector $\mathbf{w}$ is a solution to the second Farkas alternative. Thus there is no solution to the primal problem $C^T\mathbf{x} = \mathbf{b}$, $\mathbf{x} \geq 0$ and the pseudo-cycle is bad.

Now assume every node $v_i \in S$ is in at least one cycle in G'. By Lemma 6.3.4, if $b_i = 2$, there are either two rows c^l, $c^j \in C$, such that $\mathbf{b} - c^l \geq 0$ and $\mathbf{b} - c^j \geq 0$ or there is a $c^l \in C$ such that $\mathbf{b} - 2c^l \geq 0$. Because of this, if we start with the vector $\mathbf{w} = [0, \ldots, 0]^T$, there is no way to find a solution $\mathbf{w}$ that satisfies $C\mathbf{w} \geq 0$ and $\mathbf{b}^T\mathbf{w} < 0$ by increasing or decreasing the coordinates of $\mathbf{w}$. Since the solution space of $C\mathbf{w} \geq 0$ is closed, this means there is no solution $\mathbf{w}$ for the second Farkas alternative. Therefore, there is a solution to the primal problem $C^T\mathbf{x} = \mathbf{b}$, $\mathbf{x} \geq 0$. The same is true if $b_i = 1$, so the pseudo-cycle is good. $\diamond$

A.4.6 Proof of Lemma 6.3.6

This lemma essentially appears in Wiberg [13, Lemma 6.2], except here we show that the pseudo-cycle must contain two *distinct* cycles.

Let S be a pseudo-cycle in G_T. Relabel the nodes in S, $v_1, v_2, \ldots, v_L$. Now consider a node $v_i = v_j$ for $i < j$, that appears twice in S. The closed walk $v_{i+1}, \ldots, v_{j-1}$, must contain a cycle since we must visit the check node connecting v_i and v_{i+1} twice. Similarly the closed walk $v_{j+1}, \ldots, v_L, v_1, \ldots, v_{j-1}$, must also contain a cycle since we visit the check node connecting v_{i-1} and v_i twice. Now if the node v_i is not contained in a cycle, then the two closed walks must contain disjoint cycles. If the node v_i is contained in a cycle, then we have a second partially overlapping cycle in the graph that contains the node v_i. $\diamond$

A.4.7 Proof of Lemma 6.3.7

Since d is the minimum distance of the code, it is also the girth of G_T. By Lemma 6.3.6, the subgraph G' induced by S contains at least 2 cycles of minimum length d which may be partially overlapping or disjoint. If the cycles are disjoint, then $|V'| > 2d$ and the lemma is true, so we need only consider the case in which the cycles are partially overlapping.

Suppose the 2 cycles share $> \frac{d}{2}$ vertices. We can then construct a cycle, using the unshared vertices, which has length $< d$. Since this is impossible, the cycles can only share $\leq \frac{d}{2}$ vertices and thus $|V'| \geq \frac{3}{2}d$. $\diamond$

A.4.8 Proof of Lemma 6.3.8

Let l be the number of codeword nodes that appear twice in the pseudo–cycle. If $G' = (V', E')$ is the subgraph induced by the pseudo–cycle S in G, then $|V'| - l$ codeword nodes will appear only once. The pseudo–weight,

$$\mathcal{W}(S) = \frac{((|V'| - l) + 2l)^2}{(|V'| - l) + 4l} \tag{A.12}$$

$$\geq \frac{8}{9}|V'| \tag{A.13}$$

$$\geq \frac{4}{3}d, \tag{A.14}$$

where (A.13) follows from the inequality $(a + 2b)^2/(a + 4b) \geq \frac{8}{9}(a + b)$ and (A.14) follows from Lemma 6.3.7. $\diamond$

A.4.9 Proof of Theorem 6.3.9

From Theorem 6.3.5, we know that a pseudo-cycle S is bad iff it contains two disjoint closed walks connected by a path on which each node is visited twice. To prove the theorem we will first show that the shorter the connecting path, the lower the pseudo-weight of the pseudo-cycle. We will then consider the pseudo-weights for the different possible types of pseudo-cycles.

Suppose the pseudo-weight of S is $\mathcal{W}(S) = a_1^2/a_2$. If a codeword node is added to the path connecting the two closed walks, then the new pseudo-cycle S' will have pseudo-weight $\mathcal{W}(S')$ where

$$\begin{aligned} \frac{\mathcal{W}(S')}{\mathcal{W}(S)} &= \frac{(a_1+2)^2}{a_2+4} \div \frac{a_1^2}{a_2} && \text{(A.15)} \\ &\geq 1, && \text{(A.16)} \end{aligned}$$

since by definition of the pseudo-weight $a_2 \geq a_1$. So in order to minimize the pseudo-weight of S, we will assume the path connecting the two disjoint closed walks is of length 1.

The closed walks joined by the path can themselves be pseudo-cycles or just cycles. Let's first consider the case when both closed walks are cycles of length d_1 and d_2 respectively. We have

$$\begin{aligned} \mathcal{W}(S) &= \frac{(d_1+d_2+2)^2}{d_1+d_2+4} && \text{(A.17)} \\ &= \frac{(d_1+d_2)^2+4(d_1+d_2)+4}{d_1+d_2+4} && \text{(A.18)} \\ &> d_1+d_2 \geq 2d. && \text{(A.19)} \end{aligned}$$

When at least one of the closed walks is a pseudo-cycle, we can use the same argument as in Lemma 6.3.8 to show that

$$\begin{aligned} \mathcal{W}(S) &\geq \frac{8}{9}\,|V'| && \text{(A.20)} \\ &\geq \frac{20}{9}\,d, && \text{(A.21)} \end{aligned}$$

since $|V'| \geq d + 3/2d$. Thus the minimum pseudo-weight of a bad pseudo-cycle is greater than twice the minimum distance distance of the code. $\diamond$

Appendix B

Definition of a Commutative Semi–ring

The local kernels in a junction graph and the weights assigned to the edges of a directed graph both take values in a commutative semi–ring R. Here we present the definition of a semi–ring from Aji and McEliece [16].

Definition: A *commutative semi–ring* is a set R, with two binary operations denoted "+" and "·", which satisfy the following three axioms:

A1 The operator "+" is associative and commutative with an identity element "0". Thus $(R, +)$ is a commutative monoid.

A2 The operator "·" is also associative and commutative with an identity element "1". Thus $(R, \cdot)$ is also a commutative monoid.

A3 The distributive law

$$(x + y) \cdot z = x \cdot z + y \cdot z,$$

holds for all triples $(x, y, z) \in R$.

The difference between a ring and a semi–ring is that the addition operator in a semi–ring does not require an inverse, i.e., $(R, +)$ must be a commutative monoid and not necessarily a group.

Two examples of semi–rings are the *sum–product* semi–ring where $R = \mathbb{R}^{\geq 0}$ with ordinary addition and multiplication, and the *min–sum* semi–ring where $R = \mathbb{R} \cup \infty$ where $x + y = \min(x, y)$ with identity element ∞ and $x \cdot \infty = \infty$ and the operator $\cdot$ is ordinary addition with identity 0.

Bibliography

[1] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding," in *Proc. IEEE Int. Comm. Conf.*, (Geneva, Switzerland), pp. 1064–1070, 1998.

[2] R. G. Gallager, *Low Density Parity Check Codes.* Cambridge, MA: M. I. T. Press, 1963.

[3] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "Serial concatenation of interleaved codes: Performance analysis, design and iterative decoding," *IEEE Trans. on Info. Theory*, vol. 44, pp. 909–926, May 1998.

[4] D. Divsalar, H. Jin, and R. J. McEliece, "Coding theorems for "turbo-like" codes," in *Proc. 36th Allerton Conf. on Comm., Control and Computing*, (Allerton, IL), Sept. 1998.

[5] S. Benedetto and G. Montorsi, "Unveiling turbo codes: Some results on parallel concatenated coding schemes," *IEEE Trans. on Info. Theory*, vol. 42, pp. 409–428, Nov. 1996.

[6] D. Divsalar and F. Pollara, "Turbo codes for deep-space communications," *TDA Progress Report*, vol. 42–120, pp. 29–39, Feb. 1995.

[7] D. Divsalar, S. Dolinar, R. J. McEliece, and F. Pollara, "Transfer function bounds on the performance of turbo codes," *TDA Progress Report*, vol. 42–122, pp. 44–55, July 1995.

[8] D. Divsalar and R. J. McEliece, "Effective free distance of turbo codes," *Electronics Letters*, vol. 32, pp. 445–446, Feb. 1996.

[9] J. Hagenauer, E. Offer, and L. Papke, "Iterative decoding of binary block and convolutional codes," *IEEE Trans. on Info. Theory*, vol. 42, pp. 429–445, Mar. 1996.

[10] D. J. C. MacKay, "Good error correcting codes based on very sparse matrices," *IEEE Trans. on Info. Theory*, vol. 45, pp. 399–431, Mar. 1999.

[11] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. A. Spielman, "Analysis of low density codes and improved designs using irregular graphs." Available at `http://www.icsi.berkeley.edu/ luby/`.

[12] T. Richardson and R. Urbanke, "The capacity of low-density parity check codes under message-passing decoding." Available at `http://cm.bell-labs.com/who/tjr/pub.html`.

[13] N. Wiberg, *Codes and Decoding on General Graphs.* PhD thesis, Linköping University, Sweden, 1996.

[14] M. C. Davey and D. J. C. MacKay, "Low density parity check codes over GF(q)," *IEEE Communications Letters*, June 1998.

[15] T. Richardson, A. Shokrollahi, and R. Urbanke, "Design of provably good low density parity check codes." Available at `http://cm.bell-labs.com/who/tjr/pub.html`.

[16] S. M. Aji and R. J. McEliece, "The generalized distributive law." Submitted to IEEE Trans. on Info. Theory, Available at `www.systems.caltech.edu/EE/Faculty/rjm/`.

[17] F. V. Jensen, *An Introduction to Bayesian Networks.* New York: Springer-Verlag, 1996.

[18] R. M. Tanner, "A recursive approach to low complexity codes," *IEEE Trans. on Info. Theory*, vol. 27, pp. 533–547, Sept. 1981.

[19] S. M. Aji, G. B. Horn, and R. J. McEliece, "On the convergence of iterative decoding on graphs with a single cycle," in *Proc. CISS*, (Princeton, NJ), pp. 428–434, Mar. 1998.

[20] S. M. Aji, G. B. Horn, and R. J. McEliece, "Iterative decoding on graphs with a single cycle," in *Proc. ISIT*, (Boston, MA), p. 276, Aug. 1998.

[21] S. M. Aji, G. B. Horn, R. J. McEliece, and M. Xu, "Iterative min-sum decoding of tail-biting codes," in *Proc. 36th Allerton Conf. on Comm., Control and Computing*, (Allerton, IL), Sept. 1998.

[22] S. M. Aji, G. B. Horn, R. J. McEliece, and M. Xu, "Iterative min-sum decoding of tail-biting codes," in *Proc. IEEE Information Theory Workshop*, (Killarney, Ireland), pp. 68–69, June 1998.

[23] G. B. Horn and R. J. McEliece, "The iterative decoding of cycle codes," in *Proc. CISS*, (Baltimore, MD), Mar. 1999.

[24] R. G. Gallager, "Low density parity check codes," *IRE Trans. on Info. Theory*, vol. 8, pp. 21–28, Jan. 1962.

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

[26] G. D. Forney Jr., "The forward-backward algorithm," in *Proc. 34th Allerton Conference on Communication, Control and Computing*, pp. 432–446, 1996.

[27] L. E. Baum and T. Petrie, "Statistical inference for probabilistic functions of finite-state markov chains," *Ann. Math. Stat.*, vol. 37, pp. 1559–1563, 1966.

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

[29] F. R. Kschischang and B. Frey, "Iterative decoding of compound codes by probability propagation in graphical models," *IEEE Journal on Selected Areas in Communication*, vol. 16, pp. 219–230, Feb. 1998.

[30] R. J. McEliece, D. J. C. MacKay, and J.-F. Cheng, "Turbo decoding as an instance of Pearl's 'belief propagation' algorithm," *IEEE Journal on Selected Areas in Communication*, vol. 16, pp. 140–153, Feb. 1998.

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

[32] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger, "Factor graphs and the sum product algorithm." Submitted to IEEE Trans. on Info. Theory, July 1998.

[33] G. R. Shafer and P. P. Shenoy, "Probability propagation," *Annals of Mathematics and Artificial Intelligence*, vol. 2, pp. 327–352, 1990.

[34] N. Wiberg, H.-A. Loeliger, and R. Kötter, "Codes and iterative decoding on general graphs," *European Transactions on Telecommunications*, vol. 6, pp. 513–525, Sept. 1995.

[35] A. R. Calderbank, G. D. Forney Jr., and A. Vardy, "Minimal tail-biting trellises: The Golay code and more." Submitted to IEEE Trans. on Info. Theory, Aug. 1997.

[36] G. Solomon and H. C. A. van Tilborg, "A connection between block and convolutional codes," *SIAM Journal on Applied Mathematics*, vol. 37, pp. 358–369, Oct. 1979.

[37] W. W. Peterson and E. J. Weldon Jr., *Error-correcting Codes.* Cambridge, MA: M. I. T. Press, 1972.

[38] Y. Weiss, "Correctness of local probability propagation in graphical models with loops." To appear in Neural Computation.

[39] R. Motwani and P. Raghavan, *Randomized Algorithms.* Cambridge, England: Cambridge University Press, 1995.

[40] J. B. Anderson and S. M. Hladik, "Tailbiting MAP decoders," *IEEE Journal on Selected Areas in Communication*, vol. 16, pp. 297–302, Feb. 1998.

[41] R. J. McEliece, E. R. Rodemich, and J.-F. Cheng, "The turbo decision algorithm," in *Proc. 33rd Allerton Conference on Communication, Control and Computing*, pp. 366–379, 1995.

[42] D. Lind and B. Marcus, *Symbolic Dynamics and Coding.* Cambridge, England: Cambridge University Press, 1995.

[43] R. A. Horn and C. R. Johnson, *Matrix Analysis.* Cambridge, England: Cambridge University Press, 1985.

[44] S. M. Aji. Personal Communication.

[45] G. D. Forney Jr., F. R. Kschischang, and B. Marcus, "Iterative decoding of tail-biting trellises," in *Proc. IEEE Information Theory Workshop*, (San Diego, CA), pp. 11–12, Feb. 1998.

[46] R. P. Stanley, *Enumerative Combinatorics*, vol. 1. Cambridge, England: Cambridge University Press, 1997.

[47] J. H. van Lint and R. M. Wilson, *A Course in Combinatorics.* Cambridge, England: Cambridge University Press, 1992.

[48] R. J. McEliece, "How to compute weight enumerators for convolutional codes," in *Communications and Coding* (M. Darnell and B. Honary, eds.), (Somerset, England), pp. 121–141, Research Studies Press Ltd., 1998.

[49] G. D. Forney Jr., F. R. Kschischang, and A. Reznik, "The effective weight of pseudocodewords for codes defined on graphs with cycles on AWGN channels." Preprint, 1998.

[50] R. Kötter and A. Vardy. Personal Communication.

[51] J. B. Anderson and K. E. Tepe, "Properties and error performance of the tail-biting BCJR decoder." Submitted to IEEE Trans. on Info. Theory, Oct. 1998.

[52] R. Johannesson and K. S. Zigangirov, *Introduction to Convolutional Coding.* Piscataway, NJ: IEEE Press, 1999.

[53] S. L. Hakimi and J. G. Bredeson, "Graph theoretic error-correcting codes," *IEEE Trans. on Info. Theory*, vol. 14, pp. 584–591, July 1968.

[54] D. Jungnickel and S. A. Vanstone, "Graphical codes revisited," *IEEE Trans. on Info. Theory*, vol. 43, pp. 136–146, Jan. 1997.

[55] N. Christofides, *Graph Theory: An Algorithmic Approach.* New York: Academic Press, 1975.

[56] J. Franklin, *Methods of Mathematical Economics.* New York: Springer Verlag, 1980.