Spring 1-24-2010

Finite Tree-Based Decoding of Low-Density Parity-Check Codes

Eric T. Psota
University of Nebraska at Lincoln, epsota24@huskers.unl.edu

Finite Tree-Based Decoding of Low-Density Parity-Check Codes

A Dissertation

by

Eric T. Psota

Presented to the Faculty of

The Graduate College at the University of Nebraska

In partial Fulfillment of Requirements

For the Degree of Doctor of Philosophy

Major: Engineering

Under the Supervision of Professor Lance C. Pérez

Lincoln, Nebraska

January, 2010

Finite Tree-Based Decoding of Low-Density Parity-Check Codes

Eric T. Psota, Ph.D.

University of Nebraska, 2009

Advisor: Lance C. Pérez

Low-density parity-check codes are commonly decoded using iterative message-passing decoders, such as the min-sum and sum-product decoders. Computer simulations demonstrate that these suboptimal decoders are capable of achieving low probability of bit error at signal-to-noise ratios close to capacity. However, current methods for analyzing the behavior of the min-sum and sum-product decoders fails to produce usable bounds on the probability of bit error. Thus, the resulting probability of bit error when using these decoders remains largely unknown for signal-to-noise ratios beyond the reach of simulation. For this reason, it is worth considering alternative methods for decoding low-density parity-check codes.

New methods for decoding low-density parity-check codes, known as finite tree-based decoders, are presented as alternative decoders for low-density parity-check codes. The goal of the finite tree-based decoders is to achieve probability of bit error comparable to that of the min-sum and sum-product decoders, while allowing for computationally tractable performance analysis. Finite tree-based decoding requires the construction of finite trees derived from the Tanner graph of the low-density parity-check code. The resulting size of the finite trees allows for current analytical techniques, such as deviation bounds and density evolution, to be used to predict the probability of bit error of finite tree-based decoding of short-to-moderate length low-density parity-check codes. Simulation results show that finite tree-based decoders are capable of outperforming current iterative decoders at high signal-to-noise ratios. Examples are also given where finite tree-based decoding provably approaches maximum-likelihood performance as the signal-to-noise ratio grows large.

A new method is also presented for lower bounding the minimum distance of low-density parity-check codes. This new lower bound is used as a cost criteria for the construction of low-density parity-check codes with both large girth and minimum-distance properties. Codes generated with this new construction technique are shown in simulations to outperform codes generated with the progressive edge-growth algorithm, using both iterative decoding and finite tree-based decoding.

Acknowledgment

I need to thank many people for helping me get to this point. First of all, I would like to thank my advisor Dr. Lance C. Pérez. He has been pushing me to become a better researcher since the first day I started working for him. He also has a talent for helping me think outside the box whenever I get stuck in a rut. It is invaluable to have someone like that during a process as long and difficult as creating a dissertation.

I would also like to thank the rest of my committee including Dr. Khalid Sayood, Dr. Mustafa Cenk Gursoy, and Dr. Judy Walker. In particular, I need to give a huge thanks to Dr. Walker along with her students Deanna Dreher, Katie Morrison, Nathan Axvig, and her former student Emily Price from the Mathematics Department at the University of Nebraska-Lincoln. Without the experience of working with them for more than two years, this dissertation certainly would not have been possible. They helped me better understand and appreciate channel codes, decoding, and all of the underlying mathematics involved.

I need to thank my family and friends for all their amazing support. My mother and father believed in me long before I started to believe in myself. I also need to thank my grandmother Vernice Psota, who passed away in September of 2009. She will always be my 'pal', and she will be dearly missed.

Finally, I want to dedicate this dissertation to my wife Alyssa. We were married in May of 2007, one year after I started in the doctoral program. Immediately after-

wards we were forced to live apart during the first year of our marriage. Despite our situation, we have stayed strong through it all, and become even stronger because of it. In December 2009 she was deployed to Iraq as a 1st Lieutenant in the United States Air Force. I could not be more proud of her and and way she approaches her job every day. I miss her every day, and I am anxiously awaiting her safe return.

To Alyssa

Contents

List of Figures

List of Tables

Glossary of Notation

N :	code block length
K :	code dimension
C :	set of all codeword vectors in the code
G :	generator matrix
H :	parity-check matrix
M :	number of rows in the parity-check matrix
$\mathbf{u}$:	information sequence
$\mathbf{c}$:	codeword
$\mathbf{x}$:	modulated codeword
$\mathbf{n}$:	channel noise
σ :	channel noise standard deviation
$\mathbf{y}$:	received channel output
$\boldsymbol{\lambda}$:	log-likelihood ratio of the received channel output
$\hat{\mathbf{x}}$:	modulated codeword estimate
$\hat{\mathbf{c}}$:	codeword estimate
$\mathbf{s}$:	syndrome of a vector
$\hat{\mathbf{u}}$:	information sequence estimate
T :	Tanner graph of a code
F :	set of check nodes in the Tanner graph

V :	set of variable nodes in the Tanner graph
E :	set of edges in the Tanner graph
d_V :	degree of all variable nodes in a bi-regular Tanner graph
d_F :	degree of all check nodes in a bi-regular Tanner graph
$N(i)$:	set of neighbors of i
S :	a single odd-sized set containing the neighbors of i
$\bar{S}$:	set of neighbors of i not contained in S
$\mathcal{E}_i$:	all odd-sized sets containing the neighbors of i
$m_{v\to f}$:	message passed from v to f
$m_{f\to v}$:	message passed from f to v
m_v :	message computed at the root node v
ℓ :	decoding iterations
$\ell_{\max}$:	maximum number of decoding iterations
$\mathcal{S}$:	set of variable nodes in a stopping set
$\mathcal{T}$:	set of variable nodes in a trapping set
t :	number of elements in a trapping set
w_t :	weight of the syndrome of a trapping set
p_{BSC} :	BSC crossover probability
$R_v^{(\ell)}$:	computation tree rooted at variable node v after ℓ iterations
$R_v^{(\ell)}(V)$:	set of variable node in the computation tree $R_v^{(\ell)}$

$R_v^{(\ell)}(F)$:	set of check nodes in the computation tree $R_v^{(\ell)}$
$\overline{R}_v^{(\ell)}(V)$:	set of variable node not in the computation tree $R_v^{(\ell)}$
$\overline{R}_v^{(\ell)}(F)$:	set of check nodes not in the computation tree $R_v^{(\ell)}$
Δ :	set of all deviations on a computation tree
$G(\delta)$:	cost of a deviation δ
η_i :	fraction of edges connected to variable nodes of degree i
ρ_i :	fraction of edges connected to check nodes of degree i
$\mathcal{P}$:	probability density function of $m_{v\to f}$
$\mathcal{Q}$:	probability density function of $m_{f\to v}$
$C(i)$:	set of child nodes connected to node i
P_b :	probability of bit error
E_i :	event that configuration i has minimal cost
B_i :	event that configuration i has cost less than the all-zeros configuration
$\overline{X}$:	estimated mean of the probability of error at the root node of an iterative tree
μ_X :	true mean of the probability of error at the root node of an iterative tree
S_X^2 :	sample variance of the $\overline{X}$
$\mathcal{C}_{v_i}$:	set of all valid configurations on a finite tree rooted at variable node v_i
$\varsigma_{v_i,j}$:	j^{th} valid configuration in $\mathcal{C}_{v_i}$
$w(i)$:	weight of a configuration i
$\mathcal{W}$:	number of deviation weights tracked during extrinsic tree construction

b_i :	number of copies of variable node v_i in an extrinsic tree
a_i :	number of copies of variable node v_i in a deviation on the extrinsic tree
$\mathcal{G}$:	girth of a Tanner graph

Chapter 1

Introduction

In 1948, Claude E. Shannon published "A Mathematical Theory of Communication" [1] and sparked the beginning of the field of information theory. Prior to Shannon's work, increasing the reliability of communication over a noisy channel was most commonly done by dedicating more power to the transmitter to increase the signal-to-noise ratio (SNR). Shannon's noisy-channel coding theorem showed that, for a given channel, there exist channel codes capable of achieving arbitrarily low error rates as long as the rate of the code was below the channel capacity. Channel codes that achieve a probability of bit error below 10^{-5} at SNR only slightly above the minimum, as defined by Shannon, are often referred to as *capacity-achieving* codes. Unfortunately, Shannon failed to provide a method for constructing usable capacity-achieving channel codes.

Some of the first channel codes to be used were binary Hamming codes, introduced in 1950 by Hamming [2]. Hamming codes use binary generator matrices for encoding information sequences and binary parity-check matrices for decoding. While far from capacity-achieving codes, Hamming codes are nonetheless effective in dealing with channel errors for short block lengths. The minimum-distance properties of Hamming codes combined with the hard-decision syndrome decoder makes it possible to efficiently detect, and sometimes correct, bit errors introduced by the channel.

Shortly after the introduction of Hamming codes, convolutional codes were developed by Elias in 1955 [3]. Unlike Hamming codes, convolutional codes generate codewords by convolving short binary sequences with the information bits. The binary numbers in the generating sequence are often referred to as the memory elements. Convolutional codes with a large number of memory elements, or simply large memory, have the potential for increased minimum distance when compared to convolutional codes with small memory. *Minimum distance* refers to the minimum number of non-zero elements in any codeword, excluding the codeword consisting of all zeros. Codes with large minimum distance are known to have a lower probability of bit error at high SNR, when decoded with maximum likelihood decoding. The structure of convolutional codes makes them very efficient for encoding information sequences of any given length. However, it was not until 1967 that soft-decision maximum-likelihood decoding of convolutional codes became possible with the introduction of the Viterbi decoder [4]. While both Hamming codes and convolutional codes are efficient for encoding and decoding short block length codes, the computational complexity of their respective decoders made them impractical for larger block lengths.

Fourteen years after Shannon proved the existence of capacity-achieving codes, Gallager [5] discovered low-density parity-check (LDPC) codes. Gallager was able to prove that LDPC codes are capable of capacity-achieving performance. However, due to the limitations of computers at the time of their discovery, it was impractical to simulate the encoding and decoding of large block length LDPC codes. Therefore, nobody was able to demonstrate the exceptional performance of LDPC codes with large block lengths, and they were largely forgotten for the next three decades.

In 1993, there was a breakthrough in the field of channel coding when C. Berrou, A. Glavieux, and P. Thitimajshima discovered a class of near-capacity achieving codes known as turbo codes [6]. Unlike LDPC codes, the discovery of turbo codes came with simulations demonstrating their exceptionally low probability of bit error at SNRs close to Shannon's capacity bound. A fundamental innovation of turbo codes was the use of a suboptimal, soft-decision iterative decoder. Prior to the introduction of turbo codes, suboptimal iterative decoders were not commonly used in the field of channel coding.

In 1996, D. J. C. MacKay and R. M. Neal revisited LDPC codes [7]. By using existing methods for suboptimal iterative decoding [8], they were able to demonstrate through simulations that LDPC codes are also capable of achieving near-capacity performance. Since then, LDPC codes have surpassed turbo codes in terms of both research activity and application. In 2003, an LDPC code was chosen over a turbo code for digital video broadcasting - satellite 2$^{\text{nd}}$ generation (DVB-S2) systems [9]. The LDPC code was chosen for two reasons. First, the LDPC code uses a lower-complexity decoder than turbo codes. Second, turbo codes exhibit a phenomenon known as an *error floor* where, at some frequently unknown SNR, the performance becomes dominated by the relatively small minimum distance of the code and the bit error rate decreases much more slowly with increasing channel SNR.

Since the discovery of turbo codes and the rediscovery of low-density parity-check codes, a great deal of research has been devoted to understanding the behavior of their suboptimal iterative decoders. The Turbo decoder is a message-passing algorithm that utilizes the well-known BCJR algorithm [10]. Turbo codes are encoded with a parallel concatenation of component convolutional codes, and the BCJR algorithm is used to minimize the probability of bit error of each component code. During turbo decoding, the BCJR outputs are passed between the component decoders. Though there is no guarantee of an optimal output using the turbo decoder, the behavior of turbo decoding as the SNR grows large can be closely approximated by assuming that the output of the turbo decoder is the same as that of the maximum-likelihood decoder [11]. Turbo decoding performance often has a poor probability of bit error at high SNR due to the existence of error floors. Therefore, turbo codes are not suitable for applications requiring exceptionally low error rates.

The two best known decoders for low-density parity-check codes are the sum-product (SP) decoder (also known as the belief propagation decoder) and the min-sum (MS) decoder. These two decoders are optimal for codes whose Tanner graph [12] representation is a tree. MS decoding is optimal in the sense that it minimizes the codeword error probability on the tree, and SP decoding is optimal in the sense that it minimizes the bit error probability on the tree [13]. Unfortunately, codes whose Tanner graphs are trees have inherently poor distance spectrums due to the existence of leaf nodes, and are not suitable for applications requiring a low probability of bit error. For this reason, it is necessary to consider codes whose graphical realizations contain cycles and are thus not trees. The MS and SP decoders are no longer optimal on codes whose graphical representations contain cycles. In addition, performance analysis is difficult when the Tanner graph of an LDPC code contains cycles.

In his dissertation, Wiberg [13] introduced the *computation tree* as an exact model for the sum-product and min-sum decoders after a finite number of iterations, even in situations where the Tanner graph of the low-density parity-check code contains cycles. Therefore, in theory the computation tree enables one to analyze the performance of the SP and MS decoders after any number of iterations. However, regardless of the block length of the code, the size of the computation tree grows exponentially with the check node and variable node degrees of the LDPC code, thus making analysis intractable even after just a small number of iterations.

It is not computationally tractable to obtain an accurate probability of bit error for SP and MS decoding over all ranges of SNR using computer simulations. Therefore analytical results are needed to estimate the probability of bit error at SNR where simulations are incapable of providing an estimate. However, even with the benefit of modern methods for characterizing the error patterns of iterative decoders [14], the behavior of the MS and SP decoders remain largely a mystery to the coding community.

Current methods for decoding low-density parity-check codes are difficult to analyze. Thus, it is of interest to formulate an alternative decoder whose performance is comparable to that of the SP and MS decoders, but which allows for tractable performance analysis. For applications that require near-error-free performance it is necessary to accurately predict or upper bound the performance beyond the range of computer simulations. Finite tree-based decoding methods are introduced in this dissertation as alternatives to current suboptimal, iterative decoders in an effort to create LDPC decoders that allow for tractable performance analysis.

Each of the finite tree-based decoding methods presented includes both a finite tree construction and a finite tree decoder. Several different methods for finite tree construction are introduced and examined in this dissertation, including extrinsic tree construction [15, 16]. Each of the new finite tree construction methods builds trees based on the Tanner graph of a given LDPC code, and the trees are constructed prior to decoding. It is important to note that, for a given code, it is only necessary to perform the finite tree construction once. After the finite tree construction is complete, the finite tree decoder operates on the trees using channel output information. The construction of the finite trees is done with the goal of minimizing the probability of a bit error at the root node, with the knowledge that they will be operated on by the finite tree decoder. In addition to minimizing the probability of a bit error, finite tree construction methods also attempt to minimize the number of nodes necessary to achieve a given probability of bit error at the root node. Simulations demonstrate that the new finite tree-based decoders are capable of outperforming existing iterative decoders at high SNR.

In this dissertation, upper bounds are given on the probability of bit error for finite tree-based decoding of codes with short-to-moderate block length ($N \leq 1500$) and fixed variable node and check node degrees. Existing methods for analyzing and upper bounding the performance of MS and SP decoding of LDPC codes, after only a small number of iterations, are applied to the finite tree-based decoders. For example, on a finite tree with a small enough number of nodes, Wiberg's notion of deviations can be used to compute upper bounds on the performance of finite tree-based decoders even when the finite trees contain more than one copy of the same variable node. In the special case where there is a maximum of one copy of each variable node in the finite tree, density evolution [17, 18] can be used to compute the exact probability of bit error of finite tree-based decoders.

There are several existing methods for constructing low-density parity-check codes. Some of the best known methods [19] create codes with a desirable cycle structure in the Tanner graph. Here, a new method referred to as independent tree-based LDPC (ITB LDPC) construction is presented. ITB LDPC codes are iteratively constructed with the goal of increasing the lower bound on the minimum distance. Instead of using existing methods for obtaining lower bounds on the minimum distance, a new finite tree-based method for lower bounding the minimum distance is derived. It is shown that the new ITB LDPC codes perform better in terms of both probability of bit error and word error than codes constructed with existing methods, using both iterative decoders and finite tree-based decoders. Simulations using finite tree-based decoding of ITB LDPC codes demonstrate improved performance when compared to the decoding of LDPC codes generated using existing methods. In one example, a finite tree-based construction method detects the minimum-weight codeword in a code of dimension fifty, and as a consequence the finite tree decoder provably achieves the bit error rate performance of a maximum-likelihood decoder as the SNR grows large.

In summary, this dissertation derives several novel methods for decoding low-density parity-check codes. Each of the new finite tree-based decoding methods constructs finite trees derived from the original Tanner graph of the code before the finite tree decoder operates on the trees. The trees are constructed with the goal of minimizing the probability of bit error at the root node. The trees are also limited in size, thus allowing tractable performance analysis even at SNRs beyond the reach of simulation. A new method is also introduced for constructing LDPC codes that perform well with finite tree-based decoders. Simulations are presented that demonstrate that finite tree-based decoders can outperform MS decoding at high channel SNRs. Examples are also given where finite tree-based decoding provably achieves maximum-likelihood decoding probability of bit error as the channel SNR grows large.

Chapter 2 provides much of the necessary background information for understanding the finite tree-based decoders. Several existing decoders are discussed, along with different methods for analyzing the error mechanisms of these decoding algorithms. With the necessary background in place, motivation for finite tree-based decoding is given at the end of Chapter 2. In Chapter 3, several methods for finite tree-based decoding are introduced. Chapter 3 concludes with the examination of two different analytical methods that can be used to predict the performance of finite-tree based decoding. Chapter 4 focuses on the design of low-density parity-check codes that perform well with finite tree-based decoders. A new lower bound on the minimum distance based upon finite trees is introduced. The new lower bound is then utilized to construct parity-check matrices that perform well with finite tree-based decoders. Finally, Chapter 5 concludes with some discussion and suggestions for future research.

Chapter 2

Background

The near-capacity performance of low-density parity-check codes was first demonstrated with efficient decoding algorithms like min-sum (MS) and sum-product (SP). These two decoding algorithms take advantage of the sparseness of the parity-check matrices of LDPC codes, since the complexity of the MS and SP algorithms scales in proportion to the number of binary 1's in the parity-check matrix. However, due to the sub-optimality of the MS and SP decoders, new decoding algorithms for LDPC codes have been created along with variations of the SP and MS decoders in an attempt to improve their performance. In this chapter, several methods for decoding LDPC codes are reviewed along with existing techniques for performance analysis.

This chapter begins with a definition of the binary-input, additive white Gaussian noise (BIAWGN) channel. The BIAWGN channel model is assumed throughout this dissertation. After introducing the channel model, the necessary background on the graphical representation of LDPC codes is given. Next, several known decoding algorithms for LDPC codes on the BIAWGN channel are discussed. These include the maximum-likelihood (ML), linear programming (LP), MS, and SP decoding algorithms. Several existing models for understanding the error mechanisms of iterative decoders are then considered. These include stopping sets, trapping sets, and deviations on the computation tree. Density evolution is then examined as a method for estimating the performance of code ensembles. Finally, a summary of the current state of research in this area is given along with a discussion of the motivation for finite tree-based decoding.

2.1 Channel Coding Over the Binary-Input Additive White Gaussian Noise Channel

A system diagram with channel coding over the binary-input, additive white Gaussian noise channel is given in Figure 2.1. A vector $\mathbf{u} \in \mathbb{F}_2^K$ of K information bits is generated by the binary source. The binary source is assumed to be memoryless, which is often the result of source coding (data compression), and therefore all information sequences in $\mathbb{F}_2^K$ are equally probable. A binary $K \times N$ generator matrix G may be used by the channel encoder to map the information bits $\mathbf{u}$ to a codeword $\mathbf{c} \in \mathbb{F}_2^N$, where

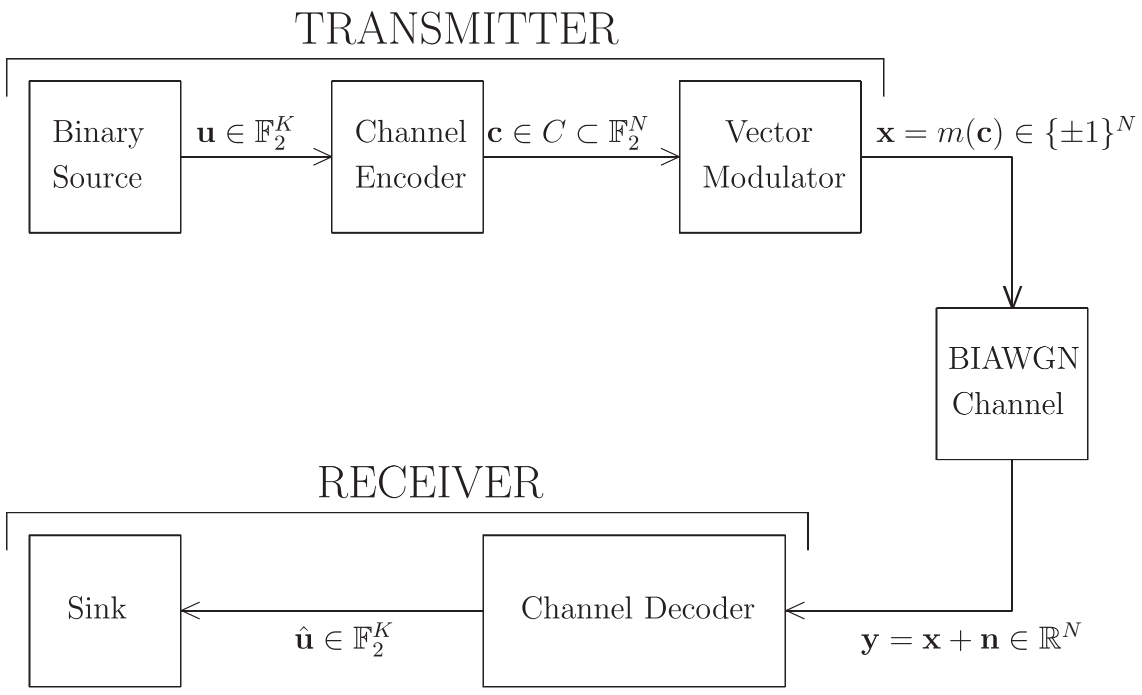

TRANSMITTER
Binary Source
$\mathbf{u} \in \mathbb{F}_2^K$
Channel Encoder
$\mathbf{c} \in C \subset \mathbb{F}_2^N$
Vector Modulator
$\mathbf{x} = m(\mathbf{c}) \in \{\pm 1\}^N$
BIAWGN Channel
RECEIVER
Sink
$\hat{\mathbf{u}} \in \mathbb{F}_2^K$
Channel Decoder
$\mathbf{y} = \mathbf{x} + \mathbf{n} \in \mathbb{R}^N$

Figure 2.1: System diagram for channel coding over the BIAWGN channel.

N is the block length of the code. If the matrix G is full-rank, the rate of the code is $R = \frac{K}{N}$. The mapping from information bits to codeword is done through the matrix multiplication $\mathbf{c} = \mathbf{u}G$. It is assumed that G is of the form $G = \left[I_{K \times K} : P_{K \times (N-K)}\right]$, where $I_{K \times K}$ is an identity matrix of size $(K \times K)$. This particular form makes it is easy to extract the information bits from the codeword, since $(c_1, \ldots, c_K) = \mathbf{u}$. Generator matrices with this form are referred to as systematic generator matrices.

Before a codeword $\mathbf{c} \in C$ is transmitted over the channel, it is mapped to a modulated vector via the transformation

$$x_i = m(c_i) = (2c_i - 1),$$

for all $i = 1, \ldots, N$. The received signal vector $\mathbf{y} \in \mathbb{R}^N$ is given by

$$\mathbf{y} = \mathbf{x} + \mathbf{n},$$

where $\mathbf{n} \in \mathbb{R}^N$ is the noise vector. The noise is assumed to be additive white Gaussian noise (AWGN) with zero mean and variance σ^2. An estimate $\hat{\mathbf{c}}$ of the transmitted codeword $\mathbf{c}$ is derived from the received vector $\mathbf{y}$ at the channel decoder. Since the generator matrix is systematic, the estimated information bits are simply $\hat{\mathbf{u}} = (\hat{c}_1, \ldots, \hat{c}_K)$. Finally, the information bits are passed to the sink.

As mentioned earlier, the assumption is made throughout this dissertation that the information sequences $\mathbf{u} \in \mathbb{F}_2^K$ are equiprobable. Since there is a one-to-one mapping between information sequences and codewords, all codewords in the code C are equiprobable as well. Therefore, $P(\mathbf{c}_i) = P(\mathbf{c}_j)$ for all $\mathbf{c}_i, \mathbf{c}_j \in C$, where $P(\mathbf{c}_i)$ is the probability that codeword $\mathbf{c}_i$ is transmitted.

2.2 Low-Density Parity-Check Matrices and Tanner Graphs

From the generator matrix G, it is possible to derive an $(N-K) \times N$ parity-check matrix H for the code. A *parity-check matrix* of a code C is any matrix H, such that $H\mathbf{c}^T = \mathbf{0}$ for all $\mathbf{c} \in C$. A systematic parity-check matrix $H = \left[(P_{K\times(N-K)})^T : I_{(N-K)\times(N-K)}\right]$, where $(P_{K\times(N-K)})^T$ is the transpose of $P_{K\times(N-K)}$, can easily be obtained from a systematic generator matrix G.

Low-density parity-check codes are often defined by their parity-check matrix H. In particular, LDPC codes are a class of codes with sparse parity-check matrices. A *sparse* parity-check matrix is any binary matrix that contains more binary 0's than binary 1's. A (d_V, d_F)*-regular* LDPC code is one that has a fixed number d_V of binary 1's in each column of the parity-check matrix and some fixed number d_F of binary 1's in each row of the parity-check matrix. These LDPC codes are often referred to as (d_V, d_F)-regular LDPC codes. An example of a small $(2,3)$-regular LDPC code of length $N = 6$ and dimension $K = 3$ is given by the parity-check matrix

$$H_{(2,3)} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}. \tag{2.1}$$

The parity-check matrix of a length N, dimension K code must contain at least $(N-K)$ rows, since the kernel of G has dimension $(N-K)$. However, it is possible for the parity-check matrix to contain more than $(N-K)$ rows. Therefore, the number of rows in the parity-check matrix is denoted by M, where $M \geq (N-K)$.

At the time when Gallager discovered low-density parity-check codes [5], he was also able to provide simulation results using his own suboptimal iterative decoders. However, due to the limitations of the processing speed of computers at the time, he was not able to demonstrate the near-capacity performance that LDPC codes were capable of achieving. While the performance capabilities of LDPC codes were not witnessed until much later, the introduction of Tanner graphs was arguably the next most important development in the study of LDPC codes.

In 1981, Tanner introduced a bipartite graphical representation of low-density parity-check matrices known as the Tanner graph [12]. The use of Tanner graphs as a graphical interpretation of LDPC codes lead researchers to consider the use of existing iterative message-passing algorithms to decode LDPC codes [13, 7], thus resulting in the rediscovery of LDPC codes. Since the rediscovery of LDPC codes, Tanner graphs have been used in the application of graph theory towards encoding, decoding, and code design for LDPC codes.

In order to construct a Tanner graph from a parity-check matrix, each column i in the parity-check matrix is assigned to a corresponding variable node v_i in the Tanner graph, and each row j is assigned to a corresponding check node f_j in the Tanner graph. The set of all variable nodes is V, and the set of all check nodes is F. There is an edge $e_{i,j}$ between variable node v_i and check node f_j in the Tanner graph if and only if the entry in H at the intersection of the j^{th} row and i^{th} column is a binary 1. The Tanner graph $T = (V \cup F, E)$ is thus defined by the set of variable nodes V, the set of check nodes F, and the set of edges E. The Tanner graph corresponding to the parity-check matrix $H_{(2,3)}$ (given by equation (2.1)) is shown in Figure 2.2.

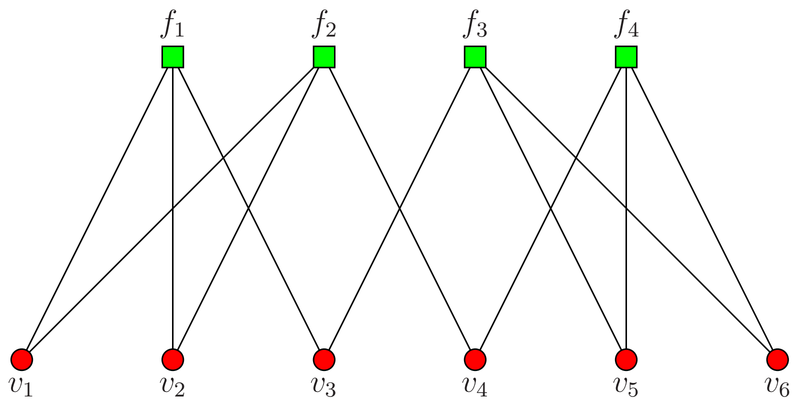


Figure 2.2: Tanner graph of a $(2, 3)$-regular LDPC code.

In this chapter, Tanner graphs and other graphical representations of low-density parity-check codes derived from Tanner graphs are used to model iterative decoders and understand their behavior. Four decoding algorithms used to decode low-density parity-check codes are examined in this chapter: maximum likelihood, linear programming, min-sum, and sum-product. After examining the decoding algorithms, some existing methods for analyzing their performance are also discussed.

2.3 Maximum Likelihood Decoding

Maximum likelihood (ML) decoding chooses the modulated codeword $\mathbf{x}$ that maximizes the probability that the channel output $\mathbf{y}$ was received given that $\mathbf{x}$ was sent. The ML decoder output $\hat{\mathbf{x}}$, is given by

$$\hat{\mathbf{x}} = \arg\max_{\mathbf{x}\in m(C)} P_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x}), \tag{2.2}$$

where $P_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x})$ is the probability that the vector $\mathbf{y}$ is the channel output, given that the modulated codeword $\mathbf{x}$ was transmitted. When using soft-decision decoding over the AWGN channel, it is necessary to use an alternative formulation of ML decoding, since $\mathbf{y}_i \in \mathbb{R}^N$ is a real number vector from a continuous probability distribution. Thus, it is not possible to deal directly with the probability distribution $P_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}|\mathbf{x})$ during decoding, becase $P_{\mathbf{Y}|\mathbf{X}}(\mathbf{y}_i|\mathbf{x}_j) = 0$ for all $\mathbf{y}_i \in \mathbb{R}^N$ and $\mathbf{x}_j \in m(C)$.

A more convenient formulation of the maximum likelihood decision criterion comes in the form of the log-likelihood ratio (LLR). An alternative realization of Equation (2.2) shows how the decoding problem can be reformulated as a minimization of the LLR cost function. When the channel is assumed to be memoryless, Equation (2.2) can be re-written as

$$\hat{\mathbf{x}} = \arg\max_{\mathbf{x}\in m(C)} \prod_{i=1}^{N} P_{Y|X}(y_i|x_i).$$

Since maximizing the probability is equivalent to maximizing the natural log of the probability,

$$\begin{aligned}\hat{\mathbf{x}} &= \arg\max_{\mathbf{x}\in m(C)} \log\left(\prod_{i=1}^{N} P_{Y|X}(y_i|x_i)\right)\\ &= \arg\max_{\mathbf{x}\in m(C)} \sum_{i=1}^{N} \log\left(P_{Y|X}(y_i|x_i)\right).\end{aligned}$$

Each x_i is equal to either $+1$ or -1, and $m^{-1}(x_i)$ is a single binary variable where either $m^{-1}(x_i) = 0$ or $m^{-1}(x_i) = 1$. Therefore, the log of the probability can be separated into two distinct components given by

$$\begin{aligned}\hat{\mathbf{x}} &= \arg\max_{\mathbf{x}\in m(C)} \sum_{i=1}^{N} \left(\log\left(P_{Y|X}(y_i|-1)\right)\left(1-m^{-1}(x_i)\right) + \log\left(P_{Y|X}(y_i|1)\right) m^{-1}(x_i)\right)\\ &= \arg\max_{\mathbf{x}\in m(C)} \sum_{i=1}^{N} \left(\log\left(P_{Y|X}(y_i|-1)\right) + \left(\log\left(P_{Y|X}(y_i|1)\right) - \log\left(P_{Y|X}(y_i|-1)\right)\right) m^{-1}(x_i)\right).\end{aligned}$$

The additive term $\log\left(P_{Y|X}(y_i|-1)\right)$ is independent of the value of x_i, and thus it does not impact the maximization. Removing $\log\left(P_{Y|X}(y_i|-1)\right)$ and combining the remaining two probabilities into the same logarithm results in

$$\begin{aligned}\hat{\mathbf{x}} &= \arg\max_{\mathbf{x}\in m(C)} \sum_{i=1}^{N} \left(\log\left(P_{Y|X}(y_i|1)\right) - \log\left(P_{Y|X}(y_i|-1)\right)\right) m^{-1}(x_i)\\ &= \arg\max_{\mathbf{x}\in m(C)} \sum_{i=1}^{N} \log\left(\frac{P_{Y|X}(y_i|1)}{P_{Y|X}(y_i|-1)}\right) m^{-1}(x_i).\end{aligned}$$

Finally, after scaling the argument by (-1) and changing the decision rule from a maximization to a minimization, ML decoding can be realized as the cost minimization

$$\begin{aligned}\hat{\mathbf{x}} &= \underset{\mathbf{x}\in m(C)}{\arg\min} \sum_{i=1}^{N} \log\left(\frac{P_{Y|X}(y_i|-1)}{P_{Y|X}(y_i|1)}\right) m^{-1}(x_i) && (2.3)\\ &= m^{-1}\left(\underset{\mathbf{c}\in C}{\arg\min} \sum_{i=1}^{N} \lambda_i c_i\right), && (2.4)\end{aligned}$$

where $\lambda_i = \frac{P_{Y|X}(y_i|-1)}{P_{Y|X}(y_i|1)}$ is the log-likelihood ratio (LLR) obtained from the channel output y_i. Thus, the codeword estimate obtained from ML decoding is the one that minimizes the cost function

$$\hat{\mathbf{c}} = \underset{\mathbf{c}\in C}{\arg\min} \sum_{i=1}^{N} \lambda_i c_i. \qquad (2.5)$$

Performing brute-force maximum-likelihood decoding requires that every codeword $\mathbf{c} \in C$ be considered by the decoder. In order to evaluate the LLR cost of each codeword, the decoder needs to perform $2^K N$ symbol-wise comparisons, where K is the dimension of the binary code C. While this is practical for codes with small dimension K, it quickly becomes computationally intractable for codes with larger dimension. For example, a relatively small code with dimension $K = 40$ requires 8.8×10^{13} symbol-wise comparisons to perform brute-force ML decoding after a single codeword transmission. Therefore, alternative algorithms are necessary to decode codes with large block length and dimension.

2.4 Linear Programming Decoding

One alternative to maximum likelihood decoding of low-density parity-check codes was introduced by Feldman in 2001, in the form of linear programming (LP) decoding [20]. The LP decoder is based on a set of linear inequalities derived from the rows of the parity-check matrix H of the code. For each row j in the parity-check matrix, let $\mathcal{E}_j = \{S \subseteq N(j) : |S| \text{ odd } \}$ be the collection of all sets S of column indices in $N(j) = \{i|h_{j,i} = 1\}$ with odd cardinality, where $h_{j,i}$ is the binary entry in H at the intersection of the j^{th} row and the i^{th} column. Also, let $\bar{S} = \{i \in N(j)|i \notin S\}$ be the set of all column indices in $N(j)$, but not in S. For each row j in the parity-check matrix H, the output $\hat{\mathbf{c}}$ of the LP decoder must satisfy the inequality

$$\sum_{i\in S} \hat{c}_i + \sum_{i\in \bar{S}} (1 - \hat{c}_i) \leq |N(j)| - 1 \tag{2.6}$$

for each $S \in \mathcal{E}_j$. The inequality given by 2.6 forces $\hat{\mathbf{c}}$ to satisfy each of the parity-check equations in H when $\hat{\mathbf{c}}$ is binary. Each bit estimate $\hat{c}_i$ must also be in the range

$$0 \leq \hat{c}_i \leq 1. \tag{2.7}$$

An important property of the set of linear inequalities derived from (2.6) and (2.7) is that all codewords in C satisfy each inequality with equality. The set of linear inequalities given in (2.6) and (2.7) are used in conjunction with a cost function to be minimized, given by

$$\sum_{i=1}^{N} \lambda_i c_i, \tag{2.8}$$

to form a complete linear program. Finally, using linear optimization techniques such as the well-known simplex algorithm [21], a solution is obtained that has minimal cost and is within the bounds defined by the set of inequalities.

If the solution to the inequalities is restricted to integer vectors $\hat{\mathbf{x}} \in \mathbb{F}_2^N$, then the optimization problem is called an integer linear program. The set of solutions to the integer linear program defined by (2.6), (2.7), and (2.8) is equal to the set of codewords. Since the cost function given in (2.8) is the same as the ML cost function given in (2.5), the output of the integer linear program is always the same as the output of ML decoding. Unfortunately, integer linear programming problems are non-deterministic polynomial-time hard, meaning that it is not possible to solve integer linear programs using an algorithm that runs in polynomial-time [20]. Therefore, allowing non-integer solutions is required to make LP decoding practical for large codes.

The complete set of vectors $\hat{\mathbf{x}} \in \mathbb{R}^N$ that satisfy (2.6) and (2.7) is known as the fundamental polytope, and the vertices of the fundamental polytope adequately represents all possible optimal solutions to the linear program. Although vertices in the fundamental polytope do not always correspond to codewords, each codeword in C corresponds to a unique vertex in the fundamental polytope.

As pointed out by Feldman [20], linear programming decoding has the maximum likelihood certificate. The *ML certificate* guarantees that if the LP decoder outputs a codeword, it is the ML codeword. This is straightforward, since the cost function minimized by the LP decoder is the same as the cost function minimized by the ML decoder. Since each codeword is a vertex of the fundamental polytope, a codeword chosen by the LP decoder must have cost less than or equal to the cost of all other codewords.

The primary disadvantage of the non-integer linear programming decoder is that the outputs are not restricted to codewords. Vertices of the fundamental polytope are referred to as *linear programming pseudocodewords*. Many times, there exist LP pseudocodewords that do not correspond to codewords, and these are referred to as *non-codeword linear programming pseudocodewords*. Since the LP decoder has the ML certificate, non-codeword LP pseudocodewords are the only type of output that results in an LP decoder error in situations where the transmitted codeword is the ML codeword.

An effort to establish a connection between the linear programming decoder and other known iterative decoders resulted in an alternative formulation of the linear programming decoder known as the graph-cover decoder [22]. This formulation shows that the outputs of the LP decoder can each be realized as a graph-cover configuration, or a *graph-cover pseudocodeword*. Furthermore, it is shown that a graph-cover pseudocodeword corresponding to the output of the LP decoder is always included in the set of optimal outputs of the graph-cover decoder. Since graph-covers are locally identical to the Tanner graph, there is an intuitive connection between the output of graph cover decoding and the outputs of local algorithms like the MS and SP decoders. Relationships between graph-cover pseudocodewords and the MS decoder outputs have recently been established [14], [23], [24]. However, there remain many open questions regarding connections between the LP decoder and existing iterative decoders.

Using linear programming to decode low-density parity-check codes is still a relatively new concept. In practice, the LP decoder offers similar bit error and word error performance when compared to the MS and SP decoders. However, the complexity of the LP decoder is much higher than the MS and SP decoders due to the need to solve large sets of linear equations. For this reason, the MS and SP decoders are considered to be more practical methods for decoding LDPC codes.

2.5 Min-Sum Decoding

The min-sum decoder is a low-complexity, sub-optimal iterative decoder that can be used to decode low-density parity-check codes. Given a particular parity-check matrix, the MS decoder operates by passing messages between the check nodes and the variable nodes along the edges of the Tanner graph of the code. Wiberg [13] shows that the MS decoder is equivalent to the ML decoder when operating on a code whose corresponding Tanner graph is a tree. When the Tanner graph of the code contains cycles, the MS decoder output is no longer guaranteed to match the ML decoder output. Though provably suboptimal, its performance is still comparable to the best known decoders of LDPC codes, and it has very low complexity when compared to other soft-decision decoders of LDPC codes.

Before introducing the min-sum decoder, some additional notation must be introduced. The set of neighbors of check node f_i in the Tanner graph is denoted $N(f_i) = \{v_j | h_{i,j} = 1\}$, and similarly the set of neighbors of variable node v_i in the Tanner graph is denoted $N(v_i) = \{f_j | h_{j,i} = 1\}$. To denote the set of neighbors of check node f_i excluding variable node v_j, the notation $N(f_i)\backslash v_j$ is used. Similarly, the set of neighbors of variable node v_j excluding check node f_i is denoted $N(v_j)\backslash f_i$. During MS decoding, messages are passed between neighboring check nodes and variable nodes along the edges of the Tanner graph. Messages from check node f_i to variable node $v_j \in N(f_i)$ are denoted by $m_{f_i \to v_j}$, and messages from variable node v_i to check node $f_j \in N(v_i)$ are denoted $m_{v_i \to f_j}$. Given a modulated codeword $\mathbf{x}$ sent by the transmitter, a channel output $\mathbf{y}$ available at the receiver, and a maximum number of iterations $\ell_{\max}$, the steps for MS decoding are given in Algorithm 2.5.1.

Algorithm 2.5.1 (Min-Sum Decoding)**.**

<u>*Step* **0***: Initialization*</u>

Set the number of iterations to $\ell = 0$*. For all messages* $m_{v_i \to f_j}$*, set*

$$m_{v_i \to f_j} = \lambda_i = \tfrac{P_{Y|X}(y_i|-1)}{P_{Y|X}(y_i|1)} = \tfrac{-2}{\sigma^2} y_i.$$

Step **1***: Variable Node to Check Node*

Set $\ell = \ell + 1$. For all messages $m_{f_i \to v_j}$, set

$$m_{f_i \to v_j} = \left(\prod_{v_k \in N(f_i) \setminus v_j} \operatorname{sgn}(m_{v_k \to f_i}) \right) \left(\min_{v_k \in N(f_i) \setminus v_j} |m_{v_k \to f_i}| \right).$$

Step **2***: Check Node to Variable Node*

For all messages $m_{v_i \to f_j}$, set

$$m_{v_i \to f_j} = \lambda_i + \sum_{k \in N(v_i) \setminus j} m_{f_k \to v_i}.$$

Step **3***: Check Stop Criteria*

For all m_{v_i}, set

$$m_{v_i} = \lambda_i + \sum_{k \in N(v_i)} m_{f_k \to v_i}.$$

For all $\hat{c}_i$, set

$$\hat{c}_i = \begin{cases} 0 & \textit{if } m_{v_i} > 0 \\ 1 & \textit{if } m_{v_i} < 0 \end{cases}.$$

If $H\hat{\mathbf{c}}^T = \mathbf{0}$ or $\ell \geq \ell_{max}$, stop decoding, otherwise return to Step 1.

In practice, the min-sum decoder does not always output a codeword. It has been shown that when the MS decoder does not output a codeword after a large number (> 200) of iterations has been performed, the output is often trapped in a repeating sequence of two or more non-codeword outputs [14].

One of the primary strengths of the min-sum decoder is the relatively small number of operations performed during each iteration. During each iteration, the messages $m_{v_i \to f_j}$ and $m_{f_j \to v_i}$ must be computed for each binary 1 in the parity-check matrix. For a (d_V, d_F)-regular LDPC code, there are $(N \times d_V) = (M \times d_F)$ binary 1's in the parity-check matrix. When the degree of the nodes and the number of iterations is fixed, the complexity of MS decoding scales linearly with the length N of the code.

In addition to only requiring a small number of operations, the min-sum decoder has two other important strengths: the simplicity of the operations and the lack of the need for channel estimation. The first strength is a result of the fact that addition and minimization are the two primary operations performed by the MS decoder. These operations can both be performed very quickly and efficiently in hardware. In regards to the second strength, it is not necessary to know the channel SNR during MS decoding since minimizations are not affected by scale factors. Thus, the channel LLR can be scaled from $-\frac{2}{\sigma^2}\mathbf{y}$ to $-\mathbf{y}$ and the output of the MS decoder will remain unchanged. This simplification removes the burden of channel estimation at the receiver.

2.6 Sum Product Decoding

Similar to the min-sum decoder, the sum-product decoder is an efficient, iterative, sub-optimal decoder that can be used to decode low-density parity-check codes. The

SP decoder, also known as the belief propagation decoder, was introduced by Pearl [8] in 1988. Whereas the MS decoder outputs the ML codeword when decoding an LDPC code whose Tanner graph is a tree, the SP decoder outputs the highest probability binary assignment to each variable node when considering all codewords on the tree. In other words, the MS decoder minimizes the probability of word error on trees and the SP decoder minimizes the probability of bit error on trees.

Given a modulated codeword $\mathbf{x}$ sent by the transmitter, a channel output $\mathbf{y}$ available at the receiver, and a maximum number of iterations $\ell_{\max}$, the steps for SP decoding are given in Algorithm 2.6.1.

Algorithm 2.6.1 (Sum-Product Decoding)**.**

Step **0***: Initialization*

Set the number of iterations to $\ell = 0$*. For all messages* $m_{v_i \to f_j}$*, set*

$$m_{v_i \to f_j} = \lambda_i = \frac{P_{Y|X}(y_i|-1)}{P_{Y|X}(y_i|1)} = \frac{-2}{\sigma^2} y_i.$$

Step **1***: Variable Node to Check Node*

Set $\ell = \ell + 1$*. For all messages* $m_{f_i \to v_j}$*, set*

$$m_{f_i \to v_j} = 2 \cdot \tanh^{-1} \left(\prod_{v_k \in N(f_i) \setminus v_j} \tanh\left(\frac{m_{v_k \to f_i}}{2}\right) \right).$$

Step **2***: Check Node to Variable Node*

For all messages $m_{v_i \to f_j}$*, set*

$$m_{v_i \to f_j} = \lambda_i + \sum_{k \in N(v_i) \setminus j} m_{f_k \to v_i}.$$

*Step **3**: Check Stop Criteria*

For all m_{v_i}, set

$$m_{v_i} = \lambda_i + \sum_{k \in N(v_i)} m_{f_k \to v_i}.$$

For all $\hat{c}_i$, set

$$\hat{c}_i = \begin{cases} 0 & \text{if } m_{v_i} > 0 \\ 1 & \text{if } m_{v_i} < 0 \end{cases}.$$

If $H\hat{\mathbf{c}}^T = \mathbf{0}$ or $\ell \geq \ell_{max}$, stop decoding, otherwise return to Step 1.

When compared to the min-sum decoder, the sum-product decoder is not as computationally efficient. One reason for this is that the SP decoder requires the computation of the hyperbolic tangent and the inverse hyperbolic tangent. Current software implementations of these two operations requires that these functions be stored in tabular form in memory. In addition to the increase in memory storage requirements, a table look-up is required each time a hyperbolic tangent or an inverse hyperbolic tangent is needed. The second reason the SP decoder is not as efficient as the MS decoder is that the decoder requires a reliable channel estimate. Unlike the MS decoder, the SP decoder does not allow for the LLR ratio to be scaled without changing the decoder output. This is because the hyperbolic tangent is sensitive to scaling whereas minimizations are not.

Neither the min-sum decoder or the sum-product decoder is guaranteed to output a codeword if the Tanner graph of the low-density parity-check code contains cycles. For this reason, non-codeword outputs are possible with both decoders. Many attempts have been made to characterize these non-codeword outputs, and explain what causes them to occur. Two existing explanations for the non-codeword outputs of iterative decoders are stopping sets and trapping sets.

2.7 Stopping Sets

The notion of stopping sets was first introduced by Forney *et al.* [25] in 2001. Two years later, a formal definition of stopping sets was given by Changyan *et al.* [26].They demonstrated that the bit and word error probabilities of iteratively decoded LDPC codes on the binary erasure channel (BEC) can be determined exactly from the stopping sets of the parity-check matrix.

Definition 2.7.1 (Stopping Sets [26])**.** *A stopping set $\mathcal{S}$ is a subset of the set of variable nodes V, such that any check node connected to a variable node contained in $\mathcal{S}$ is connected to at least two variable nodes in $\mathcal{S}$.*

A small example of a stopping set is given in Figure 2.3. Consider the subset $\mathcal{S} = \{v_1, v_2, v_3, v_4\}$ of the set of variable nodes V. There are five check nodes $\{f_1, f_2, f_3, f_4, f_5\}$ connected to the set $\mathcal{S}$, and each of them is connected to $\mathcal{S}$ at least two times. Note that only f_2 is connected to the set $\mathcal{S}$ an odd number of times. If each of the check nodes is connected to $\mathcal{S}$ an even number of times, $\mathcal{S}$ corresponds to a codeword support set where each bit in $\mathcal{S}$ can be flipped without changing the overall parity of any of the check nodes.

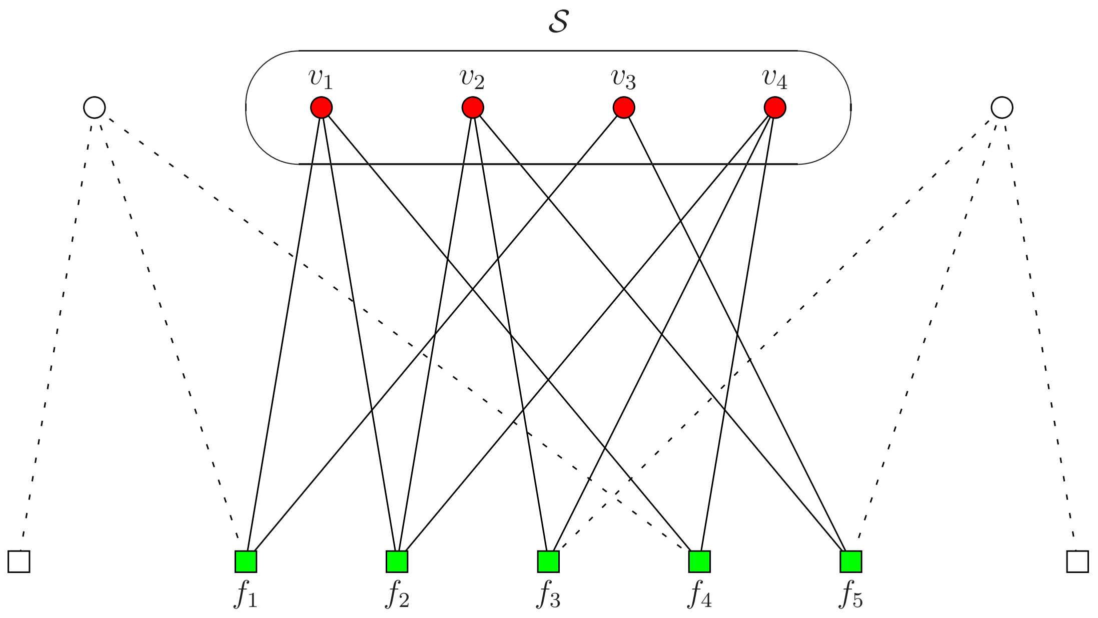


Figure 2.3: Example of a stopping set in the Tanner graph of an LDPC code.

The intuition behind stopping sets begins with an understanding of iterative message-passing decoders. Information given to a specific variable node from a neighboring check node is derived from all other variable nodes connected to that check node. Consider two variable nodes $v_i, v_j \in N(f_k)$, where both variable nodes contain an erasure. In this case, each of the sets $N(f_k)\backslash v_i$ and $N(f_k)\backslash v_j$ contains at least one erasure, thus making it impossible for the check node f_k to determine the parity of either set. For this reason, none of the check nodes connected to a stopping set is capable of resolving erasures if each variable node contained in the stopping set begins with an erasure from the channel.

Work relating linear programming pseudocodewords to stopping sets for the binary erasure channel [25], and both the binary symmetric channel (BSC) and the additive white Gaussian noise channel [27], has revealed a relationship between linear programming pseudocodewords and the size of stopping sets. Although stopping sets have a strong relationship with LP pseudocodewords, the performance of neither the MS decoder or the SP decoder on the BSC and AWGN channels can be predicted using stopping sets alone.

2.8 Trapping Sets

Trapping sets were first introduced by MacKay and Postol [28] to provide an explanation for the weaknesses of algebraically constructed low-density parity-check codes. They define trapping sets as follows.

Definition 2.8.1 (Trapping Sets [28]). *Consider a length N code with parity-check matrix H, and let $\mathcal{T} \subseteq \{1, \ldots, N\}$ be a set containing $|\mathcal{T}| = t$ coordinates. Consider a binary vector $\mathbf{y}$ with 1's in the coordinates of $\mathcal{T}$ and 0's elsewhere. If the syndrome $\mathbf{s} = H\mathbf{y}$ has Hamming weight w_t, the set $\mathcal{T}$ is referred to as a (t, w_t) trapping set.*

Consider the trapping set shown in Figure 2.4, where the set $\mathcal{T} = \{1, 2, 3, 4\}$ corresponds with a set of variable nodes $\{v_1, v_2, v_3, v_4\}$ in the Tanner graph of the parity-check matrix H. There are four variable nodes in the set, so $t = 4$, and if all variable nodes are set to a binary 1, only check nodes f_2 and f_3 are connected to an odd number of binary 1's, so the syndrome $\mathbf{s}$ has Hamming weight equal to 2. Therefore, this set of variable nodes defines a $(4, 2)$ trapping set.

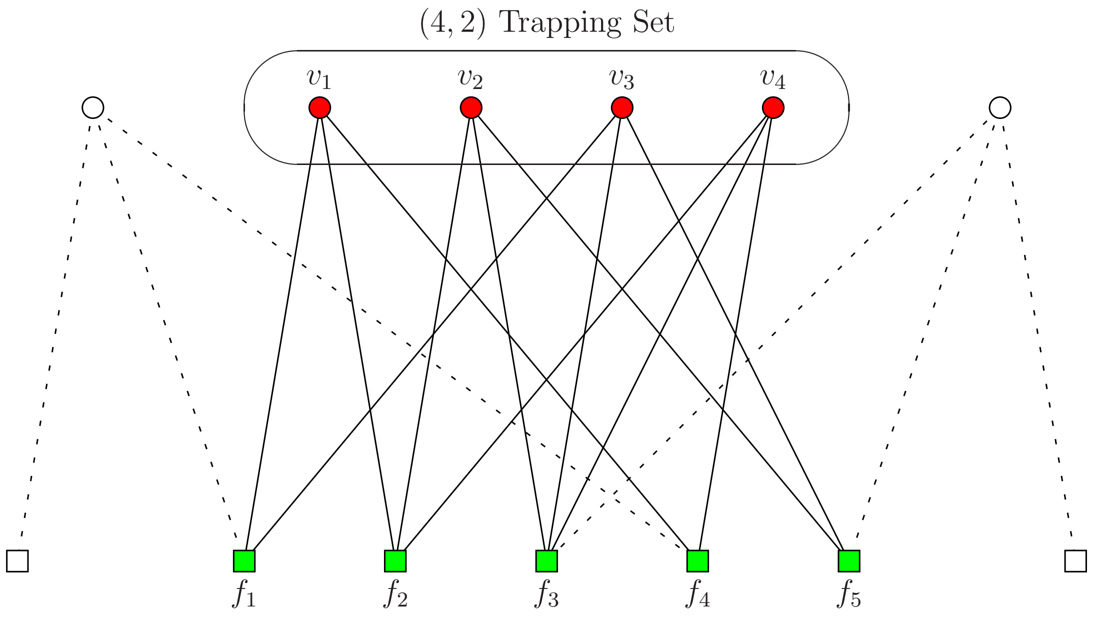


Figure 2.4: Example of a $(4, 2)$ trapping set in the Tanner graph of an LDPC code.

It is important to note that any set of variable nodes can be considered a trapping set defined by some set of parameters, and the significance of trapping sets varies greatly depending on the parameters (t, w_t). Consider transmission of the all-zeros codeword over a BSC with crossover probability $p_{\text{BSC}} = P(y_i = 1|x_i = 0) = P(y_i =$

$0|x_i = 1) < 0.5$. The channel output $\mathbf{y}$ is more likely if the Hamming weight of $\mathbf{y}$ is $t < \frac{N}{2}$. Therefore, using reasonable channel assumptions, the vector $\mathbf{y}$ is expected to contain more binary 0's than binary 1's. Additionally, if the syndrome has a low Hamming weight w_t, then only a small number of parity-checks are capable of detecting the error. Trapping sets with small t and small w_t are thought to be particularly problematic to iterative decoders.

In [29], trapping sets are examined for different decoders on the binary erasure channel, binary symmetric channel, and the additive white Gaussian noise channel. Whereas stopping sets can be used to precisely determine the probability of error on the BEC, trapping sets appear to cause errors on the AWGN channel. Richardson [29] uses the parameters and multiplicity of various problematic trapping sets to estimate the error floor of LDPC codes at bit error rates where simulations are not feasible. Unfortunately, the somewhat vague definition of problematic trapping sets makes it difficult to use them for performance analysis.

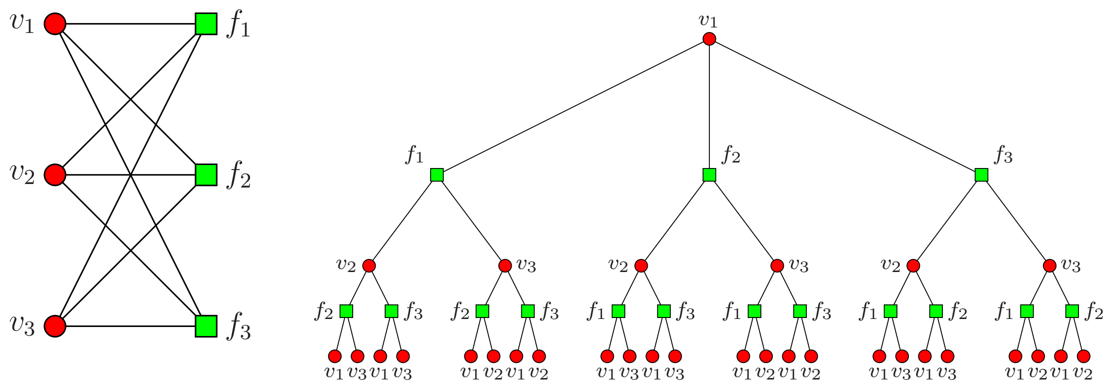
2.9 Computation Trees and Deviations

In his 1996 dissertation, Wiberg [13] presented groundbreaking analytical results with respect to iterative decoding of low-density parity-check codes. He provided extensive analysis of both the MS and SP decoders by introducing a model of iterative decoding known as the computation tree. Wiberg showed that the MS decoder minimizes the probability of word error when decoding a code whose Tanner graph is a tree, while for the same type of code the SP decoder minimizes the probability of bit error.

In addition to introducing computation trees, Wiberg also introduced the concept of deviations. Wiberg proved that deviations on the computation tree with negative cost are required in order for errors to occur during MS and SP decoding. Because of the importance of computation trees and deviations in understanding finite tree-based decoding, they are examined in detail in this section.

Consider a low-density parity-check code represented by a Tanner graph $T = (V \cup F, E)$. A computation tree rooted at variable node v_i after ℓ iterations is denoted $R_{v_i}^{(\ell)}$. In order to construct a computation tree from the Tanner graph, a variable node v_i is placed at the top level (root) of a descending tree. To construct the next level in the tree directly below v_i, each of v_i's neighbors in $N(v_i)$ is added to this level and connected to v_i. This process continues level-by-level, where nodes in the previous level are used to determine nodes on next level, while maintaining that each node in the computation tree has the same set of neighbors as its corresponding node in the Tanner graph. For example, if variable node v_j on the last completed level is connected to check node f_k on the level above it, then all check nodes in $N(v_j)\backslash f_k$ must appear on the next level and be connected to v_j, thereby ensuring that v_j is connected to exactly one copy of each check node in $N(v_j)$.

Figure 2.5 gives an example of a small Tanner graph, and its corresponding computation tree rooted at v_1 after two iterations. Nodes at the bottom level of the computation tree are referred to as *leaf nodes*. Notice that the leaf nodes are the only nodes in the computation tree that are not connected to a copy of each of their neighbors in the original Tanner graph.



(a) Tanner graph (b) Computation tree

Figure 2.5: Computation tree of a simple repetition code after $\ell = 2$ iterations.

Computation trees are precise models for analyzing the performance and behavior of min-sum and sum-product decoding for a finite set of iterations. Each of these decoders can be perfectly modeled after ℓ iterations by constructing N different computation trees that contain $2\ell + 1$ levels of nodes including the root node. The N computation trees are each rooted at a different variable node from the original Tanner graph. Then, for every variable node v_i in each computation tree, the LLR cost γ_i is assigned to that variable node. At this point, MS or SP decoding operations can be performed from the leaf nodes up to the root node. The final cost at each of the root nodes determines the binary estimate of the transmitted codeword computed by the decoder. Because the MS and SP decoders are optimal on Tanner graphs that are trees, the MS and SP decoders are optimal on each of the computation trees derived from the Tanner graph. MS chooses the least cost valid configuration on the tree, where a *valid configuration* refers to any assignment of binary numbers to the variable nodes such that each check node is adjacent to an even number of variable nodes assigned to a binary 1. The SP decoder, on the other hand, chooses the value at the root node that has the highest probability over all valid configurations.

Although the computation tree model is precise, after a small number of iterations it becomes impractical to analyze the performance of specific codes by considering all valid configurations on the computation tree. The number of valid configurations on the computation tree can be computed by treating the computation tree as a Tanner graph. In order to define a Tanner graph given the computation tree, treat all check nodes and variable nodes in the computation tree separately. For example, if multiple copies of variable node v_1 are distributed throughout the computation tree, each copy is treated as a distinct variable node. After regarding each variable node in the computation tree as distinct, one can show that each check node on the computation tree corresponds to a linearly independent parity-check equation. If there are $\left|R_{v_i}^{(\ell)}(V)\right|$ variable nodes and $\left|R_{v_i}^{(\ell)}(F)\right|$ check nodes on a computation tree rooted at variable node v_i after ℓ iterations, then there are a total of $2^{\left|R_{v_i}^{(\ell)}(V)\right|-\left|R_{v_i}^{(\ell)}(F)\right|}$ valid configurations on the tree. On a (d_V, d_F)-regular LDPC code, the number of variable nodes after ℓ iterations is given by

$$\left|R_{v_i}^{(\ell)}(V)\right| = 1 + \sum_{i=0}^{\ell-1} d_V(d_F-1)\left(((d_V-1)(d_F-1))^i\right) \tag{2.9}$$

and the number of check nodes is given by

$$\left|R_{v_i}^{(\ell)}(F)\right| = \sum_{i=0}^{\ell-1} d_V\left(((d_V-1)(d_F-1))^i\right).$$

To illustrate the growth rate in the number of valid configurations on the computation tree, consider an LDPC code where each variable node has degree $d_V = 3$ and each check node has degree $d_F = 6$. These commonly used code parameters define what is known as a $(3, 6)$-regular LDPC code. Table 2.1 shows the number of variable nodes given by

$$\left|R_{v_i}^{(\ell)}(V)\right| = 1 + \sum_{i=0}^{\ell-1} 15(10^i),$$

the number of checks nodes given by

$$\left|R_{v_i}^{(\ell)}(F)\right| = \sum_{i=0}^{\ell-1} 3 \cdot 10^i,$$

and the corresponding number of valid configurations on the computation tree after 1, 2, and 3 iterations. Note that the growth rate is not affected by the block length of the code.

Iterations	Variable Nodes	Check Nodes	Configurations
1	16	3	8192
2	166	33	$\approx 10^{40}$
3	1666	333	$\approx 10^{401}$

Table 2.1: The number of nodes and valid configurations on the computation tree of a $(3, 6)$-regular LDPC code.

Table 2.1 illustrates the computational complexity associated with considering each valid configuration on the computation tree. In light of this, Wiberg [13] derived a simplified bound on the performance of MS decoding operating on a particular computation tree. In order to obtain this bound, Wiberg introduced the concept of deviations on the computation tree. A *deviation* is any set of variable nodes on the computation tree satisfying the following three conditions.

1. Each check in the computation tree is adjacent to either two or zero variable nodes in the deviation set.

2. A deviation set must contain the root node of the computation tree.

3. No proper and non-empty subset of variable nodes in the deviation form a valid configuration on the computation tree.

Figure 2.6 shows an example of a deviation on the computation tree given in Figure

2.5(b). The larger blue variable nodes are contained in the deviation, whereas the smaller red nodes are not.

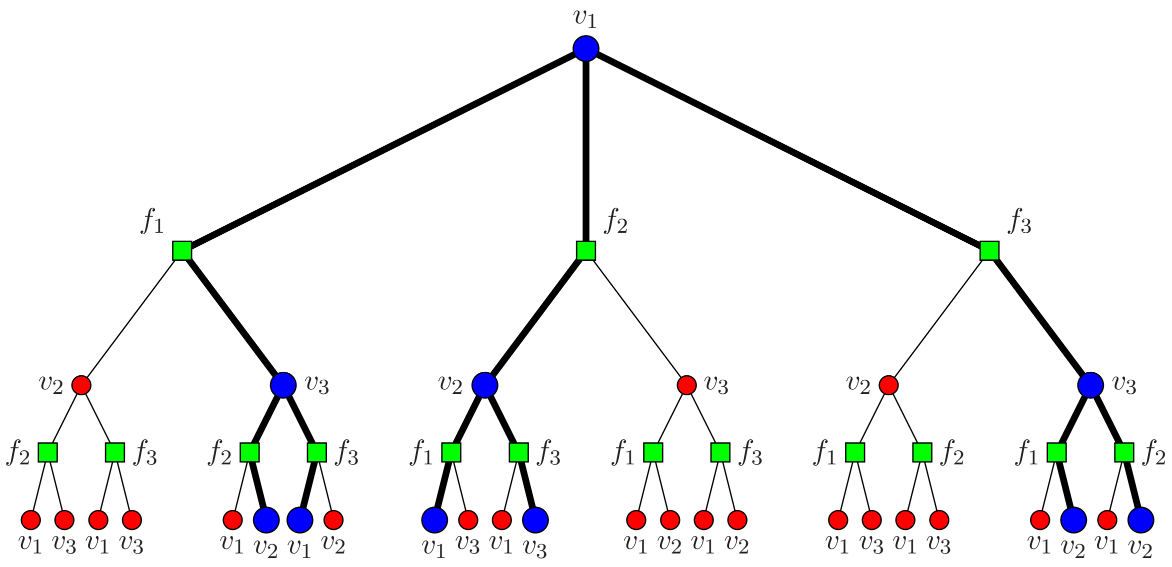

v_1
f_1
f_2
f_3
v_2
v_3
v_2
v_3
v_2
v_3
f_2 f_3 f_2 f_3 f_1 f_3 f_1 f_3 f_1 f_2 f_1 f_2
v_1 v_3 v_1 v_3 v_1 v_2 v_1 v_2 v_1 v_3 v_1 v_3 v_1 v_2 v_1 v_2 v_1 v_3 v_1 v_3 v_1 v_2 v_1 v_2

Figure 2.6: Example of a deviation on the computation tree.

Wiberg uses the set of deviations on the computation tree to derive an upper bound on the performance of the min-sum decoder. It is necessary, but not sufficient, for at least one deviation δ in the set of all deviations Δ to have negative cost in order for an error to occur at the root node. This condition assumes that the all-zeros codeword was transmitted over the channel. The cost of the deviation, denoted by $G(\delta)$, can be found by summing the LLR cost of each of the nodes in the support of the deviation. The cost of a deviation is given by

$$G(\delta) = \sum_{v_i \in \delta} \gamma_i,$$

where copies of $v_i \in \delta$ are counted as many times as they appear in the deviation. A necessary, but not sufficient, condition for an error to occur on the computation tree rooted at variable node v_i is

$$\min_{\delta \in \Delta} G(\delta) < 0.$$

Using this condition, a bound can be derived on the probability that the minimum-cost configuration on the computation tree contains a binary 1 at the root node. This bound is

$$\begin{aligned} P(v_i = 1) &\leq P(\min_{\delta \in \Delta} G(\delta) < 0) \\ &\leq \bigcup_{\delta \in \Delta} P(G(\delta) < 0) \end{aligned}$$

which can be further loosened to

$$P(v_i = 1) \leq \sum_{\delta \in \Delta} P(G(\delta) < 0) \tag{2.10}$$

by using the union bound.

Wiberg [13] shows that the bound given by (2.10) can be used to predict the performance of min-sum decoding of infinite-length codes after a specific number of iterations. Wiberg begins by assuming that the computation trees have no repeated nodes. This assumption simplifies the weight enumerators of the deviations for regular LDPC codes. Wiberg also shows that (2.10) can be used to bound MS decoder performance when there are multiple copies of each variable node in the tree. Thus, in theory, Wiberg's deviation bound can be used to bound the performance of MS decoding of finite codes. The following proposition shows that the number of deviations grows exponentially with d_V, thus making it computationally intractable to enumerate the deviations even after a small number of iterations.

Proposition 2.9.1. *Let $R_{v_i}^{(\ell)}$ be the computation tree of a (d_V, d_F)-regular LDPC code, rooted at variable node v_i after ℓ iterations. Then, the number of deviations that exist on $R_{v_i}^{(\ell)}$ is*

$$(d_F - 1)^{\sum_{i=1}^{\ell} d_V (d_V - 1)^{i-1}}. \tag{2.11}$$

Proof. By the definition of a deviation, we must assign the root node v_i to a binary 1. Each of the d_V check nodes immediately below v_i must assign exactly one of the their $(d_F - 1)$ child variable nodes to a binary 1. Thus, there are a total of $(d_F - 1)^{d_V}$ deviations after one iteration. In addition, there are exactly d_V leaf nodes in the support of each deviation after one iteration.

Each of the previous d_V leaf nodes gets connected to $(d_V - 1)$ check nodes after two iterations. Each of these check nodes assigns one of their $(d_F - 1)$ child variable nodes to a binary 1. Therefore, for each deviation after one iteration there are $(d_F - 1)^{d_V(d_V - 1)}$ different deviations after two iterations. This brings the total number of deviations to $(d_F-1)^{d_V}(d_F-1)^{d_V(d_V-1)} = (d_F-1)^{(d_V)^2}$ after two iterations. The total number of leaf nodes in the support of the deviation after two iterations is $d_V(d_V-1)$.

Following this pattern, the $d_V(d_V-1)$ variable nodes in support of the deviation after two iterations branchs out to $d_V(d_V-1)^2$ check nodes. There are $(d_F-1)^{d_V(d_V-1)^2}$ ways of assigning the leaf nodes to the support of the previous deviation. This brings the total number of deviations to $(d_F-1)^{(d_V)^2}(d_F-1)^{d_V(d_V-1)^2} = (d_F-1)^{(d_V)^3-(d_V)^2+d_V}$ after three iterations.

After ℓ iterations, the $d_V(d_V-1)^{\ell-2}$ old leaf nodes in the support of the deviation branch out to $d_V(d_V-1)^{\ell-1}$ new leaf nodes in the support of the deviation. There are $(d_F-1)^{d_V(d_V-1)^{\ell-1}}$ ways of assigning the support to the previous deviation, and the total number of deviations after ℓ iterations is

$$\prod_{i=1}^{\ell}(d_F-1)^{d_V(d_V-1)^{i-1}} = (d_F-1)^{\sum_{i=1}^{\ell} d_V(d_V-1)^{i-1}}$$

$\square$

The number of deviations on the computation tree of a $(3,6)$-regular low-density parity-check code is given in Table 2.2 for iterations 1 through 5. Even after only a small number of iterations, it becomes impractical to enumerate each of the deviations in order to compute the upper bound on the probability of bit error of the root variable node of the computation tree.

Using computation trees, Wiberg provided a precise model of the behavior of

Iterations	# of Deviations
1	125
2	1,953,125
3	4.7684×10^{14}
4	2.8422×10^{31}
5	1.0097×10^{65}

Table 2.2: Number of deviations at iterations 1-5 for a $(3, 6)$-regular LDPC code.

the min-sum and sum-product decoders. Unfortunately, the size of the computation trees and the number of configurations on them grows too large for practical analysis. Deviations provide a simplified approach to the analysis of computation trees, but the number of deviations also grows exponentially with the number of iterations. Thus, it is still of interest to find a tractable method for computing bounds on the performance of iterative decoders.

2.10 Density Evolution

Density evolution takes a different approach than stopping sets and trapping sets to analyzing the performance of iterative decoders of low-density parity-check codes. Stopping sets and trapping sets attempt to explain the error mechanisms of iterative decoders with finite codes, but neither approach leads to analytical results for the SP or MS decoder operating on LDPC codes transmitted over the BIAWGN channel. As an alternative to stopping sets and trapping sets, computation trees and the notion of deviations lead to proven upper bounds on the performance of MS decoding. However, as shown in Section 2.9, analysis of computation trees quickly becomes computationally intractable after only a small number of iterations. Thus, tractable analysis of MS and SP decoding of finite-length LDPC codes remains an open problem.

Density evolution is a method for analyzing the performance capabilities of infinite-length low-density parity-check codes with iterative message-passing decoders [17, 18]. Shannon's capacity bound [1] was also computed under the assumption of infinite-length codes. However, unlike Shannon's capacity bound, density evolution takes into account the structure of the parity-check matrix of the code and the corresponding iterative decoder.

In Section 2.10.1, background is provided regarding the probability distribution of messages passed between nodes during sum-product decoding. Section 2.10.2 shows how density evolution can be used to track the changing probability distributions with each iteration of iterative decoding.

2.10.1 Message Distributions

Assuming a binary-input, additive white Gaussian noise channel, each received value y_i, $1 \leq i \leq N$, can be described by its probability density function (PDF) or the associated cumulative distribution function (CDF). If $x_i = +1$, the PDF of y_i can be realized as a Gaussian distribution centered at $+1$ with a standard deviation of σ derived from the channel SNR. The LLR λ_i can be similarly modeled by its PDF.

Using sum-product decoding, messages $m_{f\to v}$ get added together during each iteration. It is possible to obtain the probability distribution of the sum of these messages, since the PDF of the sum of independent random variables is the convolution of their individual PDFs. The distribution of all messages $m_{v\to f}$ at the ℓ^{th} iteration is given by $\mathcal{P}_\ell$, and the sum of two messages has a PDF given by $\mathcal{P}_\ell \star \mathcal{P}_\ell = \mathcal{P}_\ell^{\star 2}$, where $\star$ denotes convolution. The convolution makes it possible to obtain $m_{v\to f}$ from previous values of $m_{f\to v}$.

Now, consider the map $\gamma : [-\infty, +\infty] \to \mathrm{GF}(2) \times [0, +\infty]$ defined by

$$\gamma := (\gamma_1(x), \gamma_2(x)) := \left(\operatorname{sgn}(x), -\ln\left(\tanh\left|\frac{x}{2}\right|\right)\right),$$

where

$$\operatorname{sgn}(x) = \begin{cases} 0 & \text{if } x \geq 0 \\ 1 & \text{if } x < 0 \end{cases}.$$

The messages in sum-product decoding are first passed from variable node to check node and then back from check node to variable node. The update equation in Step

1 of the SP decoder given by Algorithm 2.6.1 can be rewritten using γ as

$$m_{f_i \to v_i} = \gamma^{-1} \left(\sum_{v_j \in N(f_i) \setminus \{v_i\}} \gamma \left(m_{v_j \to f_i} \right) \right).$$

Note that the product has been replaced by a sum, since the γ function maps to the natural logarithm while preserving sign. A formulation of the PDF of $\gamma\left(m_{v \to f}\right)$ would allow for the PDF of $m_{f \to v}$ to be computed using convolution. In order to obtain the PDF of $\gamma\left(m_{v \to f}\right)$, the Γ function is used. The details of the Γ and Γ^{-1} functions can be found in [18].

2.10.2 Evolution of Message Distributions

Assume sum-product decoding is used with a low-density parity-check code with infinite block length, and all the cycles in the Tanner graph are also infinite in length. These assumptions allow all messages passed within the Tanner graph to be independent. Furthermore, by characterizing LDPC codes based upon a probabilistic model of their check and variable node degrees, it is possible to greatly simplify the analysis of the evolution of the message PDFs. This probabilistic model requires that the Tanner graph of the LDPC code satisfies two conditions. First, it has a fraction of edges connected to variable nodes of degree i given by η_i. Secondly, it has a fraction of edges connected to check nodes of degree i given by ρ_i. The pair of vectors $\boldsymbol{\eta}$ and $\boldsymbol{\rho}$ are called the *degree profiles* of the LDPC code. Note that the degree profiles define an ensemble of LDPC codes instead of a particular LDPC code.

The distribution of $m_{f\to v}$ at iteration ℓ is denoted by $\mathcal{Q}_\ell$, and the distribution of $m_{v\to f}$ is $\mathcal{P}_\ell$. The update equations for the distributions are

$$\mathcal{Q}_\ell = \Gamma^{-1}\left(\sum_{i\geq 2}\rho_i(\Gamma(\mathcal{P}_{\ell-1}))^{\star(i-1)}\right), \tag{2.12}$$

and

$$\mathcal{P}_\ell = \mathcal{P}_0 \star \sum_{i\geq 2}\eta_i(\mathcal{Q}_\ell)^{\star(i-1)} \tag{2.13}$$

which are consistent with the update equations from the SP decoder. Density evolution begins by computing $\mathcal{Q}_\ell$ using equation (2.12) after initializing $\mathcal{P}_{\ell-1} = \mathcal{P}_0$.

Density evolution can be used to obtain the exact probability of bit error for sum-product decoding of an infinite-length low-density parity-check code with infinite-length cycles. To obtain the exact probability of a bit error at a given SNR, equations (2.12) and (2.13) must be repeated in sequence an infinite number of times to obtain $\mathcal{P}_\infty$. However, a good estimate of the probability of a bit error can be obtained by picking a sufficiently large number of iterations ℓ.

Density evolution is commonly used as a design tool for finding degree profiles that result in low-density parity-check codes with the potential for low bit error rates at signal-to-noise ratios close to capacity. Unfortunately, since density evolution assumes an infinite block length, it is only capable of providing an estimate of the performance of finite LDPC codes with finite-length cycles. However, as will be shown in Section 3.6.2, density evolution can be used to obtain the exact probability of bit error of finite tree-based decoding methods under certain conditions.

2.11 Problem Statement and Related Work

This chapter has examined several methods already in the literature for decoding low-density parity-check codes and analyzing the performance of the decoders. Since their rediscovery, LDPC codes have attracted great interest in the field of coding theory. Along with Turbo codes, they are one of the few classes of codes that is capable of demonstrating near-capacity performance. The SP and MS decoders have emerged as the two most popular iterative decoders used for LDPC codes. However, the lack of proven analytical results on the decoding performance of the SP and MS decoders makes it difficult to accurately predict their performance. While computer simulations are capable of providing an estimate of the decoder performance for a limited range of SNRs, analytical results are required to predict the performance over the entire range of SNRs.

Several current methods for analyzing the behavior and error mechanisms of iterative decoding have been discussed in this chapter. These methods include stopping sets, trapping sets, density evolution, and computation tree analysis. Unfortunately, none of these methods is capable of providing an upper bound on the decoding performance of MS or SP decoding over the BIAWGN channel. The current state of research in the area of low-density parity-check codes indicates that there is a need for a decoding technique that allows for computationally tractable performance analysis.

Rather than attempting to derive new upper bounds for the min-sum and sum-product decoders, this dissertation presents several new finite tree-based construction methods for which upper bounds, and in one special case exact analytical results, can be computed. Finite tree construction methods and the finite tree decoder are presented and examined in this dissertation as an alternative to the MS and SP decoders, with the following two primary goals:

1. Achieve bit error rates comparable to those of current iterative decoders, such as the MS and SP decoders.

2. Maintain a simple enough graphical representation of the decoder to allow for practical analysis and bounds.

A decoding method capable of achieving these two goals could provide great benefits to current and future applications of LDPC codes. The expected probability of bit error of LDPC codes could be accurately predicted beyond the reach of computer simulations. In systems where bit errors cannot be tolerated, near-error-free performance with a probability of bit error below 10^{-20} could be guaranteed for specific ranges of SNR. In addition, analytical results could allow for new methods of LDPC code construction, where codes are designed to perform well at high SNR.

This dissertation presents alternative methods for decoding low-density parity-check codes. In particular, all the methods presented in Chapter 3 operate on finite trees based upon the Tanner graph. Two existing methods for decoding LDPC codes, namely the tree-pruning decoder and the self-correcting min-sum decoder, can also be modeled as decoders that operate on finite trees based upon the Tanner graph.

In [30], tree-pruning decoding is presented for codes represented by graphs, e.g., low-density parity-check codes. The tree-pruning decoder is motivated by the results presented by Weitz [31], proving that there exists a way to prune a computation tree such that the probability of error at the root node approaches that of maximum a posteriori decoding on the Tanner graph, when using the SP decoder on the tree. Trees generated using this method are referred to as self-avoiding walk (SAW) trees. Unfortunately, the size of SAW trees grows exponentially with the size of the original Tanner graph T. For this reason, in [30] the authors are forced to apply further truncation to the SAW trees to make the tree-pruning decoder computationally tractable for LDPC codes. Unfortunately, this truncation eliminates the guaranteed performance advantages of using the un-truncated SAW tree. Due to rate of increase in the computational complexity of the resulting trees, the results of [30] are limited to simulations using very small block-length LDPC codes.

In 2008, Savin introduced the self-correcting min-sum (SCMS) decoder [32]. The SCMS decoder utilizes a simple modification to the MS decoder in order to improve its performance. In simulations, the SCMS decoder demonstrates performance within 0.1 dB of the SP decoder at bit error rates below 10^{-6}. The SCMS decoder modifies regular MS decoding by setting the message $m_{v_i \to f_j}$ to zero if it changes signs on consecutive iterations. By setting this variable-to-check message to zero, all check node messages $m_{f_j \to v_k}$ where $v_k \neq v_i$ will be set to zero at the next iteration.

With the computation tree interpretation of decoding in mind, setting a check node message to zero is equivalent to eliminating the check node and all of its descendants from the computation tree. From this point of view, SCMS is essentially performing regular MS operations on a modified computation tree with the result being a greatly reduced probability of error at the root node. Unfortunately, the structure of the modified computation tree depends on the specific channel output vector, thus making it difficult to characterize the modified computation trees under the assumption of random channel noise.

Both tree-pruning decoding and self-correcting min-sum decoding demonstrate some of the advantages of decoding low-density parity-check codes using finite, modified computation trees. Unfortunately, the trees produced with SCMS grow at nearly the same rate as computation trees, and they can only be constructed and analyzed after a very small number of iterations. Additionally, the tree-pruning decoder produces finite trees that only allow for construction and analysis using small block length codes where $N < 50$.

Chapter 3 introduces and examines several different finite tree-based decoding methods, and shows how to obtain bounds and analytical results on the performance of these new decoders. Chapter 4 presents a new method for constructing LDPC codes designed to perform well with finite tree-based decoders. When compared to an existing method of code construction, the new method produces LDPC codes that not only have improved cycle structure, but also have higher minimum distance properties as measured with a new lower bound on the minimum distance of the code.

Chapter 3

Finite Tree-Based Decoding

Current methods for decoding low-density parity-check codes are capable of achieving exceptionally low bit error rates in computer simulations. However, the behavior of the MS and SP decoders remains largely a mystery at error rates below those achievable in simulations. Applications like hard disc drive storage, where near error-free performance is desired, demand a decoding solution with guaranteed performance at error rates below the simulation capabilities of computers. Thus, the lack of performance bounds for MS and SP decoding prevents their usage in applications requiring exceptionally low bit error rate.

Computation trees are a precise model for the behavior of the min-sum and sum-product decoders. Unfortunately, as demonstrated in Section 2.9, analysis using computation trees quickly becomes intractable after only a small number of iterations.

In addition to being difficult to analyze, the differences between computation trees and the Tanner graph of an LDPC code are such that the optimal configurations on the two graphs do not always coincide. Example 3.0.1 demonstrates a situation where the ML decoder output on the Tanner graph does not equal the MS/ML decoder output on the computation tree.

Example 3.0.1. *Consider an LDPC code with the Tanner graph shown in Figure*

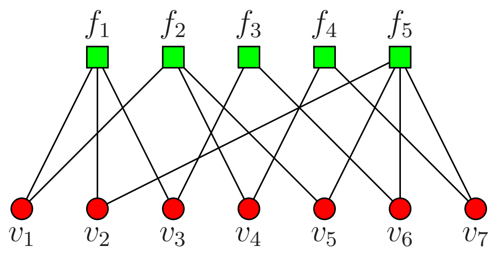


Figure 3.1: Tanner graph of a length $N = 7$, dimension $K = 3$ LDPC code.

3.1. After $\ell = 2$ iterations, the computation tree rooted at variable node v_1 is shown by the graph in Figure 3.2. Assume that the channel output at $SNR = 5.0$ dB results in a LLR cost vector given by

$$\boldsymbol{\lambda} = (+0.3, -0.1, +1.3, +1.0, -0.1, +1.1, +1.4).$$

The ML decoder considers all codewords $\mathbf{c} \in C$ and outputs the codeword that minimizes

$$\hat{\mathbf{c}} = \arg\min_{\mathbf{c} \in C} \sum_{i=1}^{N} \lambda_i c_i.$$

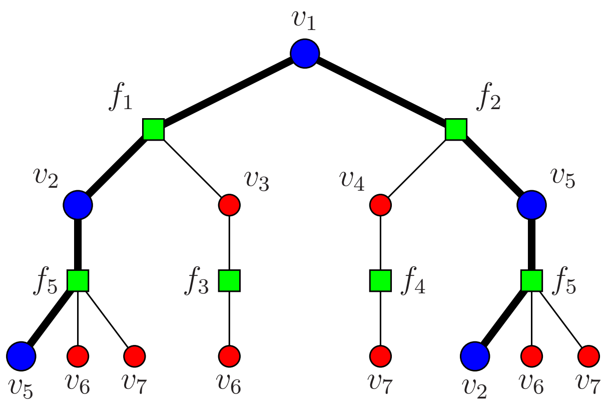


Figure 3.2: Computation tree rooted at variable node v_1 after $\ell = 2$ iterations.

In this example, the ML decoder outputs the codeword $(0, 0, 0, 0, 0, 0, 0)$*. However, the minimum-cost configuration on the computation tree rooted at* v_1 *does not coincide with maximum-likelihood valid configuration on the Tanner graph. Instead, the minimum-cost configuration shown in Figure 3.2 corresponds to the codeword* $(1, 1, 0, 0, 1, 0, 0)$*, and the MS decoder outputs a binary 1 at the root node* v_1*. In Figure 3.2, the large blue variable nodes are assigned to a binary 1, and the small red variable nodes are assigned a binary 0. Although there are a total of seven computation trees needed to compute the output of the MS decoder after* $\ell = 2$ *iterations, the computation tree rooted at variable node* v_1 *sufficiently demonstrates the difference between the ML decoder output and optimal valid configurations on computation trees.*

Example 3.0.1 demonstrates that it is possible for min-sum decoding, after a fixed number of iterations, to have a different output than the maximum-likelihood decoder.

Therefore, in order to model the behavior of the MS decoder, valid configurations on the computation tree need to be considered. In Section 2.9, it was shown that the performance of MS decoding can be bounded by considering only the weights of the deviations on the computation tree. In fact, if MS decoding performance is to approximate that of ML decoding, the weight distribution of the deviations on the computation tree should resemble the weight distribution of the codewords in the code.

A histogram of codeword weights is given in Figure 3.3(a) for a length $N = 20$, dimension $K = 10$, $(2, 4)$-regular LDPC code. From these weights, the performance of ML decoding could be upper bounded using the union bound. Figures 3.3 and 3.4 also show the weight distribution of the deviations on the computation tree after iterations 1 through 6. Notice that as the number of iterations increases, the weight distribution of deviations on the computation tree begins to resemble that of the codeword weights. The evolution of the deviation weight histogram suggests that the weight distribution of the deviations might eventually resemble that of the codewords as the number of iterations increases.

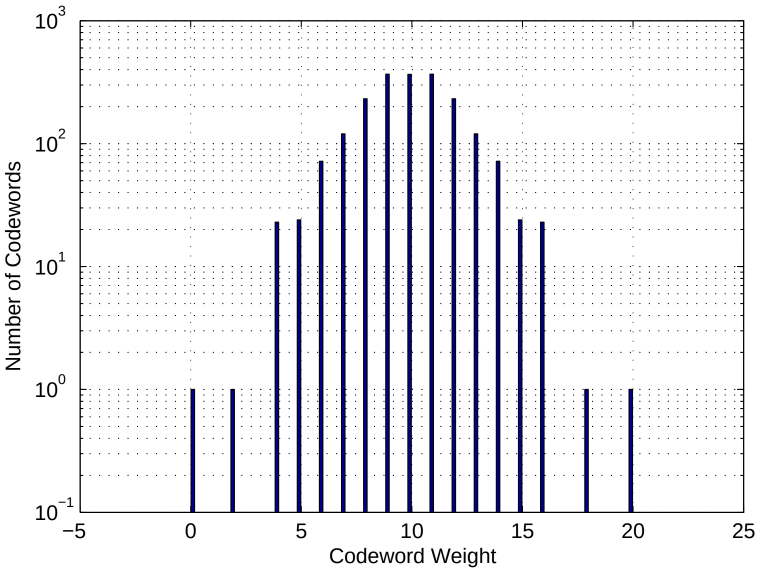
Figure 3.5 shows the codeword weights and deviation weights for a length $N = 20$, dimension $K = 10$, $(3, 6)$-regular LDPC code after iterations 1 and 2. Unfortunately, it is not computationally feasible to enumerate the deviations for more than two iterations for the $(3, 6)$-regular LDPC code, since after three iterations there are

4.7684×10^{14} deviations on the computation tree. Therefore, though the histogram of the weight distribution of deviations is an effective tool for visualizing the distance properties of computation trees, it is not practical beyond a few iterations.

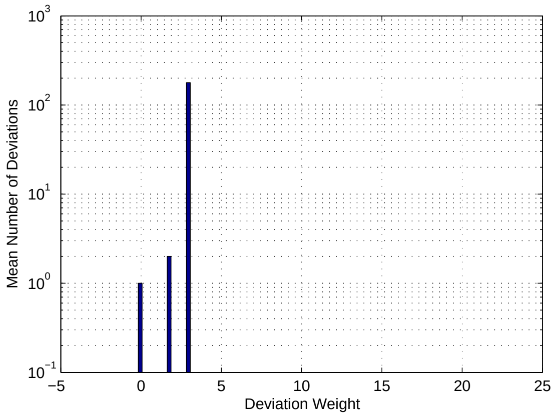
In Figures 3.3 through Figure 3.5, the evolution of deviation weights was shown on the computation tree modeling the min-sum decoder. The computation trees grow systematically with each additional iteration, and with no regard to the structure of the valid configurations that are allowed on the tree. Therefore, there is no guarantee that the valid configurations on the computation tree will correspond with codewords after any number of iterations. It is also uncertain whether or not the valid configurations will be problematic to the MS decoder. For example, if a valid configuration has weight lower than the minimum weight codeword, the MS decoder will perform worse than the ML decoder as SNR grows large.

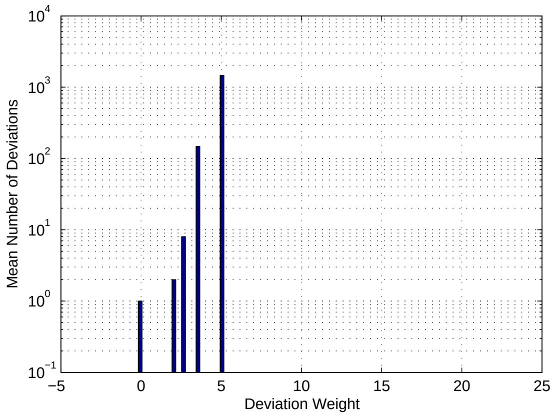
This dissertation presents finite tree-based decoders as an alternative method for decoding LDPC codes. The goal of finite tree-based decoders is to provide a method for decoding LDPC codes that performs comparably to existing iterative decoders, while having predicable performance at very low error rates in the form of computationally tractable analytical expressions or bounds. Unlike the MS decoder, finite tree-based decoding aims to create finite trees while maintaining control over the valid configurations that are possible on the finite trees.

In this chapter, several novel finite tree-based decoders are introduced and their



(a) Codewords

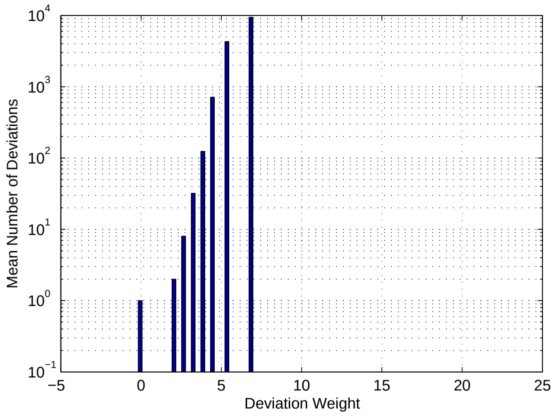




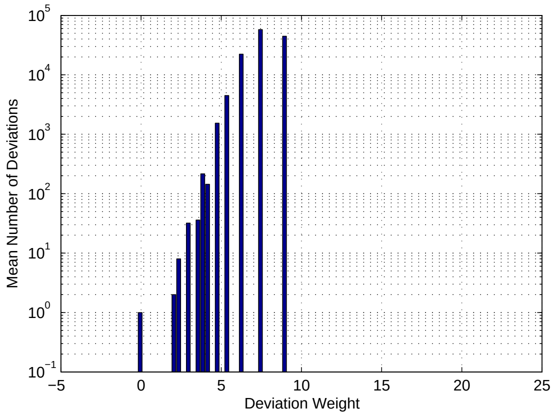
(b) 1 Iteration

(c) 2 Iterations

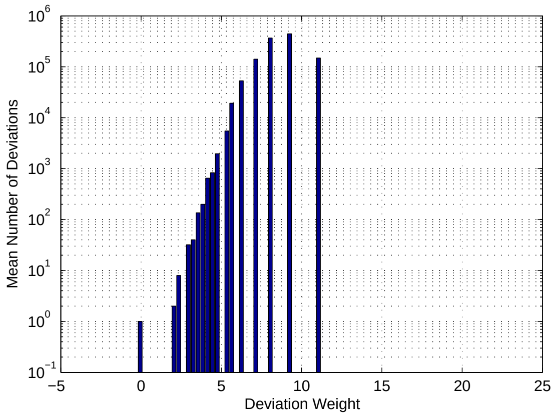
Figure 3.3: Codeword and deviation weight histograms for a length $N = 20$, dimension $K = 10$, $(2, 4)$-regular LDPC code for iterations 1 and 2.



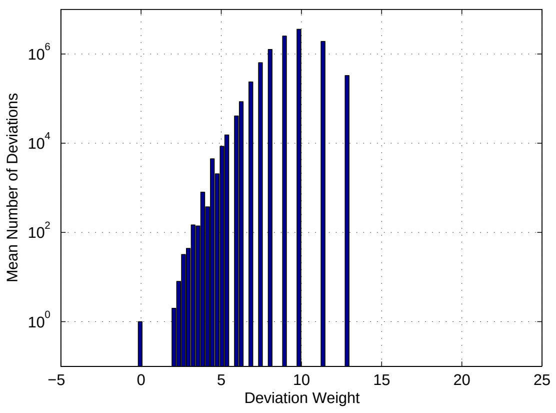
(a) 3 Iterations



(b) 4 Iterations



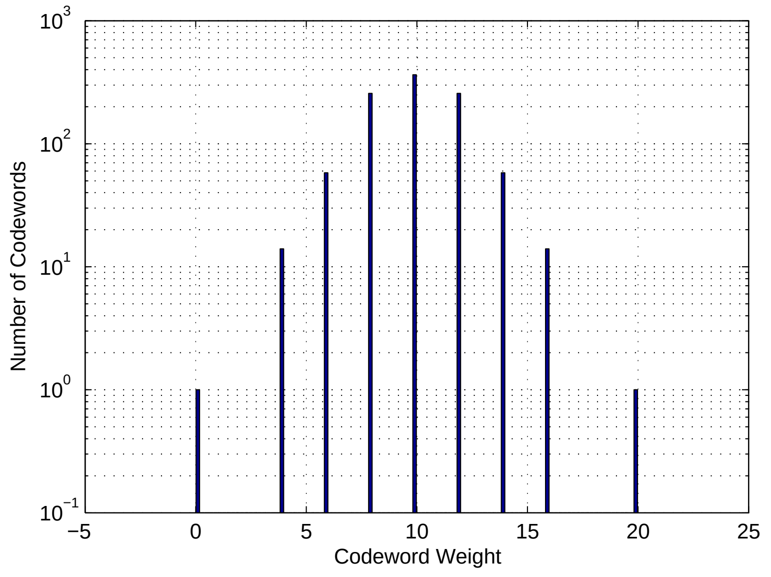
(c) 5 Iterations



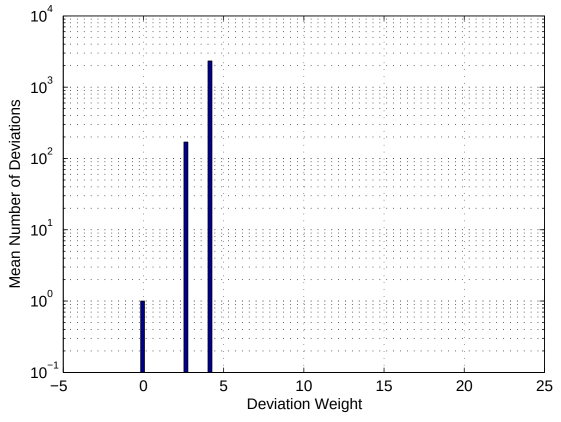
(d) 6 Iterations

Figure 3.4: Deviation weight histograms for a length $N = 20$, dimension $K = 10$, $(2, 4)$-regular LDPC code for iterations 3 through 6.

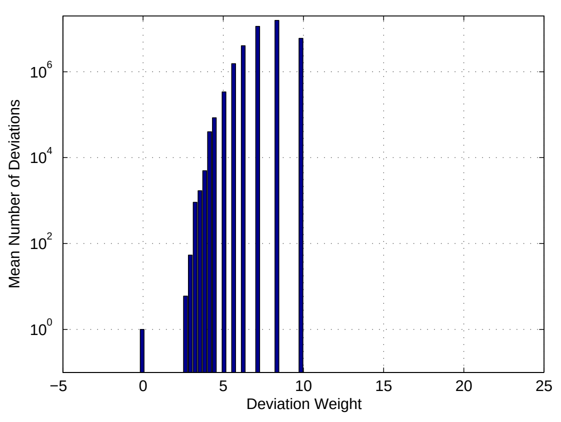
performance analysis examined. In Section 3.1, motivation for finite tree-based decoding is given in the form of iteratively constructed trees. The iterative algorithm for building trees node-by-node shows how including parts of the tree can either increase



(a) Codewords



(b) 1 Iteration



(c) 2 Iterations

Figure 3.5: Codeword and deviation weight histograms for a length $N = 20$, dimension $K = 10$, $(3, 6)$-regular LDPC code or iterations 1 and 2.

or decrease the probability of error at the root node. It is also demonstrated that different tree structures are desirable at different SNRs. In Section 3.2, independent trees are introduced as a primitive method for generating finite trees with desirable analytical properties. Two more sophisticated methods for constructing finite trees, known as extrinsic tree construction and deviation path-forcing tree construction, are then introduced in Section 3.3 and Section 3.5, respectively. Section 3.6 concludes the chapter with a discussion of techniques that can be used to upper bound the probability of bit error for finite tree decoders. Finally, techniques for deriving exact performance curves using density evolution are examined and applied to the independent trees introduced in Section 3.2.

3.1 Iterative Tree Construction

This section presents a method for iteratively constructing trees based on the Tanner graph of a low-density parity-check code. The goal of iterative tree building is to construct finite trees with a low probability of error at the root node of the tree. The first step in the iterative tree building process is to fix the channel SNR. With the channel SNR fixed, the procedure for iterative tree building is given by Algorithm 3.1.1.

Algorithm 3.1.1 (Iterative Tree Construction)**.**

```
for $i = 1, \ldots, N$
 – Root the tree at variable node $v_i$.
 – Set the level of the tree to $\ell = 0$.
 – Set the current probability of error at the root node to $P_{b,v_i} = 1.0$.
    while one or more variable nodes exist in level $\ell$ of the tree.
        for each variable node $v_k$ in the tree at level $\ell$.
            if check node $f_j \in N(v_k)$ is not the parent node of $v_k$.
             –Connect $f_j$ to $v_k$, and connect variable nodes $v_l \in N(f_j) \backslash v_k$
              to $f_j$ at level $\ell + 1$.
             –Perform simulations to obtain a new estimate of the probability
              of error at the root node.
              if the new probability is less than $P_{b,v_i}$.
               –Keep $f_j$ and its corresponding child variable nodes.
               –Set $P_{b,v_i}$ equal to the new probability of error at the root node.
              else
               –Eliminate $f_j$ and its corresponding variable nodes.
        end
        $\ell = \ell + 1$.
    end
end
```

In order to obtain an estimate of the probability of error at the root node, decoding must be performed on the tree. The first step to decoding on the tree is to assign LLR costs to each of the variable nodes in the iteratively constructed tree. Then, using the same variable node and check node operations as the MS decoder [13], the decoding operations are performed from the leaf nodes of the tree up to the root node. The decoding process is given in Algorithm 3.1.2.

Algorithm 3.1.2 (Finite Tree Decoding)**.** *The message m_{v_i} at each leaf node in the tree is initialized to the LLR cost λ_i assigned to that node. Let $C(f)$ denote the child nodes of check node f, and let $C(v)$ be the child nodes of variable node v. At each check node, the message*

$$m_{f_i} = \left(\prod_{v_k \in C(f_i)} \operatorname{sgn}(m_{v_k}) \right) \left(\min_{v_k \in C(f_i)} |m_{v_k}| \right)$$

is computed from the costs of its child node(s). Next, the variable node message

$$m_{v_i} = \lambda_i + \sum_{f_k \in C(v_i)} m_{f_k}$$

is computed from the check node message(s) of its children. These two steps are performed at every check node and variable node from the lowest nodes in the tree up toward the root node. This process continues until the message $m_{v_{\text{root}}}$ has been computed at the root node of the tree. The message obtained at the root node represents the difference in cost between the minimum-cost configuration assigning a binary 0 at the root node and the minimum-cost configuration assigning a binary 1 at the root *node. This final cost is then quantized using*

$$\hat{c}_{\text{root}} = \begin{cases} 0 & \textit{if } m_{v_{\text{root}}} > 0 \\ 1 & \textit{if } m_{v_{\text{root}}} \leq 0 \end{cases}$$

to determine the decoded binary value of the root node. The set of binary root node values assigned to each of the trees is the final codeword estimate given by the decoder.

Note that the finite tree decoder is the decoder used with all methods of finite tree construction presented in this dissertation. Also note that the finite tree decoder uses the MS cost criteria instead of the SP cost criteria. There are two primary reasons for choosing the MS cost criteria. The first reason is that it negates the need for channel estimation at the decoder, thus reducing the complexity of the decoder. The second reason is related to the connection between the MS decoder and deviations, since the MS decoder calculates the cost difference between choosing and not choosing the minimum-cost deviation on the computation tree. In addition, the MS decoder is guaranteed to choose the minimum-cost valid configuration on the computation tree, whereas the SP decoder may choose a configuration that does not satisfy all the checks on the computation tree, and thus is not a valid configuration. Now, consider the following example illustrating the iterative construction of a tree based upon the Tanner graph of a LDPC code.

Example 3.1.3. *Iterative tree construction is used to build a tree derived from the Tanner graph given in Figure 3.6, and the tree chosen for this example is the one* *rooted at variable node v_2. This example begins with iterative tree construction that utilizes finite tree decoder simulation results where the channel SNR is fixed at 0.0 dB. The effect of changing the SNR is shown later in this example.*

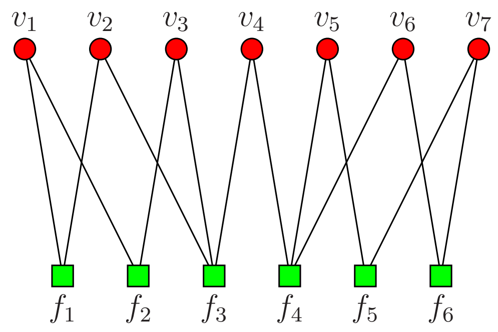


Figure 3.6: Tanner graph of a length $N = 7$, dimension $K = 2$ LDPC code.

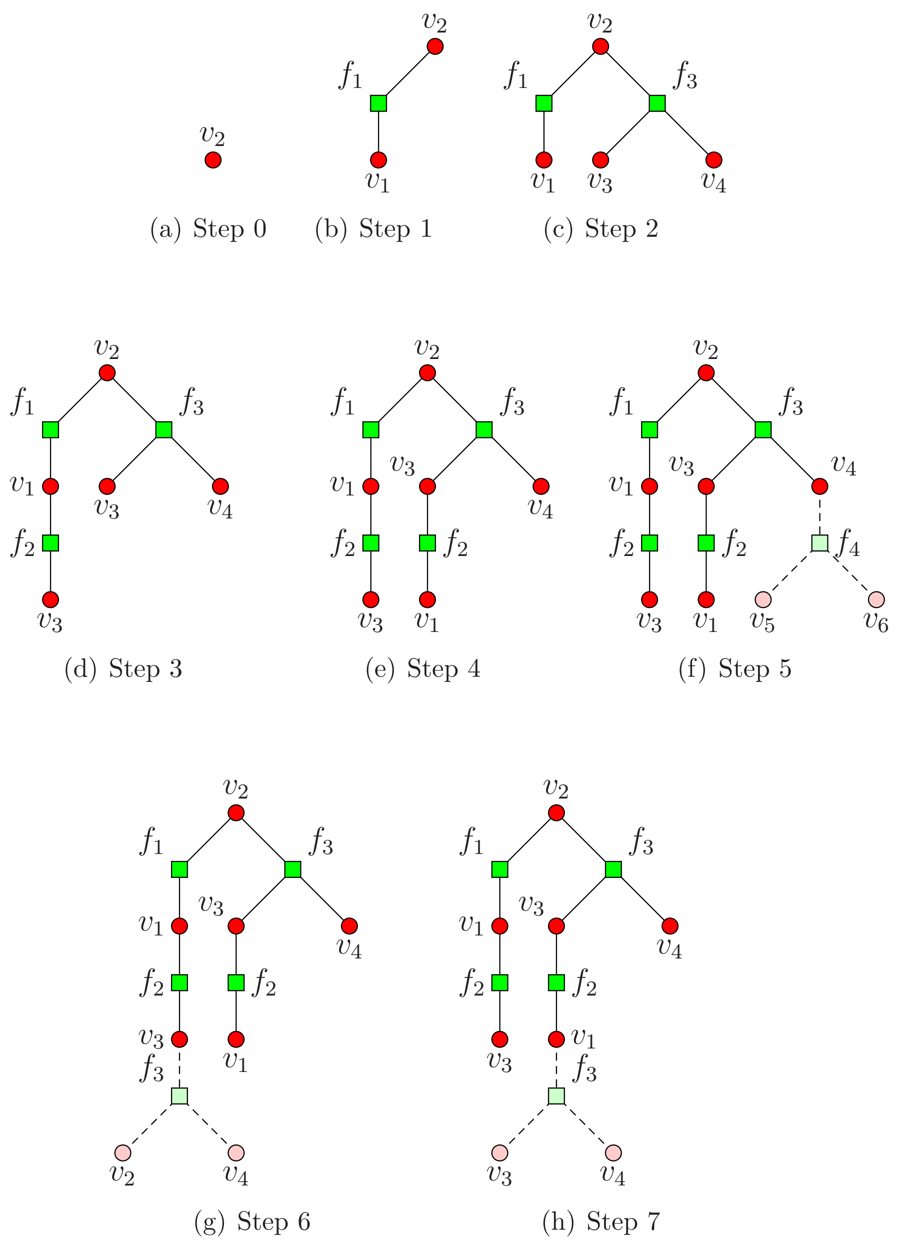
Figure 3.7 illustrates each step of the tree building process for this example. To begin the construction, the tree is rooted at variable node v_2. This is referred to as Step 0 in the tree building process. The probability of error at the root node is simulated, given that the all-zeros codeword was sent. The probability of error at the root node for each step is given in Table 3.1.

Once the root node has been set, and a minimum probability of error has been established, the next step in the iterative tree building process is to add check node f_1 below the root node. Check node f_1 is added to the tree since $f_1 \in N(v_2)$. In addition to adding f_1 to the tree, all other variable nodes in $N(f_1)\backslash v_2$ are added to the tree below f_1. In this example, only variable node $v_1 \in N(f_1)\backslash v_2$ is added to the tree *and connected to f_1. With the new check node added to the tree along with its child variable node, the probability of error is once again simulated. Here, the probability of error at the root node is reduced from* $P_{b,v_2} = 0.4496917980$ *to* $P_{b,v_2} = 0.1428583711$, *so the additional check node and corresponding variable nodes are retained in the tree.*

In Steps 2 through 4 additions to the tree are retained since P_{b,v_2} *decreases at each of these steps. In Steps 5 through 7, the additions to the tree are not retained since they would result in an increase in* P_{b,v_2}*. Finally, since no additions were made at the lowest level in the tree, the iterative tree construction is compete.*

Step	P_{b,v_2}	Minimum P_{b,v_2}	Added
0	0.4496917980	0.4496917980	Yes
1	0.1428583711	0.1428583711	Yes
2	0.1272847652	0.1272847652	Yes
3	0.1047076583	0.1047076583	Yes
4	0.1033010930	0.1033010930	Yes
5	0.1044096351	0.1033010930	No
6	0.1088432968	0.1033010930	No
7	0.1109668538	0.1033010930	No

Table 3.1: Probability of bit error P_{b,v_2} at the root node for steps 1 through 7 during the construction of the tree rooted at v_2 for an SNR = 0.0 dB.



(a) Step 0 (b) Step 1 (c) Step 2

(d) Step 3 (e) Step 4 (f) Step 5

(g) Step 6 (h) Step 7

Figure 3.7: Iterative construction of a tree rooted at v_2 for SNR = 0.0 dB.

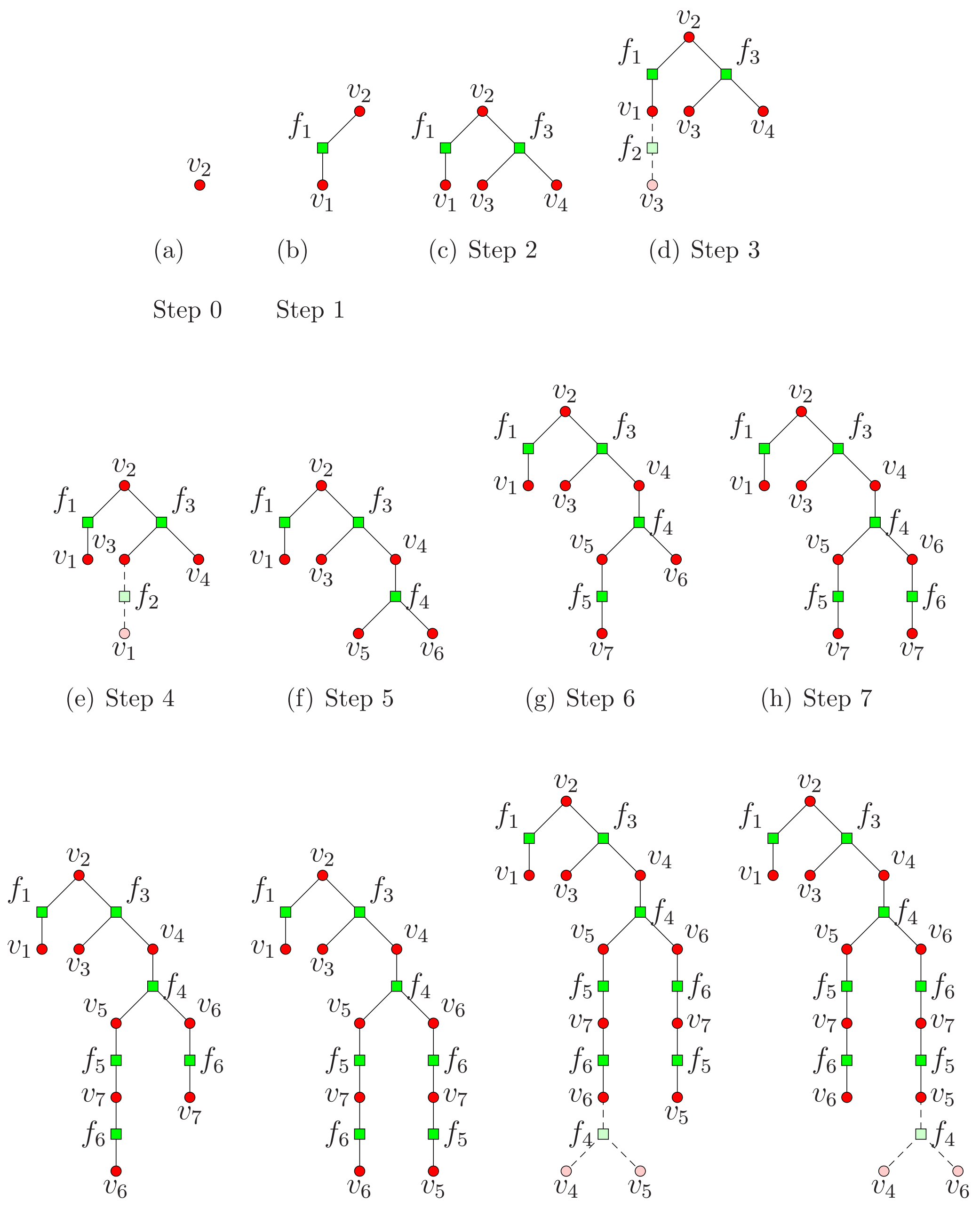
Recall that the previous tree building was performed at SNR = 0.0 dB. A new tree building process is performed at SNR = 9.0 dB, and the results for the tree rooted at

v_2 are given in Figure 3.8 and Table 3.2. Note that the trees developed at low and high SNR are significantly different from each other.

Step	P_{b,v_2}	Minimum P_{b,v_2}	Added
0	0.0331305835	0.0331305835	Yes
1	0.0012862213	0.0012862213	Yes
2	0.0002069710	0.0002069710	Yes
3	0.0002382885	0.0002069710	No
4	0.0003455956	0.0002069710	No
5	0.0001275513	0.0001275513	Yes
6	0.0001227266	0.0001227266	Yes
7	0.0001134427	0.0001134427	Yes
8	0.0001130205	0.0001130205	Yes
9	0.0001115520	0.0001115520	Yes
10	0.0001123801	0.0001115520	No
11	0.0001119743	0.0001115520	No

Table 3.2: Probability of bit error P_{b,v_2} at the root node for steps 1-10 during the construction of the tree rooted at v_2 for an SNR = 9.0 dB.

Using Algorithm 3.1.1, trees are constructed iteratively based upon the simulated probability of error at each step. This leads to the question: "How accurate does the


v_2
f_1 v_2 v_1
f_1 v_2 f_3 v_1 v_3 v_4
v_2 f_1 f_3 v_1 v_3 v_4 f_2 v_3
v_2 f_1 f_3 v_1 v_3 v_4 f_2 v_1
v_2 f_1 f_3 v_1 v_3 v_4 f_4 v_5 v_6
v_2 f_1 f_3 v_1 v_3 v_4 f_4 v_5 v_6 f_5 v_7
v_2 f_1 f_3 v_1 v_3 v_4 f_4 v_5 v_6 f_5 f_6 v_7 v_7
v_2 f_1 f_3 v_1 v_3 v_4 f_4 v_5 v_6 f_5 f_6 v_7 v_7 f_6 v_6
v_2 f_1 f_3 v_1 v_3 v_4 f_4 v_5 v_6 f_5 f_6 v_7 v_7 f_6 f_5 v_6 v_5
v_2 f_1 f_3 v_1 v_3 v_4 f_4 v_5 v_6 f_5 f_6 v_7 v_7 f_6 f_5 v_6 v_5 f_4 v_4 v_5
v_2 f_1 f_3 v_1 v_3 v_4 f_4 v_5 v_6 f_5 f_6 v_7 v_7 f_6 f_5 v_6 v_5 f_4 v_4 v_6

(a) Step 0 (b) Step 1 (c) Step 2 (d) Step 3

(e) Step 4 (f) Step 5 (g) Step 6 (h) Step 7

(i) Step 8 (j) Step 9 (k) Step 10 (l) Step 10

Figure 3.8: Iterative construction of a tree rooted at v_2 for an SNR = 9.0 dB.

probability of error have to be to build the finite trees?" If the probability of error is not accurate and an error is made in the early stages of the tree building, this can cause the tree to be drastically different than what was intended. Example 3.1.4 illustrates how inaccuracies in the probability of a bit error can affect the tree.

Example 3.1.4. *Consider a length $N = 40$ $(3,6)$-regular LDPC code. Algorithm 3.1.1 is used to construct a tree rooted at variable node v_1 for SNR = 5.0 dB. Table 3.3 shows the minimum number of bit errors that occured during simulations while building the tree at each step, along with the minimum P_{b,v_1} encountered during the building process and also the value of P_{b,v_1} estimated during a simulation of the final tree where 20,000 bit errors were counted.*

Example 3.1.4 indicates that a large number of bit errors need to be simulated in order to obtain a high level of confidence that the check node and its child variable node(s) should or should not be retained at each step. Instead of fixing the number of bit errors counted in simulations, it is possible to use the statistics of the simulation to adaptively change the required number of bit errors. An adaptive simulation method using confidence intervals is now examined in detail.

It is possible to use statistics to compute the confidence that the actual probability of error has increased or decreased after a specific number of bit errors has been counted in simulations. In order to establish statistical confidence, bit error statistics are recorded while the simulations take place.

Bit Errors Simulated	Minimum P_{b,v_1} during building	Final P_{b,v_1}	Variable Nodes
25	0.0033552544	0.0056401738	66
50	0.0025213051	0.0040533668	116
100	0.0019811396	0.0028328805	151
250	0.0027112910	0.0032020134	111
500	0.0031489914	0.0036082678	96
1000	0.0027901551	0.0030695035	131
2000	0.0025202089	0.0026876337	191
4000	0.0023306615	0.0024206347	246
8000	0.0021044342	0.0021926407	351

Table 3.3: Final probability of bit error P_{b,v_1} at the root node for SNR = 5.0 dB after construction of the tree rooted at v_1 using different bit error thresholds for estimating P_{b,v_1}.

While performing simulations, an estimate $\overline{X}$ of the expected mean bit error probability μ_X is obtained. The law of large numbers states that the estimated mean will approach the expected mean as the sample size approaches infinity. However, since it is impossible to obtain an infinite number of samples, a confidence level will be established based upon the number of samples available at any given time. If there are n samples available from the random variable X, and it is assumed that the random variable X has a normal distribution, only two statistics are needed to establish confidence that $\overline{X}$ is within a certain range of the true mean μ_X. The first statistic is the sample mean given by

$$\overline{X} = \frac{1}{n}\sum_{i=1}^{n} x_i,$$

where n is the number of samples available. The other necessary statistic is the sample variance, given by

$$S_X^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{X})^2.$$

Since it is impossible to obtain the true variance of the random variable, it is common to treat the sample variance as the true variance after a sufficient number of samples has been obtained. In this application, after at least 100 samples have been obtained, the sample variance is assumed to be equal to the true variance. With the assumption that $\sigma^2 \approx S_X^2$, a level of confidence can be established about the true mean μ_X given a set of samples from the random variable X. If a set of n samples is obtained, it is of interest to know the probability that $\overline{X}$ is within a distance τ from the true mean μ_X. This probability is given by

$$P(\mu_X - \tau < \overline{X} < \mu_X + \tau) = \int_{\mu_X-\tau}^{\mu_X+\tau} \frac{1}{\sqrt{2\pi S_X^2/n}} e^{\frac{-x^2}{2S_X^2/n}} dx,$$

where the variance S_X^2 is inversely scaled by the number of samples n to obtain the variance $\frac{S_X^2}{n}$ of the mean $\overline{X}$. This expression can be simplified by moving the mean to zero and recognizing the symmetry of the Gaussian distribution. The simplified expression is given by

$$P(\mu_X - \tau < \overline{X} < \mu_X + \tau) = 2 \int_0^{\tau} \frac{1}{\sqrt{2\pi S_X^2/n}} e^{\frac{-x^2}{2S_X^2/n}} dx. \tag{3.1}$$

Using the expression given by equation (3.1), it is possible to implement a new criteria for iterative tree building where an adaptive number of simulations is performed before each new addition to the tree is included or discarded. For example, requiring a 95% certainty that any new addition to the tree would either increase or decrease the existing probability of error at the root node requires that the new mean probability of error is at least 2.24 standard deviations above or below the existing probability of error.

Bit Errors Simulated	Minimum P_{b,v_1} during building	Final P_{b,v_1}	Variable Nodes
8000	0.0021044342	0.0021926407	351
Adaptive (99.73%)	0.0014652347	0.0014688394	397

Table 3.4: Final probability of bit error P_{b,v_1} at the root node for SNR = 5.0 dB after construction of the tree rooted at v_1 using 8000 bit errors and the adaptive criteria for estimating P_{b,v_1}.

Table 3.4 compares the results of iterative tree building using 8000 bit errors and the new adaptive criteria. During construction using the adaptive criteria it is required that there is a 99.73% probability that the addition of a check node and its child variable nodes decreases the existing probability of error at the root node. Statistical confidence was limited to 99.73% due to computational complexity associated with requiring higher levels of confidence, as higher levels of confidence always require an increase in the number of samples. Using the adaptive criteria with 99.73% confidence, the final probability of error at the root node of the tree is significantly improved when using the adaptive criteria.

It is worthwhile to compare the results of finite tree decoding on iterative trees to those of min-sum decoding of v_1 from the same $N = 40$ $(3, 6)$-regular LDPC code. Results of MS decoding are given in Table 3.5. The MS decoder can be modeled by the computation tree and the number of variable nodes in the computation tree is shown for each iteration. At four iterations, the computation tree of the MS decoder uses far more variable nodes than the tree built using the adaptive criteria. Even though the MS decoder uses far more variable nodes, the probability of bit error of the MS decoder and finite tree decoding on iterative trees are very close. The excess of nodes in the computation tree without decreased error rates indicates that, in this example, there is a weakness in the graphical structure of the computation trees. It takes more than 10 iterations for the MS decoder to show significant performance advantages over finite tree decoding on iterative trees using the adaptive criteria. These results show that, with a fixed number of variable nodes, finite trees can be designed with improved bit error rates when compared to computation trees.

Iterations	P_{b,v_1}	Variable Nodes
1	0.008333263919	16
2	0.003102959367	166
3	0.001689280500	1666
4	0.001411635778	16666
10	0.001327487058	$\approx 1.666 \times 10^{10}$
20	0.000760487514	$\approx 1.666 \times 10^{20}$
40	0.000504333235	$\approx 1.666 \times 10^{40}$
80	0.000340443716	$\approx 1.666 \times 10^{80}$
160	0.000273044687	$\approx 1.666 \times 10^{160}$

Table 3.5: Minimum P_{b,v_1}.

Experimental results with iterative tree building indicate a weakness in standard computation trees at finite iterations. With this weakness in mind, the primary goal of finite tree-based decoding is to develop a systematic method for constructing the finite trees. The trees must be small enough in size to allow for performance analysis, while maintaining performance comparable to the MS decoder.

3.2 Independent Tree Construction

Computation trees are used to create an exact model of the behavior of the min-sum and sum-product decoders. However, they are difficult to analyze when they contain multiple copies of the same variable node. With multiple copies of the same variable node in the computation tree, it is possible for a valid configuration on a computation tree to also include multiple copies of the same variable node. When this happens, the probability that the decoder is more likely to choose that particular valid configuration over a configuration that assigns all-zeros to the variable nodes is no longer simply a function of the number of variable nodes in the valid configuration. Instead, each variable node in the valid configuration must be considered along with the number of times it appears within the configuration. As shown in Section 2.9, this becomes computationally intractable as the computation tree grows large.

As shown in the following construction method, independent trees can be designed to include at most one copy of each variable node. Trees with this property assign independent channel information to each variable node during decoding. As shown later in this chapter, the fact that only one copy of each variable node is included in the independent tree also makes it possible to compute a precise probability of error at the root node.

The first step of independent tree (IT) decoding for a given code is to build an independent tree rooted at each variable node in the code. The process for building the independent trees is given in Algorithm 3.2.1.

Algorithm 3.2.1 (Independent Tree Construction)**.**

```
for $i = 1, \ldots, N$
    Root the tree at variable node $v_i$, and set the level of the tree to $\ell = 0$.
    while one or more variable nodes exist in level $\ell$ of the independent tree.
        for each variable node $v_k$ in the independent tree at level $\ell$
            (a) If check node $f_j \in N(v_k)$ is not the parent node of $v_k$,
                connect $f_j$ to $v_k$, and connect all other variable nodes
                $v_l \in N(f_j)\backslash v_k$ to $f_j$ at level $\ell + 1$.
            (b) If no other copy of variable node $v_l \in N(f_j)\backslash v_k$ already exists in
                the independent tree, keep $f_j$ and its corresponding child
                variable nodes. Otherwise, eliminate $f_j$ and its corresponding
                variable nodes from the independent tree.
        end
        $\ell = \ell + 1$.
    end
end
```

Independent trees are constructed prior to decoding and their size and structure do not change during decoding. Decoding is performed by first assigning LLR costs to each of the variable nodes in the independent tree. Then operations are performed from the leaf nodes to the root node using the finite tree decoder given in Algorithm 3.1.2. The following example demonstrates the guaranteed error correcting capability of the independent tree decoder and compares it to the probability of error using maximum-likelihood decoding.

Example 3.2.2. *Consider a length $N = 7$, dimension $K = 2$ low-density parity-check code C consisting of the following set of codewords.*

$$C = \left\{ \ (0000000), (1110000), (0000111), (1110111) \ \right\}$$

The probability that the ML decoder outputs a different codeword than the one that was transmitted can be determined by first visualizing the code in two dimensions. With the first three coordinates of the code mapped to the x-axis and the last three coordinates mapped to the y-axis, the code can be visualized as shown in Figure 3.9. From this two-dimensional visualization of the code it is easy to establish optimal decision boundaries between the codeword points at $x = 0.5$ and $y = 0.5$.

Since the code is linear, and assuming that each codeword is equiprobable, the probability of error can be computed assuming the all-zeros codeword (0000000) *was transmitted. A codeword error using the ML decoder occurs when at least one of the other three codewords has lower cost than the all-zeros codeword, as computed by equation (2.5). In particular, it is easy to visualize in Figure 3.9 that if either the first three coordinates or the last three coordinates of the code have lower cost when decoded*

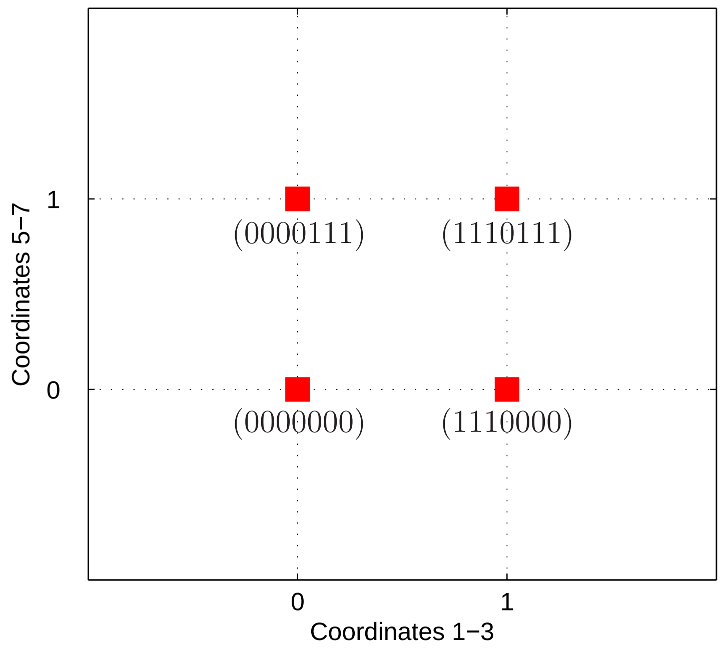


Figure 3.9: Two-dimensional visualization of the length $N = 7$ dimension $K = 2$ LDPC code.

to (111) *as opposed to* (000)*, an ML decoder error will occur. This corresponds to the condition that either*

$$\sum_{i=1}^{3} \lambda_i < 0$$

or

$$\sum_{i=5}^{7} \lambda_i < 0.$$

On the AWGN channel with variance σ^2, this condition reduces to

$$\begin{aligned} 0 \quad > \quad & \sum_{i=1}^{3} \log \left(\frac{\frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(y_i+1)^2}{2\sigma^2}}}{\frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(y_i-1)^2}{2\sigma^2}}} \right) \\ & = \sum_{i=1}^{3} \log \left(e^{\frac{-(y_i+1)^2+(y_i-1)^2}{2\sigma^2}} \right) \\ & = \sum_{i=1}^{3} \frac{-2y_i}{\sigma^2} \end{aligned} \tag{3.2}$$

or

$$\sum_{i=5}^{7} \frac{-2y_i}{\sigma^2} < 0 \tag{3.3}$$

Since ML decoding is only interested in the argument that minimizes cost, the conditions of (3.2) and (3.3) can be simplified to

$$\sum_{i=1}^{3} y_i > 0$$

and

$$\sum_{i=5}^{7} y_i > 0.$$

Since the all-zeros codeword was transmitted, each coordinate $c_i = 0$ is modulated and transmitted as $x_i = m(c_i) = -1$. The probability that a single y_i is greater than zero is given by

$$P(y_i > 0) = \int_0^\infty \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(y+1)^2}{2\sigma^2}} dy$$

where y_i is a Gaussian random variable with mean $\mu_{y_i} = -1$ and variance σ^2. The *probability that the sum of three y_i's is greater than 0 is given by*

$$P(y_i + y_j + y_k > 0) = \int_0^\infty \frac{1}{\sqrt{2\pi(3\sigma^2)}} e^{\frac{-(y+3)^2}{2(3\sigma^2)}} dy,$$

since y_i, y_j, and y_k are independent Gaussian variables. Finally, a decoding error occurs at each bit c_1, c_2, c_3, c_5, c_6, and c_7 with the same probability

$$P_{b,c_1} = P_{b,c_2} = P_{b,c_3} = P_{b,c_5} = P_{b,c_6} = P_{b,c_7} = \int_0^\infty \frac{1}{\sqrt{2\pi(3\sigma^2)}} e^{\frac{-(y+3)^2}{2(3\sigma^2)}} dy$$

when using ML decoding. Furthermore, $P_{b,c_4} = 0$ because none of the codewords have non-zero values at bit position c_4. The overall probability of word error using ML decoding is given by

$$P_w = 2\int_0^\infty \frac{1}{\sqrt{2\pi(3\sigma^2)}} e^{\frac{-(y+3)^2}{2(3\sigma^2)}} dy - \left(\int_0^\infty \frac{1}{\sqrt{2\pi(3\sigma^2)}} e^{\frac{-(y+3)^2}{2(3\sigma^2)}} dy\right)^2.$$

Now, consider the following parity-check matrix for the length $N = 7$, dimension $K = 2$ LDPC code where $\mathbf{c} \in C$ if and only if $H\mathbf{c} = \mathbf{0}$,

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

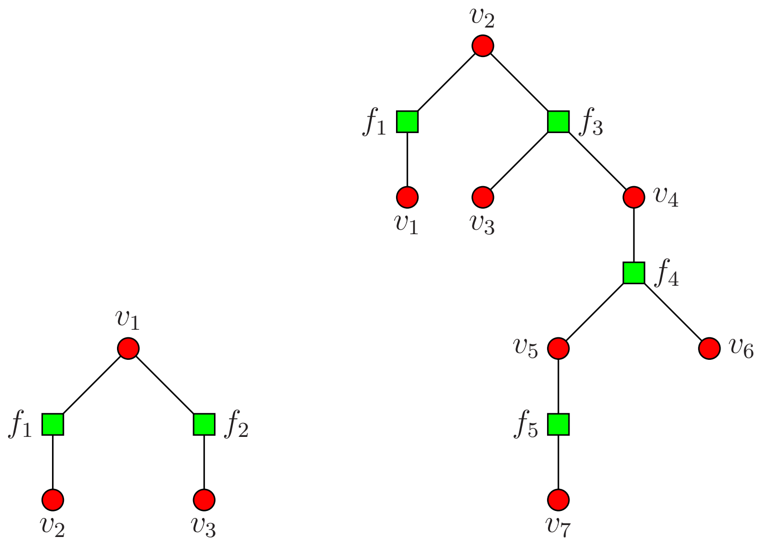
The Tanner graph representation of H is shown in Figure 3.6. For each variable node, an independent tree is shown in Figures 3.10 and 3.11. Applying independent tree *decoding as given in Algorithm 3.1.2 ensures that the minimum-cost configuration on each independent tree is chosen. The independent trees each define a code consisting of all valid configurations on their respective independent trees. The codes corresponding to each of the independent trees are shown below, with the root node highlighted in red. Independent trees rooted at* v_1 *and* v_7 *define permutation equivalent codes, and thus only the code on the independent tree rooted at* v_1 *is given. The independent trees rooted at* v_2*,* v_3*,* v_5*, and* v_6 *also define permutation equivalent codes, so only the code on the independent tree rooted at* v_2 *is given.*

$$\mathcal{C}_{v_1} = \left\{ \; (111), (000) \; \right\}$$

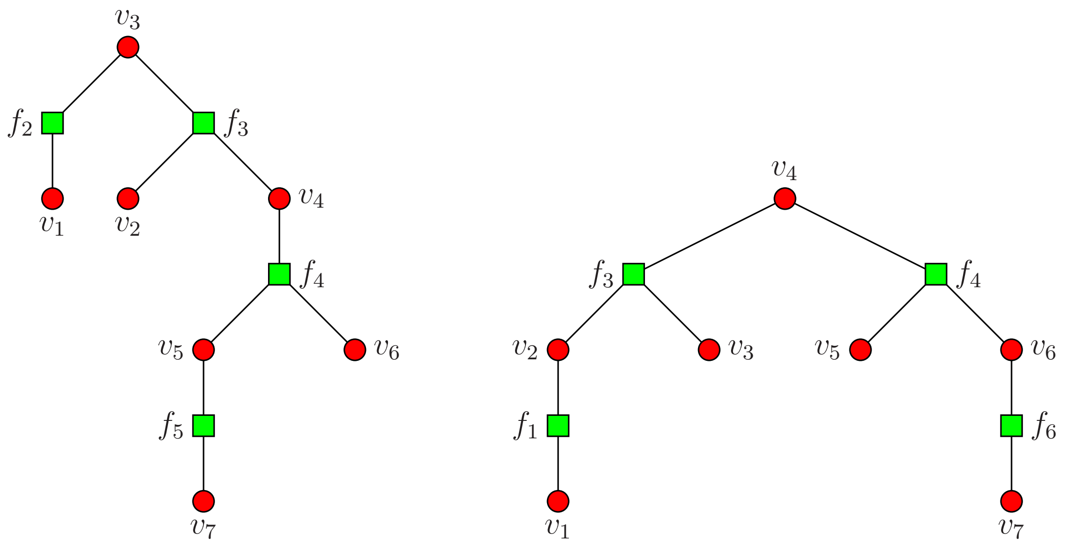
$$\mathcal{C}_{v_2} = \left\{ \begin{array}{c} (0000000), (0000111), (0011010), \\ (0011101), (1110000), (1110111), \\ (1101010), (1101101) \end{array} \right\}$$

$$\mathcal{C}_{v_4} = \left\{ \begin{array}{c} (0000000), (1110111), (1110000), \\ (0000111), (0011100), (1101100), \\ (1101011), (0011011) \end{array} \right\}$$

The probability of error at the root node of each independent tree can be determined by computing the probability of choosing any valid configuration on the independent tree where the root node is equal to a binary 1. The probability that either v_1 *or* v_7 *is*

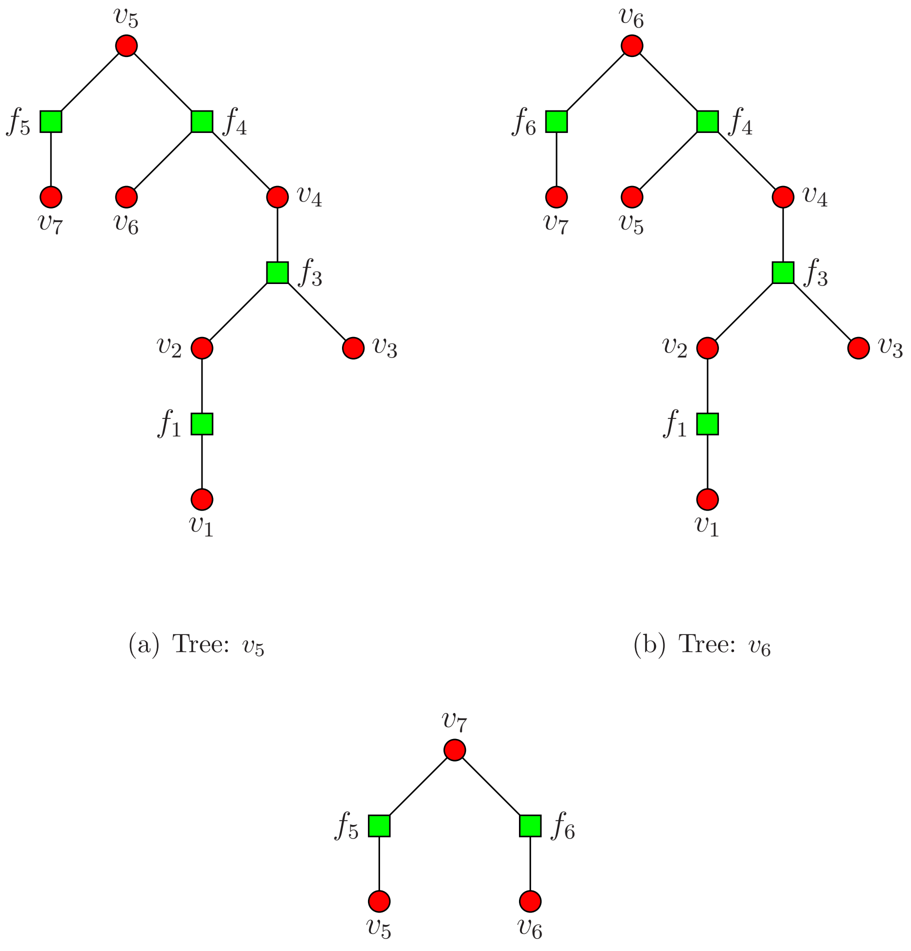


(a) Tree: v_1 (b) Tree: v_2



(c) Tree: v_3 (d) Tree: v_4

Figure 3.10: Independent trees rooted at v_1, v_2, v_3, and v_4 for the LDPC code.



(a) Tree: v_5 (b) Tree: v_6

(c) Tree: v_7

Figure 3.11: Independent trees rooted at v_5, v_6, and v_7 for the LDPC code.

decoded to a binary 1 is given by

$$P_{b,v_1} = P_{b,v_7} = \int_0^\infty \frac{1}{\sqrt{2\pi(3\sigma^2)}} e^{\frac{-(y+3)^2}{2(3\sigma^2)}} dy.$$

The probability of error for bits v_2, v_3, v_4, v_5, and v_6 is more difficult to derive. Each of the independent trees has four valid configurations where the root node is equal to *a binary 1. The probability of error for bits v_2, v_3, v_5, and v_6 is given by*

$$P_{b,v_2} = P_{b,v_3} = P_{b,v_5} = P_{b,v_6} = \Pr\left(\bigcup_{i=5}^{8} E_i\right) \tag{3.4}$$

where E_i is the event that a valid configuration, given by $\varsigma_{v_2,i}$, on the independent tree rooted at v_2 has minimum cost on the independent tree. The set of events $\{E_1, \ldots, E_4\}$ corresponds to the independent tree decoder outputting a value of 1 at the root node of the trees rooted at v_2, v_3, v_5, and v_6. In order for the valid configuration $\varsigma_{v_2,i}$ to have minimum cost on the independent tree, it is necessary for the cost of choosing $\varsigma_{v_2,i}$ to be less than the cost of choosing the all-zeros configuration $\varsigma_{v_2,1}$. Therefore, the expression given in equation (3.4) can be loosened to an upper bound on the probability of root node error given by

$$P_{b,v_2} = P_{b,v_3} = P_{b,v_5} = P_{b,v_6} \leq \Pr\left(\bigcup_{i=5}^{8} B_i\right) \tag{3.5}$$

where B_i is the event that $\varsigma_{v_2,i}$ has cost less than $\varsigma_{v_2,1}$. Using the union bound, (3.5) can be further relaxed to

$$P_{b,v_2} = P_{b,v_3} = P_{b,v_5} = P_{b,v_6} \leq \sum_{i=5}^{8} \Pr\left(B_i\right).$$

The probability of each event B_i can be found by using the Hamming weight of the valid configuration $\varsigma_{v_2,i}$, denoted by $w(\varsigma_{v_2,i})$. Since $\varsigma_{v_2,i}$ and $\varsigma_{v_2,1}$ differ in exactly $w(\varsigma_{v_2,i})$ places, the probability that $\varsigma_{v_2,i}$ has cost less than $\varsigma_{v_2,1}$ is

$$\Pr(B_i) = \int_0^{\infty} \frac{1}{\sqrt{2\pi(w(\varsigma_{v_2,i})\sigma^2)}} e^{\frac{-(y+w(\varsigma_{v_2,i}))^2}{2(3\sigma^2)}} dy.$$

Thus the probability of error at the root node of the independent trees rooted at v_2, v_3, v_5, and v_6 is upper-bounded by

$$\begin{aligned}
P_{b,(v_2,v_3,v_5,v_6)} &\leq \sum_{i=5}^{8} \int_0^\infty \frac{1}{\sqrt{2\pi(w(\varsigma_{v_2,i})\sigma^2)}} e^{\frac{-(y+w(\varsigma_{v_2,i}))^2}{2(3\sigma^2)}} dy \\
&= \int_0^\infty \frac{1}{\sqrt{2\pi(3\sigma^2)}} e^{\frac{-(y+3)^2}{2(3\sigma^2)}} dy + \int_0^\infty \frac{1}{\sqrt{2\pi(4\sigma^2)}} e^{\frac{-(y+4)^2}{2(4\sigma^2)}} dy \\
&\quad + \int_0^\infty \frac{1}{\sqrt{2\pi(5\sigma^2)}} e^{\frac{-(y+5)^2}{2(5\sigma^2)}} dy + \int_0^\infty \frac{1}{\sqrt{2\pi(6\sigma^2)}} e^{\frac{-(y+6)^2}{2(6\sigma^2)}} dy.
\end{aligned}$$

Similarly, the probability of error at the root node of the independent tree for v_4 is upper bounded by

$$\begin{aligned}
P_{b,v_4} &\leq \sum_{i=5}^{8} \int_0^\infty \frac{1}{\sqrt{2\pi(w(\varsigma_{v_2,i})\sigma^2)}} e^{\frac{-(y+w(\varsigma_{v_2,i}))^2}{2(3\sigma^2)}} dy \\
&= \int_0^\infty \frac{1}{\sqrt{2\pi(3\sigma^2)}} e^{\frac{-(y_i+3)^2}{2(3\sigma^2)}} dy + 2\left(\int_0^\infty \frac{1}{\sqrt{2\pi(4\sigma^2)}} e^{\frac{-(y+4)^2}{2(4\sigma^2)}} dy\right) \\
&\quad + \int_0^\infty \frac{1}{\sqrt{2\pi(5\sigma^2)}} e^{\frac{-(y+5)^2}{2(5\sigma^2)}} dy.
\end{aligned}$$

Figure 3.12 shows the exact probability of error for maximum-likelihood decoding for variable nodes v_1, v_2, v_3, v_5, v_6, and v_7. The exact probability of error for variable nodes v_1 and v_7 using finite tree decoding on independent trees is identical to that of ML decoding, and is denoted ML(123567)/IT(17) in Figure 3.12. The upper bound on the probability of error for variable nodes v_2, v_3, v_4, v_5, and v_6 using finite tree decoding on independent trees is denoted IT(2356):UB, and is also shown in Figure 3.12. As the SNR becomes large, the performance of IT decoding approaches that of ML decoding for variable nodes v_2, v_3, v_5, and v_6. This is because high-weight *valid configurations on the independent trees become less likely to cause errors at high SNR, and the bound is dominated by the minimum-weight valid configurations on the independent trees. It is worth noting that, while variable node v_4 has the highest upper bound on the probability of error, denoted IT 4:UB in Figure 3.12, using finite tree decoding on the independent tree, the probability of error using ML decoding is zero since all of the codewords in C have a binary 0 in the fourth coordinate.*

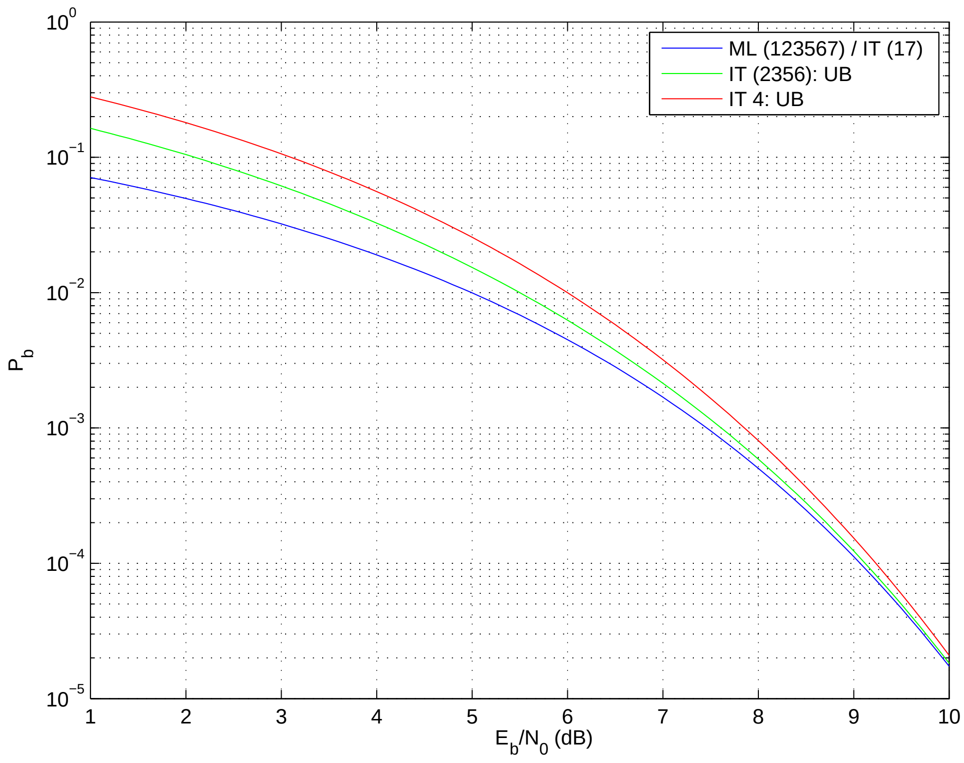


Figure 3.12: Probability of bit error for ML and IT decoding of the length $N = 7$, dimension $K = 2$ code.

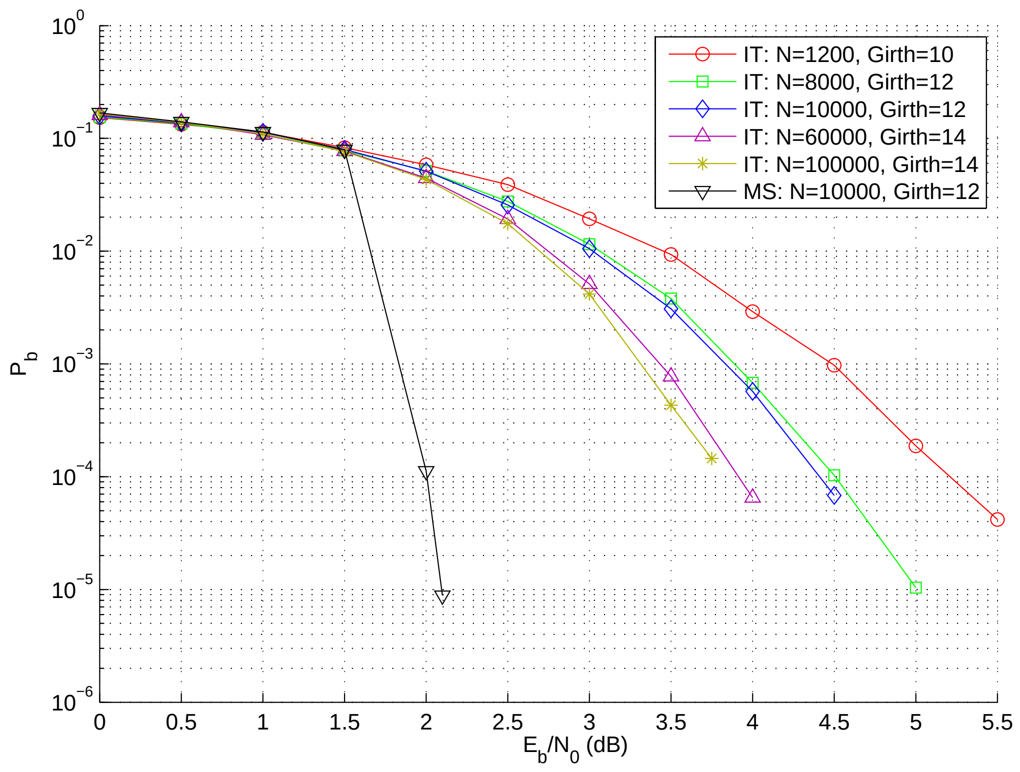


Figure 3.13: Probability of bit error for several rate $\frac{K}{N} = \frac{1}{2}$, $(3, 6)$-regular LDPC codes decoded with the finite tree decoder on independent trees and the MS decoding performance of a length $N = 10000$, $(3, 6)$-regular LDPC code.

The results of Example 3.2 demonstrate a weakness in the independent tree construction process. While some variable nodes are decoded with asymptotically ML performance, others are not. Achieving asymptotic ML decoder performance requires that each tree has minimum-weight deviations with weight equal to the minimum-weight codeword in C involving the root node. While the performance of finite tree decoding on independent trees is reasonable for codes with short block length, such as the one used in Example 3.2, it is much worse than the performance of MS decoding for larger LDPC codes. Figure 3.13 shows the performance of finite tree decoding on independent trees for several rate $\frac{K}{N} = \frac{1}{2}$, $(3, 6)$-regular LDPC codes. MS decoding performance of a length $N = 10{,}000$ LDPC code outperforms finite tree decoding on independent trees with lengths up to $N = 100{,}000$ by approximately 1.75 dB at $P_b = 10^{-4}$. This performance gap might be attributed to the fact that the independent trees often contain less than 80% of the variable nodes in the Tanner graph, and less than 70% of the check nodes in the Tanner graph when constructed from large $(3, 6)$-regular LDPC codes. Therefore, not all the available channel information is used to decode each bit.

Recall that the goal of decoding on trees is to have performance comparable to that of current iterative decoders. Therefore, a new approach to tree construction is needed to make finite tree decoding competitive with current iterative decoders.

3.3 Extrinsic Tree Construction

Independent trees are restricted to only allow one copy of each variable node to appear in the tree. Since the variable nodes are each affected by independent channel noise, this restriction guarantees that each addition to the independent tree either increases or maintains the weight of all existing deviations. However, it is possible that additional copies of a node would increase the minimum deviation weight.

A new method for building finite trees, called extrinsic tree construction, is now studied. This new construction method aims to construct finite trees with high minimum-weight deviations. Much like iterative tree construction, the nodes are added to the tree using a decision criteria directly related to the probability of error at the root node. Whereas the decisions made during iterative tree construction were SNR dependent, the decisions made during extrinsic tree construction are not SNR dependent. Extrinsic tree construction focuses only on low-weight deviations on the tree, aiming to minimize the error rates as SNR grows large. While independent trees are also built with the goal of maximizing the minimum deviation weight, extrinsic trees allow multiple copies of variable nodes to appear in the tree if they increase the weight of the minimum-weight deviations or decrease their multiplicity. The process for building extrinsic trees rooted at each variable node in the code is given by Algorithm 3.3.1.

Algorithm 3.3.1 (Extrinsic Tree Construction)**.**

```
for $i = 1, \dots, N$
 – Root the tree at variable node $v_i$.
 – Set the level of the tree to $\ell = 0$.
 – Set the minimum-weight deviation to weight 1.0 with multiplicity 1.
   while one or more variable nodes exist in level $\ell$ of the tree.
```

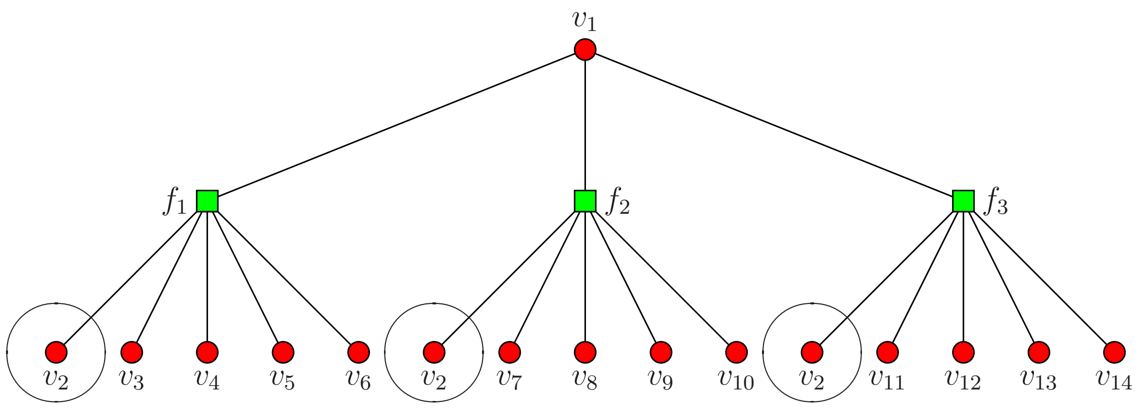
```
for each variable node $v_k$ in the extrinsic tree at level $\ell$.
  if check node $f_j \in N(v_k)$ is not the parent node of $v_k$.
   –Connect $f_j$ to $v_k$, and connect all other variable nodes
    $v_l \in N(f_j) \backslash v_k$ to $f_j$ at level $\ell + 1$.
   –Find the weight and multiplicity of all deviations.
    for $m = 1, \ldots, \mathcal{W}$
      if the $m^{\text{th}}$ smallest deviation weight is increased, or kept
        the same with decreased multiplicity
       –Keep $f_j$ and its corresponding child variable nodes.
       –Exit for loop.
      if the $m^{\text{th}}$ smallest deviation weight is decreased, or kept
        the same with increased multiplicity
       –Eliminate $f_j$ and its corresponding variable nodes
       –Exit for loop.
      if the $m^{\text{th}}$ smallest deviation weight and multiplicity are
        kept the same and $m = \mathcal{W}$
       –Eliminate $f_j$ and its corresponding variable nodes
    end
end
$\ell = \ell + 1$.
```

```
    end
end
```

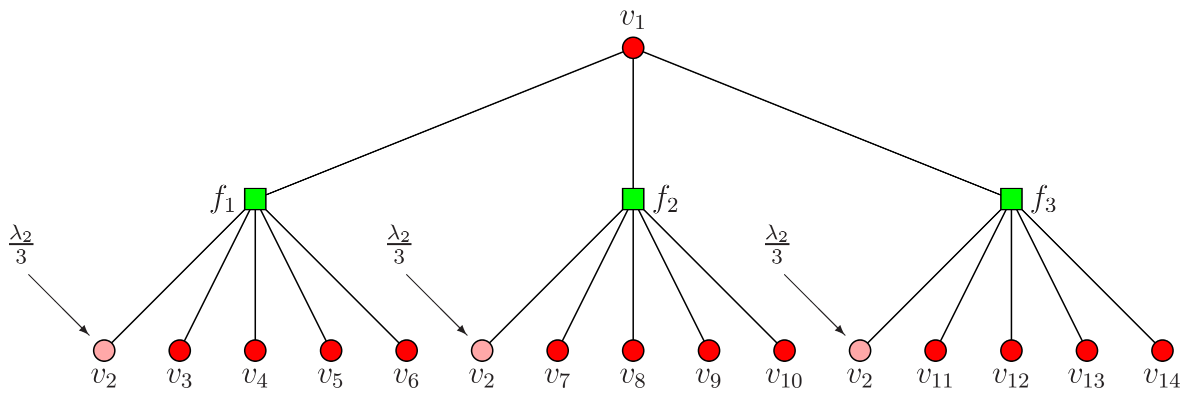
The maximum number of deviation weights used by Algorithm 3.3.1 to make a decision is denoted by $\mathcal{W}$. The addition of a check node and its corresponding variable nodes does not always change the weight of the minimum-weight deviation or its multiplicity, so additional deviation weights beyond the minimum can be used to determine if the additional nodes increase or decrease the weight and multiplicity of higher-weight deviations.

When using the extrinsic tree construction algorithm, it is possible for multiple copies of an arbitrary variable node v_k to appear in the extrinsic tree. In this case, after extrinsic tree construction the log-likelihood ratio (LLR) cost λ_k is split evenly among each of the copies of variable node v_k. Figure 3.14 shows an example where there are three copies of variable node v_2 in the tree. In this case, a scaled LLR cost of $\frac{\lambda_2}{3}$ is assigned to each copy of v_2. All other variable nodes with only a single copy in the tree are assigned an unscaled LLR cost. Note that, although each copy of v_2 appears on the same level in Figure 3.14, the LLR cost is scaled by the total number of copies of v_2 in the extrinsic tree regardless of what level the variable node appear on.

Justification for evenly splitting the cost among the copies of the variable node comes from an examination of deviation weight. First, consider the BIAWGN channel



(a) Multiple copies of v_2 on the extrinsic tree.



(b) Distributing the cost of v_2 between all three copies.

Figure 3.14: Example of extrinsic tree cost scaling among multiple copies of the same node.

where the LLR cost reduces to $\boldsymbol{\lambda} = -\frac{2}{\sigma^2}\mathbf{y}$. Now, consider a deviation consisting of the set of variable nodes $\{v_1, \ldots, v_\omega\}$, where there are a_i copies of each variable node v_i in the deviation, and b_i copies of each variable node v_i in the entire extrinsic tree. The probability that each node in the deviation was transmitted over the channel as a -1 and that the deviation has cost

$$\sum_{i=1}^{\omega} \frac{a_i}{b_i}(-\frac{2}{\sigma^2} y_i) < 0,$$

can be found by computing the probability that a sample of a Gaussian random variable with distribution

$$\mathcal{N}\left(\sum_{i=1}^{\omega} \frac{a_i}{b_i} \ , \ \sum_{i=1}^{\omega} (\frac{a_i}{b_i}\sigma)^2\right)$$

is less than zero. This random variable can be rescaled resulting in the distribution given by

$$\mathcal{N}\left(\left(\frac{\sum_{i=1}^{\omega} \frac{a_i}{b_i}}{\sqrt{\sum_{i=1}^{\omega} (\frac{a_i}{b_i})^2}}\right)^2 , \left(\frac{\sum_{i=1}^{\omega} \frac{a_i}{b_i}}{\sqrt{\sum_{i=1}^{\omega} (\frac{a_i}{b_i})^2}}\right)^2 \sigma^2\right),$$

where the mean

$$\left(\frac{\sum_{i=1}^{\omega} \frac{a_i}{b_i}}{\sqrt{\sum_{i=1}^{\omega} (\frac{a_i}{b_i})^2}}\right)^2 \tag{3.6}$$

is precisely the deviation weight used when constructing the extrinsic tree.

To understand why each variable v_i is assigned a log-likelihood ratio cost scaled by $\frac{1}{b_i}$, consider a deviation where the set of variable nodes $\{v_1, \ldots, v_\omega\}$ corresponds with the binary 1's of a codeword on the original Tanner graph, and the set $\{a_1, \ldots, a_\omega\} = \{b_1, \ldots, b_\omega\}$ corresponds to the number of copies of each variable node in the extrinsic tree contained in the deviation. Deviations where the set of variable nodes $\{v_1, \ldots, v_\omega\}$ correspond with the binary 1's of a codeword are referred to as *codeword deviations*.

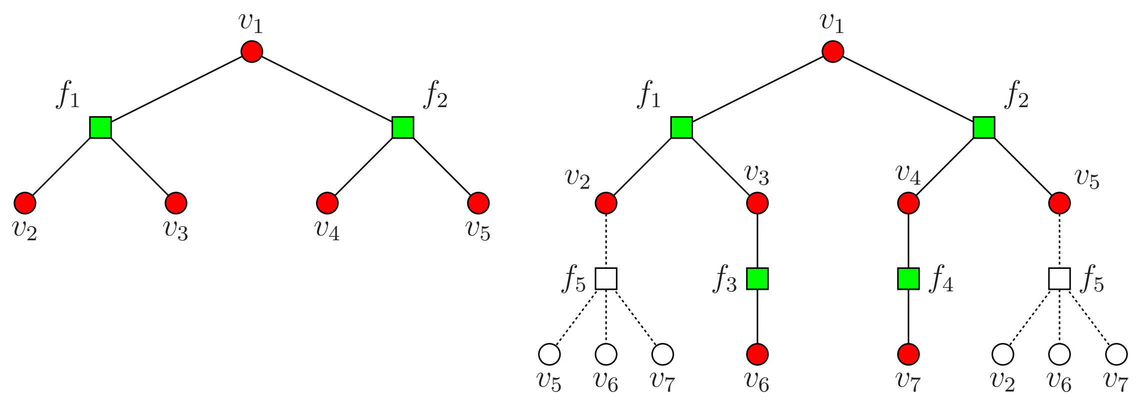
The weight of a codeword deviation reduces to $\frac{\omega^2}{\omega} = \omega$, since $\frac{a_i}{b_i} = 1$ for all i. Thus, the probability of a codeword deviation error can be found using the Hamming weight of the codeword, and is equal to the corresponding codeword error probability using maximum likelihood (ML) decoding.

The following example demonstrates the extrinsic tree construction process for the length 7, dimension 3 LDPC code whose Tanner graph is given in Figure 3.1. This example considers the extrinsic tree rooted at variable node v_1. After rooting the extrinsic tree, there is only one deviation and its weight is 1.0. Figure 3.15(a) shows the extrinsic tree with a completed Level 1 of check nodes and variable nodes. Check node f_1 with child variable nodes v_2 and v_3 is added to the tree below the root node because the minimum deviation weight is increased from 1.0 to 2.0 with a multiplicity of 2. Then, check node f_2 with child variable nodes v_4 and v_5 is added to the extrinsic tree since this increases the minimum deviation weight from 2.0 to 3.0 with a multiplicity of 4. Because variable node v_1 is only connected to two check nodes in the Tanner graph, Level 1 is complete.

A completed Level 2 of the extrinsic tree is shown in Figure 3.15(b). The construction of Level 2 begins with check node f_5 and its child variable nodes v_5, v_6, and v_7. Check node f_5 is not added to the extrinsic tree because this would decrease the minimum deviation weight from 3.0 to 2.78. Next, check node f_3 and variable node v_6 are added to the extrinsic tree since they maintain the minimum deviation weight at 3.0, and decrease the multiplicity of the deviations with weight equal to 3.0 from 4 to 2. Check node f_4 and variable node v_7 are also added since they decrease the multiplicity of weight 3.0 deviations from 2 to 1. Finally, check node f_5 and child variable nodes v_2, v_6, and v_7 are not added because, although they maintain the minimum deviation weight at 3.0, they do not decrease the multiplicity.

The minimum-weight deviation includes variable nodes v_1, v_2, and v_5, which are on Levels 0, 1, and 1, respectively. Therefore, no variable nodes will be added to Level 3 since additional variable nodes will have no effect on the current weight or multiplicity of the minimum-weight deviation including v_1, v_2, and v_5. The final extrinsic tree rooted at variable node v_1 is given in Figure 3.15(b). Note that the minimum-weight codeword in C in which v_1 is involved corresponds with the minimum-weight deviation on the extrinsic tree, so the probability of bit error approaches that of ML decoding as the SNR gets large.

Figure 3.16 shows the decoding performance of finite tree decoding on extrinsic trees with $\mathcal{W} = 1$ and $\mathcal{W} = 2$ and compares it to min-sum decoding at iterations 2 and 400 on a $(3, 6)$-regular low-density parity-check code with length $N = 40$ and dimension $K = 20$. The extrinsic trees constructed for this code average $A = 50.5$ variable nodes per tree when $\mathcal{W} = 1$ and $A = 56.3$ variable nodes per tree when $\mathcal{W} = 2$, and the maximum depth of the extrinsic trees is $\ell = 3$. Computation trees modeling MS after 2 iterations each have 166 variable nodes, yet the bit error rates



(a) Level 1 (b) Level 2

Figure 3.15: Extrinsic tree rooted at variable node v_1.

for finite tree-based decoding on extrinsic trees are lower for all SNR tested above 8 dB. Finite tree decoding probability of bit error is lower than that of MS with 400 iterations for SNRs above 9.5 dB. The upper bound on the probability of bit error of finite tree decoding of extrinsic trees is also given in Figure 3.16. The upper bound is computed enumerating all of the deviations and using equation (2.10). At SNRs above 9.5 dB, the performance of finite tree decoding on extrinsic trees is very closely predicted by the upper bound. Using the upper bound, it is shown in Figure 3.16 that the performance can be guaranteed at bit error rates well below the reach of simulations. It is worth noting that extrinsic tree construction for both $\mathcal{W} = 1$ and $\mathcal{W} = 2$ yielded deviations with minimum-weight equal to 2.0 among all the extrinsic trees. The two trees rooted at variable nodes v_{21} and v_{26} contained the

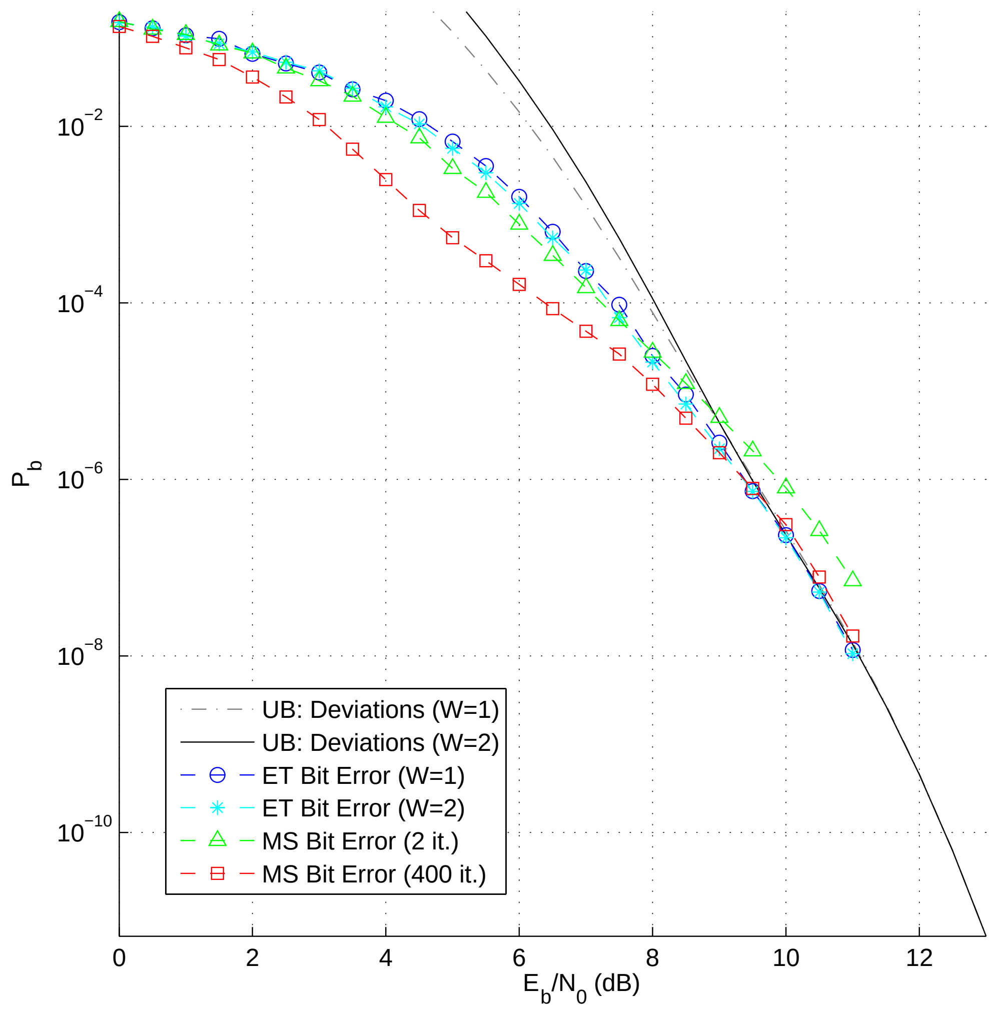


Figure 3.16: Probability of bit error for a (3,6)-regular (40,20) LDPC code decoded with the finite tree decoder on extrinsic trees with $\mathcal{W} = 1$ and $\mathcal{W} = 2$ and the MS decoder after 2 and 400 iterations.

weight 2.0 deviations, and the set of coordinates $\{21, 26\}$ also forms a codeword in C. Therefore, finite tree decoding performance on the extrinsic trees will approach ML decoding performance at high SNRs.

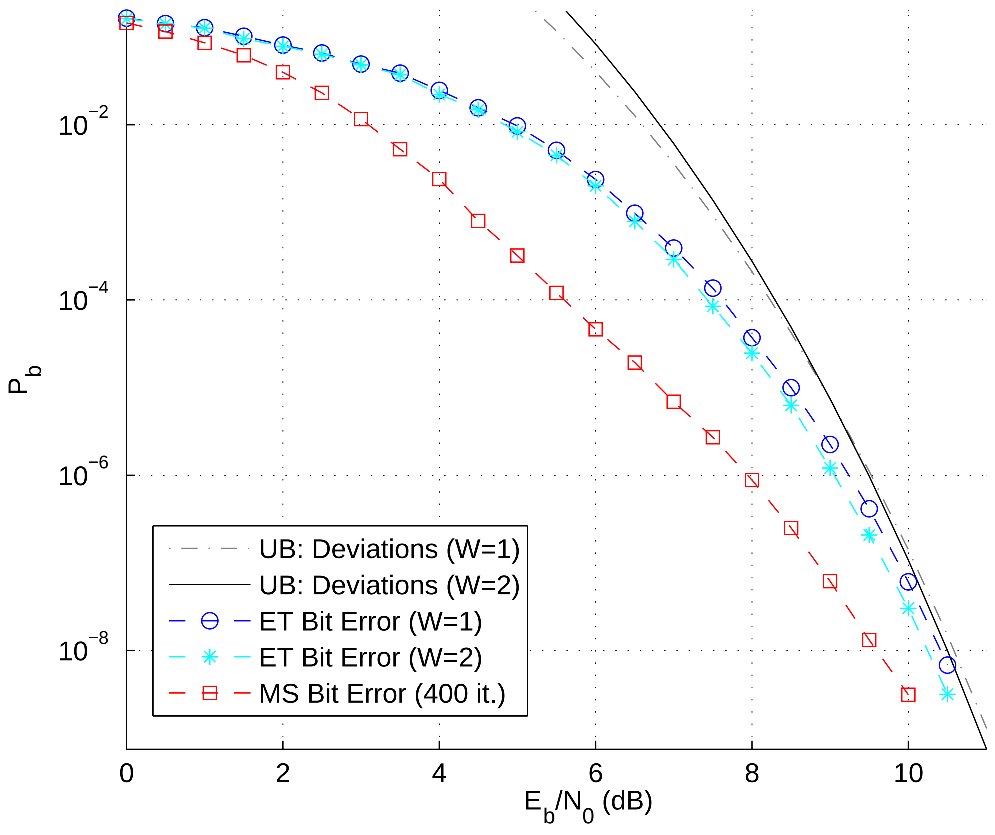


Figure 3.17: Probability of bit error for a (3,5)-regular (50,20) LDPC code decoded with the finite tree decoder on extrinsic trees with $\mathcal{W} = 1$ and $\mathcal{W} = 2$ and the MS decoder with 400 iterations.

Figure 3.17 shows the results of finite tree decoding on extrinsic trees with $\mathcal{W} = 1$

and $\mathcal{W} = 2$ and compares it to min-sum decoding at 400 iterations on a $(3, 5)$-regular low-density parity-check code with length $N = 50$ and dimension $K = 20$. Using this code, the performance of finite tree decoding on extrinsic trees shows noticeable improvement in both simulated probability of bit error and upper bounds when using $\mathcal{W} = 2$ instead of $\mathcal{W} = 1$. This is because the minimum-weight deviation was 3.74 among the extrinsic trees when $\mathcal{W} = 1$ and 3.91 when $\mathcal{W} = 2$. Simulations were performed down to a probability of bit error of 10^{-9} with MS decoding and finite tree decoding on extrinsic trees with $\mathcal{W} = 2$. The current slope of the two simulated bit error rates suggests that they will intersect between the probability of bit errors 10^{-10} and 10^{-11}.

Figure 3.18 shows the decoding performance of finite tree decoding on extrinsic trees with $\mathcal{W} = 1$ and $\mathcal{W} = 2$ and compares it to min-sum decoding at 400 iterations on an irregular low-density parity-check code with length $N = 40$ and dimension $K = 20$. The variable and check node degrees were chosen using the rate $\frac{K}{N} = \frac{1}{2}$ degree distributions given in [18] with maximum variable node degree equal to four. The $(40, 20)$ irregular code has six check nodes with degree five and fourteen check nodes with degree six. The code also has twenty-two variable nodes of degree two, two variable nodes of degree three, and sixteen variable nodes of degree four. Improvements in bit error rates at high SNRs are shown when using $\mathcal{W} = 2$ instead of $\mathcal{W} = 1$. However, both fail to come close to the probability of bit error of MS

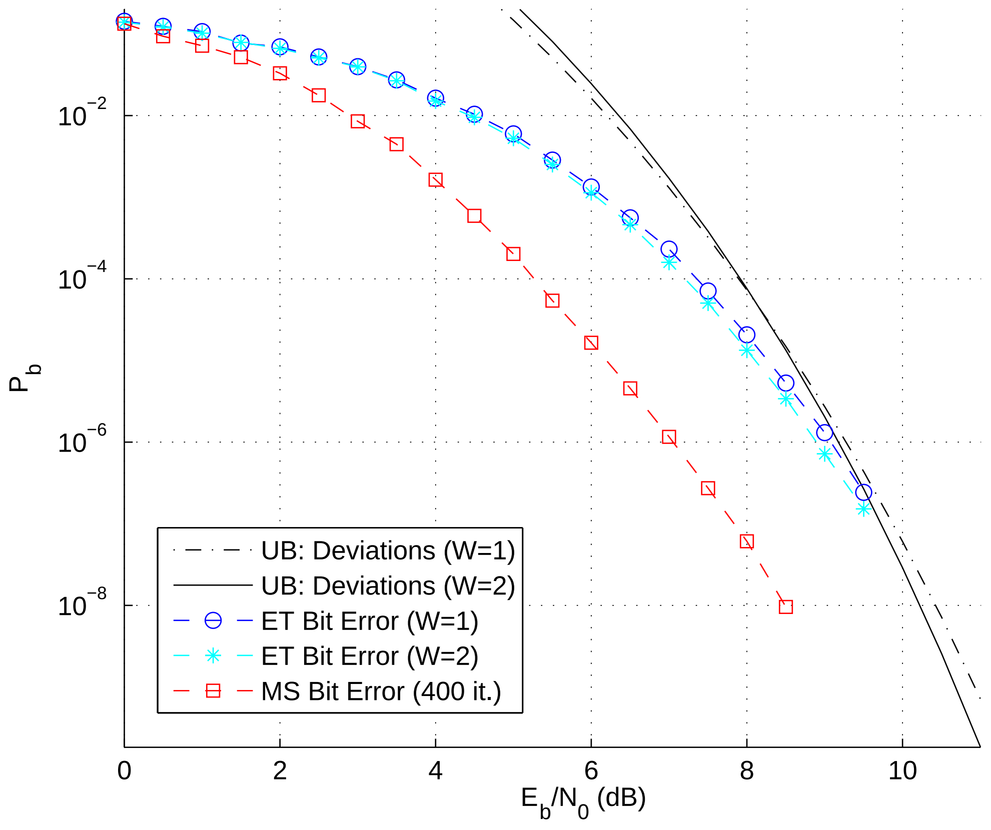


Figure 3.18: Probability of bit error for an irregular (40,20) LDPC code decoded with the finite tree decoder on extrinsic trees with $\mathcal{W} = 1$ and $\mathcal{W} = 2$ and the MS decoder with 400 iterations.

decoding. During extrinsic tree construction, lower minimum-weight deviations are observed for extrinsic trees rooted at degree two variable nodes than degree four variable nodes.

The results shown in Figure 3.16 show that finite tree decoding on extrinsic trees is capable of outperforming min-sum decoding at high SNR. Figure 3.17 also shows promising performance at high SNR, and Figure 3.18 shows considerably worse performance than MS decoding over all simulated SNRs. These results suggest that finite tree decoding on extrinsic trees is capable of better performance when using regular LDPC codes.

The performance of finite tree decoding on extrinsic trees is noticeably worse than MS decoding at lower SNRs. However, by increasing $\mathcal{W}$, it is shown that the performance of finite tree decoding improves in each of the simulations. Since the extrinsic tree construction of Algorithm 3.3.1 is designed to obtain high minimum-weight deviations, it is not surprising that good performance is observed at high SNR. At low SNR the probability of bit error is not always determined by the low-weight deviations. Since only the $\mathcal{W}$ lowest deviation weights are addressed by the construction given in Algorithm 3.3.1, it is not surprising that the performance of finite tree decoding on extrinsic trees suffers at low SNRs. While increasing $\mathcal{W}$ is shown to improve the performance of finite tree decoding on extrinsic trees, it also results in a significant increase in the computational complexity of the extrinsic tree construction. This is because an increase in $\mathcal{W}$ results in more variable nodes being included in the extrinsic trees, thus causing an increase in the number of deviations and deviation weights that are computed by Algorithm 3.3.1. Simulation results given in this section were limited to block length $N \leq 50$ in order to limit the time duration of extrinsic tree construction.

While it is feasible to construct extrinsic trees for block length $N = 40$ and $N = 50$ low-density parity-check codes, like the ones whose simulation results are given in Figures 3.16, 3.17, and 3.18, the number of deviations in the extrinsic trees causes the complexity of Algorithm 3.3.1 to grow exponentially with the number of variable nodes currently in the extrinsic tree. For this reason, it is worthwhile to consider properties related to independent trees and extrinsic trees that could potentially lead to simplified construction techniques or alternative methods for finite tree-based construction.

3.4 Properties of Independent and Extrinsic Trees

Independent trees and extrinsic trees are built node-by-node using Algorithms 3.2.1 and Algorithm 3.3.1, respectively. This section examines some of the properties of independent and extrinsic trees. First, Proposition 3.4.1 gives an upper bound on the depth of independent trees.

Proposition 3.4.1. *The maximum depth ℓ_{max} of an independent tree built according to Algorithm 3.2.1, for a (d_V, d_F)-regular LDPC code of length N, satisfies*

$$\ell_{max} \leq \left\lfloor \frac{N-1}{d_F - 1} \right\rfloor .$$

Proof. Use the following construction method to build an independent tree with a minimal number of nodes per level. Starting at the root node, there is one variable node in the tree at level $\ell = 0$. The root node connects to one check node, and this check node has $d_F - 1$ child variable nodes at level $\ell = 1$. There are now a total of d_F variable nodes in the tree. Continue by adding a single check node below only one of the variable nodes on level $\ell = 1$. The $d_F - 1$ child variable nodes get added to the tree at level $\ell = 2$. There are a total of $1 + 2(d_F - 1)$ variable nodes in the tree. If this construction continues, there will be $1 + \ell(d_F - 1)$ variable nodes in the tree after ℓ levels have been added. Each level of variable nodes must be connected to at least one check node on the next level in order for the tree to continue growing. Since the construction method only allows one connection from each level of variable nodes to the a single check node below, the resulting tree contains a minimal number of variable nodes per level. Since each of the N variable nodes appears in the tree at most once, we know $1 + \ell(d_F - 1) \leq N$, i.e. $\ell \leq \frac{N-1}{d_F-1}$. Because ℓ is an integer, we have $\ell \leq \left\lfloor \frac{N-1}{d_F-1} \right\rfloor$ as desired.

$\square$

Proposition 3.4.1 shows that there is a limit to the depth of independent trees that can be computed using the node degrees of a regular low-density parity-check code. The depth of independent trees is important because operations on finite trees can be performed in parallel, as long as they are on the same level. Therefore, the time required to perform finite tree decoding is not limited by the number of nodes in the finite tree, but instead it is limited by the maximum number of levels in the finite tree.

The weight of deviations on an independent tree of a (d_V, d_F)-regular low-density parity-check code is upper bounded in Proposition 3.4.2.

Proposition 3.4.2. *Consider an independent tree constructed using Algorithm 3.2.1 for a (d_V, d_F)-regular LDPC code of length N. The weight of any deviation δ on this tree satisfies*

$$w(\delta) \leq 1 + \left\lfloor \frac{N-1}{d_F - 1} \right\rfloor .$$

Proof. The maximum number of variable nodes that can be in the independent tree is N. There are $d_F - 1$ child variable nodes for each check node in the independent tree. With the exception of the root node, each variable node in the independent tree has exactly one parent check node. Therefore, there can be a maximum of $\lfloor \frac{N-1}{d_F-1} \rfloor$ check nodes in the independent tree. Each check node can be adjacent to a maximum of two variable nodes in the deviation. One of the two is the parent variable node, and the other is one of the child variable nodes. There is a maximum of one child variable node for each of the $\left\lfloor \frac{N-1}{d_F-1} \right\rfloor$ check nodes that can be contained in the deviation. Including the root node, the maximum weight of the deviation is $1 + \left\lfloor \frac{N-1}{d_F-1} \right\rfloor$. □

Propositions 3.4.1 and 3.4.2 upper bound the depth of the independent trees and the weight of the deviations on independent trees for (d_V, d_F)-regular low-density parity-check codes. These two properties help to establish a better understanding of independent trees. Proposition 3.4.1 leads to a limitation on the time required to perform finite tree decoding in independent trees, and Proposition 3.4.2 provides a limitation on the achievable probability of bit error at high SNR. Next, a property of finite trees is examined when making modifications to independent tree construction. In particular, it is shown that there are theoretical advantages to limiting the number of copies of each variable node in the finite trees to two or less.

While constructing independent trees using Algorithm 3.2.1, each additional check node and its child variable nodes that are added to the tree are guaranteed not to decrease the weight of any existing deviations on the independent tree. This is because deviation weight can not be decreased until more than one copy of the same variable node appears in the independent tree. However, it may be advantageous to include additional copies of a variable node in finite trees, as indicated by the performance of finite tree decoding on extrinsic trees. In Proposition 3.4.3, the effect of simultaneously adding multiple copies of the same variable node to a finite tree is examined.

Proposition 3.4.3. *Let Δ_ℓ be the set of deviations that exist on a finite tree after ℓ levels of check nodes and variable nodes have been added below the root node. Assume that some variable node v does not currently exist in the finite tree, and that there are a maximum of 2 copies of each variable node in the finite tree. Adding $k > 0$ copies* *of variable node v, and scaling the cost of each by $\frac{1}{k}$, at level $\ell+1$ can not decrease the weight of an existing deviation in Δ_ℓ, assuming none of the new variable nodes connected to the parent check nodes of v_i previously existed in the finite tree.*

Proof. Assume that a deviation $\delta \in \Delta_\ell$ has deviation weight $w(\delta)$. Suppose that by adding multiple copies of v at level $\ell+1$, the deviation weight $w(\delta)$ has been decreased to $w(\delta_{\text{new}})$. The formula for the original deviation weight is given by

$$w(\delta) = \left(\frac{\displaystyle\sum_{i=1}^{\omega} \frac{a_i}{b_i}}{\sqrt{\sum_{i=1}^{\omega} (\frac{a_i}{b_i})^2}} \right)^2,$$

where b_i is the number of copies of variable node v_i in the finite tree, and a_i is the number of copies of v_i in the deviation δ. Since there are a maximum of two copies of each variable node in the finite tree, $1 \leq a_i \leq b_i \leq 2$. The new deviation weight is given by

$$w(\delta_{\text{new}}) = \left(\frac{\displaystyle\sum_{i=1}^{\omega+1} \frac{a_i}{b_i}}{\sqrt{\sum_{i=1}^{\omega+1} (\frac{a_i}{b_i})^2}} \right)^2,$$

where $a_{\omega+1}$ is the number of copies of variable node v in the deviation δ_{new} and $b_{\omega+1} = k$ is the number of copies of variable node v in the new finite tree. In order for the weight to decrease we must have

$$\left(\frac{\displaystyle\sum_{i=1}^{\omega} \frac{a_i}{b_i}}{\sqrt{\sum_{i=1}^{\omega} (\frac{a_i}{b_i})^2}} \right)^2 > \left(\frac{\displaystyle\sum_{i=1}^{\omega+1} \frac{a_i}{b_i}}{\sqrt{\sum_{i=1}^{\omega+1} (\frac{a_i}{b_i})^2}} \right)^2.$$

Squaring both sides and cross-multiplying results in the inequality

$$\left(\frac{a_1}{b_1}+\ldots+\frac{a_\omega}{b_\omega}\right)^2\left(\frac{a_1^2}{b_1^2}+\ldots+\frac{a_{\omega+1}^2}{b_{\omega+1}^2}\right) > \left(\frac{a_1}{b_1}+\ldots+\frac{a_{\omega+1}}{b_{\omega+1}}\right)^2\left(\frac{a_1^2}{b_1^2}+\ldots+\frac{a_\omega^2}{b_\omega^2}\right).$$

Regrouping yields

$$\left(\tfrac{a_1}{b_1}+\ldots+\tfrac{a_\omega}{b_\omega}\right)^2\left(\left(\tfrac{a_1^2}{b_1^2}+\ldots+\tfrac{a_\omega^2}{b_\omega^2}\right)+\left(\tfrac{a_{\omega+1}}{b_{\omega+1}}\right)^2\right) > \left(\left(\tfrac{a_1}{b_1}+\ldots+\tfrac{a_\omega}{b_\omega}\right)+\tfrac{a_{\omega+1}}{b_{\omega+1}}\right)^2\left(\tfrac{a_1^2}{b_1^2}+\ldots+\tfrac{a_\omega^2}{b_\omega^2}\right),$$

which simplifies to

$$\left(\frac{a_1}{b_1}+\ldots+\frac{a_\omega}{b_\omega}\right)^2\left(\frac{a_{\omega+1}}{b_{\omega+1}}\right) > \left(2\left(\frac{a_1}{b_1}+\ldots+\frac{a_\omega}{b_\omega}\right)+\left(\frac{a_{\omega+1}}{b_{\omega+1}}\right)\right)\left(\frac{a_1^2}{b_1^2}+\ldots+\frac{a_\omega^2}{b_\omega^2}\right)$$

by subtracting common terms and dividing both sides by $\frac{a_{\omega+1}}{b_{\omega+1}}$. Next, dividing both sides by $\left(\frac{a_{\omega+1}}{b_{\omega+1}}\right)\left(\frac{a_1^2}{b_1^2}+\ldots+\frac{a_\omega^2}{b_\omega^2}\right)$ results in

$$\frac{\left(\frac{a_1}{b_1}+\ldots+\frac{a_\omega}{b_\omega}\right)^2}{\left(\frac{a_1^2}{b_1^2}+\ldots+\frac{a_\omega^2}{b_\omega^2}\right)} > 2\left(\frac{a_1}{b_1}+\ldots+\frac{a_\omega}{b_\omega}\right)\left(\frac{b_{\omega+1}}{a_{\omega+1}}\right)+1.$$

The inequality is still satisfied when 1 is subtracted from the right side, and so

$$\frac{\left(\frac{a_1}{b_1}+\ldots+\frac{a_\omega}{b_\omega}\right)^2}{\left(\frac{a_1^2}{b_1^2}+\ldots+\frac{a_\omega^2}{b_\omega^2}\right)} > 2\left(\frac{a_1}{b_1}+\ldots+\frac{a_\omega}{b_\omega}\right)\left(\frac{b_{\omega+1}}{a_{\omega+1}}\right).$$

Dividing both sides of the inequality by $2\left(\frac{a_1}{b_1}+\ldots+\frac{a_\omega}{b_\omega}\right)$ results in

$$\frac{\left(\frac{a_1}{b_1}+\ldots+\frac{a_\omega}{b_\omega}\right)}{2\left(\frac{a_1^2}{b_1^2}+\ldots+\frac{a_\omega^2}{b_\omega^2}\right)} > \left(\frac{b_{\omega+1}}{a_{\omega+1}}\right). \tag{3.7}$$

Since $\frac{a_i}{b_i} \geq \frac{1}{2}$ for $i = 1, \ldots, \omega$, each term $2\left(\frac{a_i^2}{b_i^2}\right)$ is at least $\frac{a_i}{b_i}$. Therefore, the left side of the inequality given by 3.7 is less than or equal to one, and since $\frac{b_{\omega+1}}{a_{\omega+1}} \geq 1$, we have a contradiction. Therefore, the weight of δ_{new} can not be less than the weight of δ.

$\square$

Proposition 3.4.3 implies that finite trees can be constructed with deviation weights that are guaranteed to increase even with multiple copies of the same variable node in the deviation. Although the construction method would only require a slight modification to Algorithm 3.2.1, the weight of each deviation would still need to be found using brute-force methods at the end of the tree construction in order to upper bound the probability of bit error at the root nodes. An alternative construction for finite trees is given in Section 3.5. This new construction results in a much more efficient way to enumerate the deviations in the trees.

3.5 Deviation Path-Forcing Trees

This section introduces a new method for building trees called deviation path-forcing trees (DPFTs). The DPFTs are designed to produce trees with deviations that are simple to enumerate. More specifically, the DPFT structure enables a simple calculation of the weight and multiplicity of each deviation.

The process for building deviation path-forcing trees rooted at a variable node is as follows.

Algorithm 3.5.1 (Deviation Path-Forcing Tree Construction)**.**

```
for $i = 1, \ldots, N$
 – Root the tree at variable node $v$.
– Set the level of the tree to $\ell = 0$.
    while one or more variable nodes exist in level $\ell$ of the tree
        for each variable node $v_k$ contained in level $\ell$ of the DPFT
            if check node $f_j \in N(v_k)$ is not the parent node of $v_k$, and $v_k$
                currently has no children in the DPFT
            –Connect $f_j$ to $v_k$ and connect variable nodes $v_l \in N(f_j)\backslash v_k$
                to $f_j$ at level $\ell + 1$.
                if none of the children $v_l \in N(f_j)\backslash v_k$ are ancestors of $f_j$
                    –Keep $f_j$ and its corresponding variable nodes.
                else
                    –Eliminate $f_j$ and its corresponding variable nodes.
        end
        $\ell = \ell + 1$.
    end
end
```

After building each of the deviation path-forcing trees, log-likelihood ratio costs are assigned to each variable node in the DPFTs. Algorithm 3.1.2 is then used to obtain an output for each of the root nodes. Note that, unlike the construction given in Algorithm 3.3.1, the LLR costs assigned to the variable nodes during finite tree decoding on DPFTs are unscaled.

There are two important properties regarding deviation path-forcing trees that are worth discussing. The first property is that every variable node in the DPFT is connected to two check nodes, except the root node and the leaf nodes, each of which is connected to only one check node. This property forces all deviations in the tree to be paths from the root node to the leaf node. Therefore, every leaf node in the DPFT defines exactly one deviation. As a consequence, enumerating deviations on the DPFTs is straightforward once the trees are constructed.

The second key property of deviation path-forcing trees is the fact that each node contained in a given deviation is distinct. This property is a consequence of step 2(b) in the DPFT construction process. Since each variable node in the deviation is unique, the weight of each deviation is proportional to the level at which the root node of the deviation appears. Example 3.5.2 demonstrates the performance of the DPFT construction.

Example 3.5.2. *A single deviation path-forcing tree has been constructed for a length* $N = 40$ $(3, 6)$*-regular low-density parity-check code rooted at variable node* v_1*. The size of the DPFT grows large as the depth of the tree is allowed to increase. However, once any path in the DPFT is terminated at a leaf node the minimum-weight deviation has already been established. Table 3.6 shows the deviation weights and multiplicities on the DPFT with varying maximum depths. Once the maximum depth of the DPFT is greater than or equal to* $\ell = 3$*, the minimum deviation weight does not increase* *beyond 4.*

Max Depth (ℓ)	$\|w(\delta) = 1\|$	$\|w(\delta) = 2\|$	$\|w(\delta) = 3\|$	$\|w(\delta) = 4\|$	$\|w(\delta) = 5\|$	$\|w(\delta) = 6\|$	$\|w(\delta) = 7\|$
0	1	0	0	0	0	0	0
1	0	5	0	0	0	0	0
2	0	0	25	0	0	0	0
3	0	0	0	125	0	0	0
4	0	0	0	6	595	0	0
5	0	0	0	6	65	2650	0
6	0	0	0	6	65	397	$11,265$

Table 3.6: Deviation weight and multiplicity for different maximum depths of the DPFT rooted at v_1 for the $N = 40$, $(3, 6)$-regular LDPC code.

Example 3.5.2 shows that the minimum-weight deviations on deviation path-forcing trees are established once a single leaf node is created. If the leaf node is established at level ℓ, additional nodes at level ℓ can decrease the number of deviations with weight $\ell + 1$, but can not increase the minimum weight. Figure 3.19 shows the performance of finite tree decoding on DPFTs with maximum depths of $\ell = 2, 3, 4$ and 5 and compares it to MS decoding at 400 iterations on a $(3, 6)$-regular LDPC code with length $N = 40$ and dimension $K = 20$. The simulation results and upper bounds show steady improvement with increasing ℓ in the probability of bit error of finite tree decoding on DPFTs between SNRs of 5.0 dB and 12.0 dB. After $\ell = 1$, leaf nodes are established on the DPFTs rooted at variable nodes v_{21} and v_{26}, so it is not surprising that increases in ℓ do not improve the performance at high

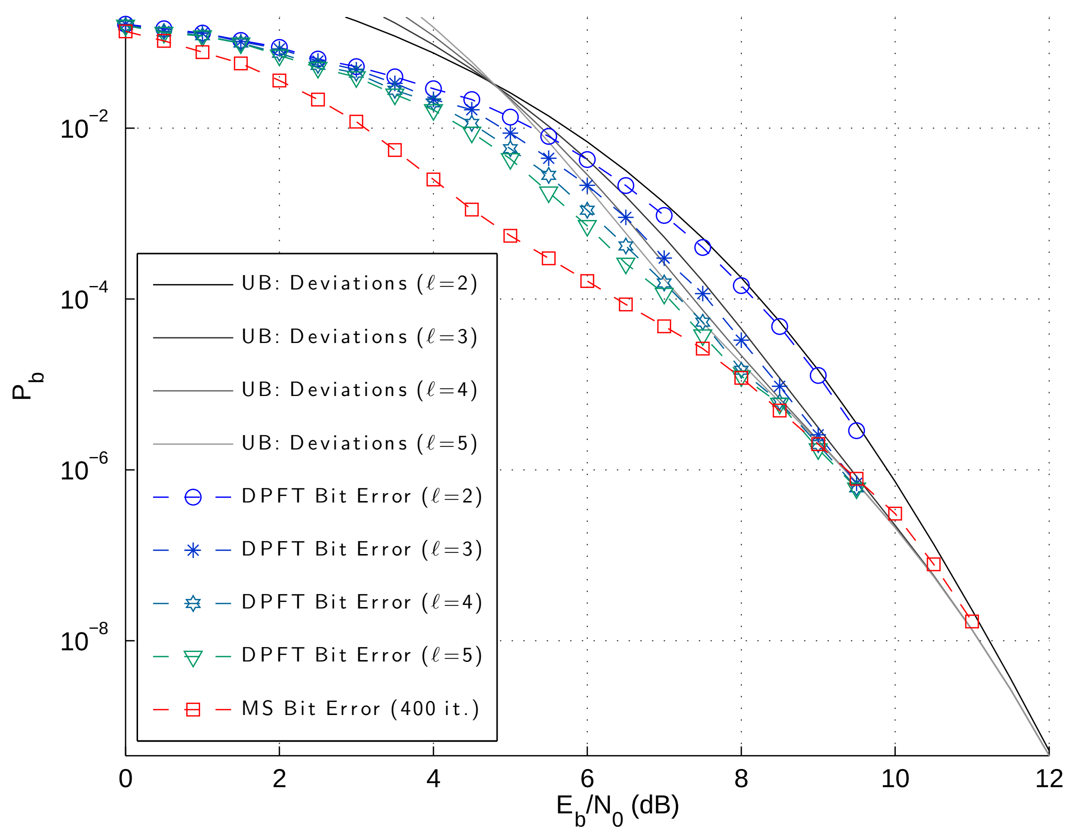


Figure 3.19: Probability of bit error for a (3,6)-regular (40,20) LDPC code decoded with the finite tree decoder on DPFTs with $\ell = 2, 3, 4$ and 5 and the MS decoder with 400 iterations.

SNRs. However, by increasing ℓ more of the low-weight deviations are converted to higher-weight deviations, thus decreasing the probability of bit error at lower SNRs.

The performance of finite tree decoding on deviation path-forcing trees with a maximum depth of $\ell = 2, 3, 4$ and 5 and min-sum decoding at 400 iterations are given

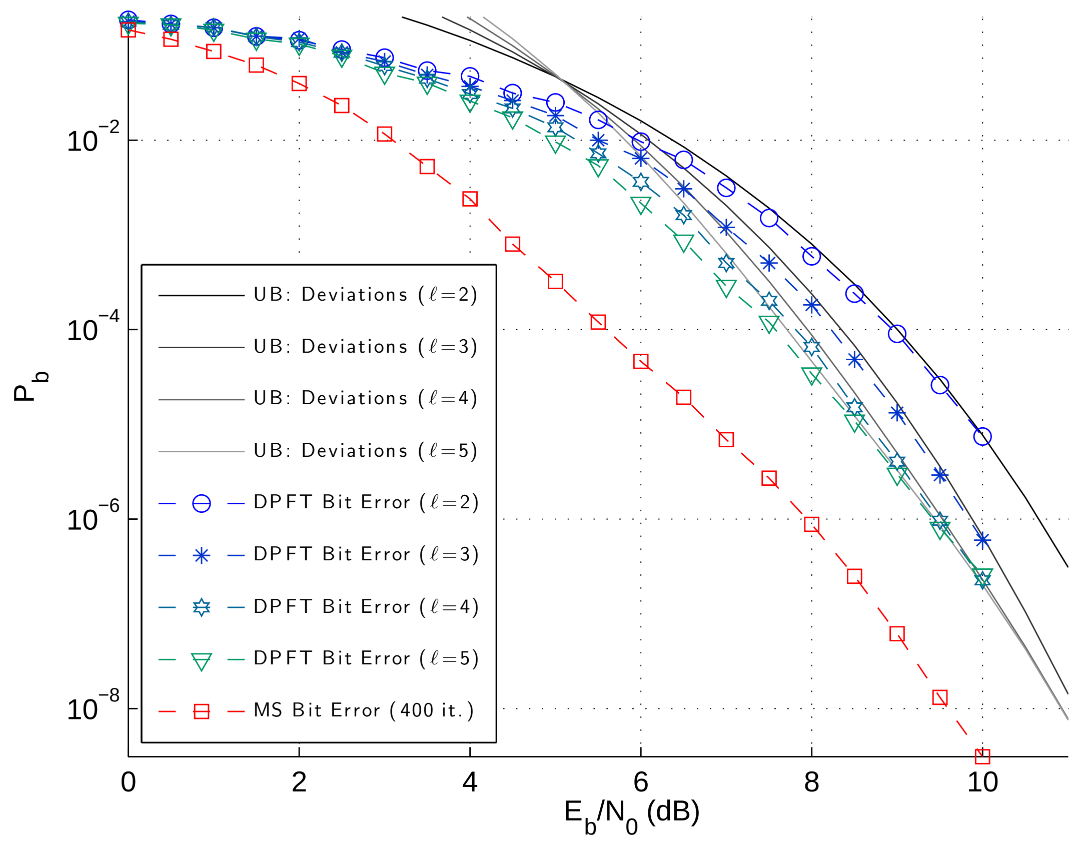


Figure 3.20: Probability of bit error for a (3,5)-regular (50,20) LDPC code decoded with the finite tree decoder on DPFTs with $\ell = 2, 3, 4$ and 5 and the MS decoder with 400 iterations.

in Figure 3.20 for a $(3, 5)$-regular low-density parity-check code with length $N = 50$ and dimension $K = 20$. Once again, lower bit error rates are achieved by increasing the maximum depth of the DPFT at SNRs between 5.0 dB and 11.0 dB. Leaf nodes are established after $\ell = 2$, resulting in weight 3.0 deviations. Compared to the results given in Figure 3.17 for finite tree decoding on extrinsic trees for the same length $N = 50$, dimension $K = 20$ LDPC code, the results are noticeably worse due to a drop in the weight of the minimum-weight deviation from 3.91 to 3.0.

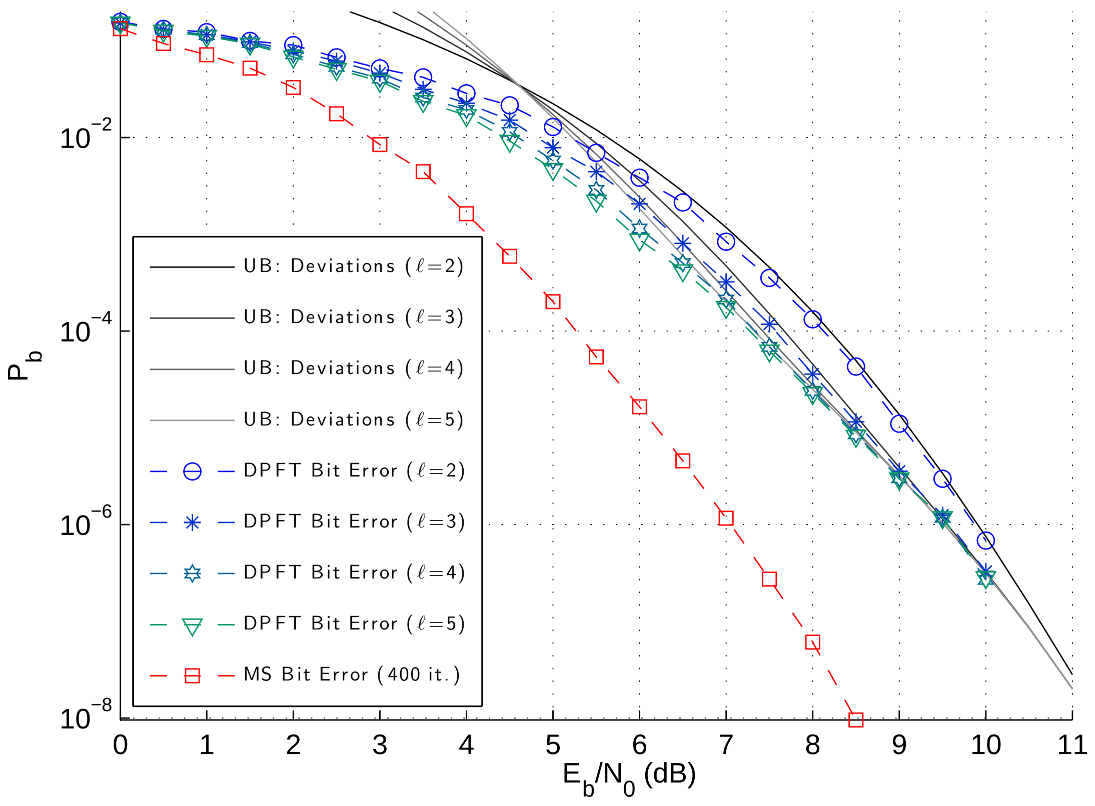


Figure 3.21: Probability of bit error for a irregular (40,20) LDPC code decoded with the finite tree decoder on DPFTs with $\ell = 2, 3, 4$ and 5 and the MS decoder with 400 iterations.

Finally, simulations of finite tree decoding on deviation path-forcing trees with maximum depths of $\ell = 2, 3, 4$ and 5 are given in Figure 3.21 along with min-sum decoding with 400 iterations on an irregular low-density parity-check code with length $N = 40$ and dimension $K = 20$. Similar improvements are demonstrated at SNRs between 4.5 dB and 11.0 dB. During the DPFT construction, leaf nodes are established after depth $\ell = 1$, thus resulting in deviations with weight equal to 2.0. The performance of finite tree decoding of this irregular code is much worse than MS decoding performance and worse than the performance of finite tree decoding on extrinsic trees, as shown in Figure 3.18.

In Figure 3.19, the performance appears to be promising for finite tree decoding on deviation path-forcing trees. However, the method for constructing the DPFTs makes them computationally intractable for large codes with large minimum distance. Consider the number of deviations in a DPFT of depth ℓ, where the weight of a minimum-weight deviation on the DPFT is greater than or equal to ℓ. The number of deviations, and correspondingly the number of leaf nodes, on the DPFT can be derived from equation (2.11), and is given by $(d_F - 1)^\ell$. For a typical $(3, 6)$-regular LDPC code, the number of deviations and leaf nodes is given in Table 3.7. Since each node in the DPFTs must be stored in memory, the algorithm requires vast amounts of memory when the depth of the DPFTs, and correspondingly, the minimum-weight of the deviations increases.

DPFT Depth (ℓ)	Deviations/Leaf Nodes
0	1
1	5
2	25
3	125
4	625
5	3125
10	$9,765,625$
15	$30,517,578,125$

Table 3.7: Number of deviations/leaf nodes on the DPFT of a $(3,6)$-regular LDPC code with varying minimum depths, when the minimum-weight deviation has weight greater than or equal to ℓ.

3.6 Finite Tree-Based Decoder Bounds

3.6.1 Deviation Bounds

The multiplicity of the deviations, given by equation (2.11), can be used to compute an upper bound on the probability of error of min-sum decoding after a finite number of iterations. Wiberg demonstrated that the probability of error at the root node, denoted P_e, can be bounded by

$$P_e \leq \bigcup_{\delta_i \in \Delta} P(B_i),$$

where Δ is the set of all deviations and B_i is the event that deviation δ_i has negative cost. This bound can be loosened to

$$P_e \leq \sum_{\delta_i \in \Delta} P(B_i). \tag{3.8}$$

using the union bound. In the computation tree of a regular LDPC code, all deviations have the same weight $w(\delta)$. So, (3.8) can be computed as

$$P_e \leq |\Delta| \int_0^\infty \frac{1}{\sqrt{2\pi(w(\delta)\sigma^2)}} e^{\frac{-(y+w(\delta))^2}{2(w(\delta)\sigma^2)}} dy. \tag{3.9}$$

Figure 3.22 shows the upper bound on the probability of error at the root node for various rate $\frac{K}{N} = \frac{1}{2}$ codes after two iterations. Because the summation in (3.8) is used to obtain the bound of (3.9) instead of the true union, the accuracy of the upper bound suffers at lower SNR. In order to tighten the bound, the overlapping regions between the events in the summation should be removed. In the bound of (3.9), the probability $P(B_i)$ that a deviation δ_i has cost less than zero is added to the probability $P(B_j)$ that a deviation δ_j has cost less than zero. However, these two events may have an intersecting probability $P(B_i \cap B_j)$ that should be subtracted from the summation in order to achieve a tighter bound. The probability of this intersection can be rewritten as

$$P(B_i \cap B_j) = P(B_i)P(B_j|B_i).$$

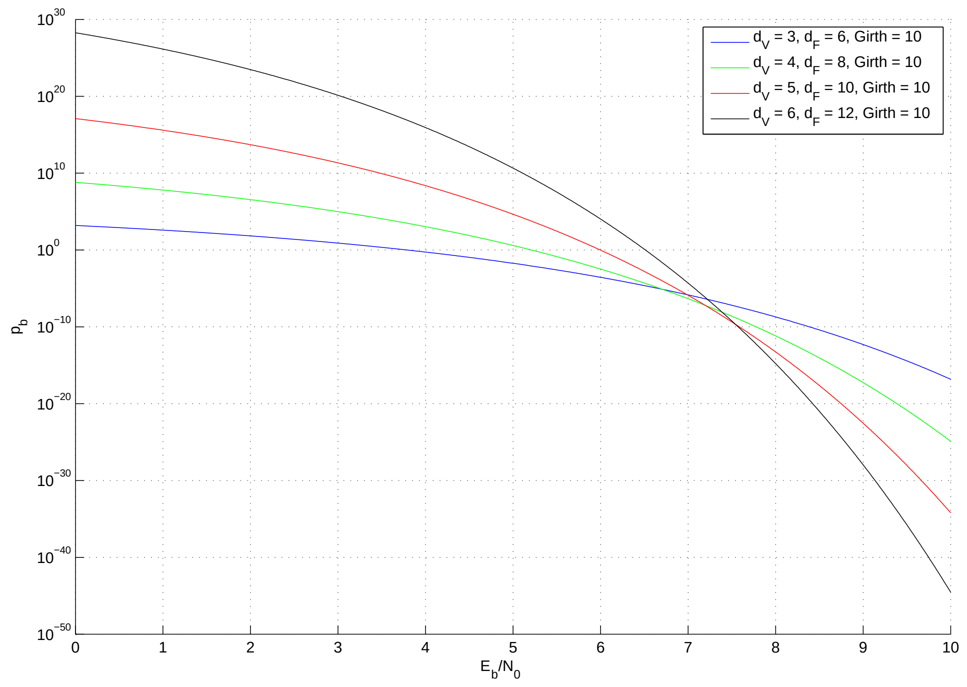


Figure 3.22: Deviation bound on various rate 1/2 codes after two iterations.

It is easy to compute $P(B_i)$ numerically using only the weight $w(\delta_i)$ of the deviation δ_i. However, the conditional probability $P(B_j|B_i)$ is not as easy to compute, since the deviations δ_i and δ_j share only a fraction of their nodes. The set of shared nodes are denoted by $\delta_{i\cap j}$. The shared variable nodes in $\delta_{i\cap j}$ have a cost distribution of $\mathcal{N}(w(\delta_{i\cap j}), w(\delta_{i\cap j})\sigma^2)$, while the variable nodes that are not shared, denoted $\delta_{i-i\cap j}$ and $\delta_{j-i\cap j}$, both have a cost distribution of $\mathcal{N}(\delta_{i-i\cap j}, \sigma^2)$.

Since the integral of the normal distribution is often calculated numerically, the value of $P(B_i \cap B_j)$ is also calculated numerically. The first step in the numerical calculation of $P(B_i \cap B_j)$ is breaking up the intersecting region into the summation

$$P(B_i \cap B_j) = \sum_{k=-\infty}^{0} P(G(\delta_i) = k) \sum_{l=-\infty}^{\infty} P\left(G(\delta_{i\cap j}) = \frac{k}{2} - l\right) P\left(G(\delta_{i-i\cap j}) = \frac{k}{2} + l\right) P\left(G(\delta_{j-i\cap j}) + \frac{k}{2} - l < 0\right) \tag{3.10}$$

where the first summation considers each individual value of the cost $G(\delta_i)$ in the region B_i. The second summation in (3.10) determines the probability of each combination of costs $G(\delta_{i\cap j})$ and $G(\delta_{i-i\cap j})$ that add up to k, and then multiplies this by the probability that $G(\delta_{j-i\cap j}) + G(\delta_{i\cap j})$ still results in a negative cost.

Equation (3.10) gives an expression for the probability of intersecting regions between two deviations. It is now necessary to find the number and weight of the intersecting regions between the deviations on a given tree. A deviation δ_i of weight $w(\delta_i)$ on the computation tree includes many leaf nodes. If leaf node v_i has $d_F - 2$ neighboring leaf nodes, then there are precisely $d_F - 2$ other deviations on the computation tree that include $w(\delta_i) - 1$ of the same variable nodes in δ_i, excluding variable node v_i. Instead, the other deviations include one of v_i's neighboring leaf nodes. This situation is true for every leaf node in deviation δ_i, of which there are $d_V(d_V - 1)^{\ell-1}$ after ℓ iterations. Therefore, there are a total of $(d_F - 2)d_V(d_V - 1)^{\ell-1}$ deviations that share all but one variable node with δ_i. This is true for every deviation on the tree. Therefore, there is a total of

$$(d_F - 1)^{\sum_{i=1}^{\ell} d_V(d_V-1)^{i-1}} \left(\frac{(d_F - 2)d_V(d_V - 1)^{\ell-1}}{2}\right)$$

$$= |\Delta| \binom{(d_F - 2)d_V(d_V - 1)^{\ell-1}}{2},$$

unordered pairs of deviations that only differ by one variable node. If the intersecting regions of each pair of deviations that only differ by one variable node are subtracted from the bound of (3.9), the result is

$$|\Delta| \int_0^\infty \frac{1}{\sqrt{2\pi(w(\delta)\sigma^2)}} e^{\frac{-(y+w(\delta))^2}{2(w(\delta)\sigma^2)}} dy - P(B_i \cap B_j)|\Delta| \binom{(d_F - 2)d_V(d_V - 1)^{\ell-1}}{2}, \tag{3.11}$$

where event B_i and event B_j represent deviations of weight $w(\delta)$ that only differ by one variable node.

The expression given in (3.11) is not an upper bound on the probability of error at the root node, because intersecting events where deviations differ by more than one variable node are not subtracted. This expression is also not a lower bound obtained by subtracting the intersection of all pairs of deviations, as would be generated using the Bonferroni inequalities. Instead, (3.11) subtracts some intersecting events, but it also includes overlap between triplets of events that needs to be added back into the expression in order to make it an upper bound. In order to find all the three-way intersections that need to be added back to the expression, consider all the two-way intersections that deviation δ_i is involved in. The deviation δ_i is involved in $(d_F - 2)d_V(d_V - 1)^{\ell-1}$ intersections with other deviations that gets subtracted using (3.11). Thus, there are $\binom{(d_F-2)d_V(d_V-1)^{\ell-1}}{2}$ three-way intersections for each deviation in Δ that need to be added back to the expression in (3.11) to make it a true upper bound.

In each of these intersections, the deviation δ_i and two other deviations are considered. Deviation δ_i differs from the other two deviations by only one variable node, while the other two deviations differ by exactly two variable nodes. Therefore, the intersection between these two variable nodes is not subtracted in (3.11), and does not need to be added back into the expression. The probability of events B_i, B_j, and B_k occurring at the same time is given by

$$P(B_i \cap B_j \cap B_k) = \sum_{k=-\infty}^{0} P(G(\delta_i) = k) \sum_{l=-\infty}^{\infty} P\left(G(\delta_{i\cap j}) = \frac{k}{2} - l\right) P\left(G(\delta_{i-i\cap j}) = \frac{k}{2} + l\right) P\left(G(\delta_{j-i\cap j}) + \frac{k}{2} - l < 0\right) \tag{3.12}$$

The resulting upper bound computed after subtracting intersecting regions of pairs of deviations, and adding back intersecting regions of triplets of deviations, is given by

$$\begin{aligned} P_e \quad &\leq \quad |\Delta| \int_0^\infty \frac{1}{\sqrt{2\pi(w(\delta)\sigma^2)}} e^{\frac{-(y+w(\delta))^2}{2(w(\delta)\sigma^2)}} dy - P(B_i \cap B_j)|\Delta| \left(\frac{(d_F-2)d_V(d_V-1)^{\ell-1}}{2}\right) \\ &+ \quad P(B_i \cap B_j \cap B_k)|\Delta| \left(\frac{(d_F-2)d_V(d_V-1)^{\ell-1}}{3}\right). \end{aligned}$$

Unfortunately, the complexity of upper bound makes it difficult to compute. Additionally, the upper bound makes the assumption that the finite tree is a computation tree with only one copy of each variable node. Under the same assumption, Section 3.6.2 demonstrates how precise analytical results can be obtained using density evolution.

3.6.2 Density Evolution on Independent Trees

Density evolution was first introduced as an analytical technique for analyzing sum-product decoding of low-density parity-check codes [17] [18]. Shortly afterwards, a technique was developed for density evolution of MS decoding of LDPC codes [33]. This section examines density evolution applied to MS decoding and shows how this technique can be used to obtain exact analytical results for finite tree-based decoding when independent trees are built using Algorithm 3.2.1.

In order to visualize the behavior of the min-sum decoder, messages are interpreted as being passed from the leaf nodes up the computation tree to the root node. The message computed at the lowest level of check nodes is

$$m_{f_i \to v_j} = \left(\prod_{v_k \in N(f_i) \setminus v_j} \operatorname{sgn}(-\frac{2}{\sigma^2} y_k) \right) \left(\min_{v_k \in N(f_i) \setminus v_j} |-\frac{2}{\sigma^2} y_k| \right),$$

which can be simplified to

$$m_{f_i \to v_j} = \left(\prod_{v_k \in N(f_i) \setminus v_j} \operatorname{sgn}(-y_k) \right) \left(\min_{v_k \in N(f_i) \setminus v_j} |y_k| \right),$$

since only minimizations and summations are used by the MS decoder. The following two assumptions make density evolution possible. The first is that the all-zeros codeword was transmitted, and the second is that there is at most one copy of each variable node on the computation tree. The second condition is known to be true for independent trees.

The cumulative distribution function (CDF) of each codeword coordinate $-y_i$, $1 \leq i \leq N$, received from the channel is Gaussian with mean $\mu = +1$ and variance $\sigma^2 = \frac{K \times 10^{\frac{\text{SNR}}{10}}}{2N}$. To be consistent with density evolution of the SP decoder discussed in Section 2.10, the probability density function (PDF) corresponding to the CDF of the channel output is denoted as $\mathcal{P}_0$. Assuming that the LDPC code is (d_V, d_F)-regular, then the CDF of $m_{f_i \to v_j}$ is computed as follows.

To calculate the cumulative distribution function of $m_{f_i \to v_j}$, the positive and negative portions are split into two regions. First, consider the probability $P(m_{f_i \to v_j} \leq \tau)$, when $\tau < 0$. The cost from each leaf node $v_k \in N(f_i) \backslash v_j$ must have absolute value $|y_k| \geq |\tau|$. There must also be an odd number of $-y_k < 0$. Given that $\tau < 0$, the CDF of $m_{f_i \to v_j}$ is computed from the probability

$$P(m_{f_i \to v_j} \leq \tau) = \sum_{\{\text{odd } \alpha \,|\, 1 \leq \alpha \leq (d_F - 1)\}} \binom{d_F - 1}{\alpha} P(-y_k \leq \tau)^{\alpha} P(-y_k \geq -\tau)^{d_F - 1 - \alpha}. \tag{3.13}$$

Next, consider the case when $\tau \geq 0$. In this case it is easier to first compute the probability $P(m_{f_i \to v_j} \geq \tau)$. To satisfy the condition that $m_{f_i \to v_j} \leq \tau$, there must be an even number (including zero) of $-y_k < 0$. Given that $\tau \geq 0$, the CDF of $m_{f_i \to v_j}$

is computed from the probability

$$P(m_{f_i \to v_j} \leq \tau) = 1 - \left(\sum_{\{\text{even } \alpha \,|\, 0 \leq \alpha \leq (d_F - 1)\}} \binom{d_F - 1}{\alpha} P(-y_k \leq -\tau)^{\alpha} P(-y_k \geq \tau)^{d_F - 1 - \alpha} \right). \tag{3.14}$$

Equations (3.13) and (3.14) are used to compute $P(m_{f_i \to v_j} \leq \tau)$ for each $\tau \in \mathbb{R}$, thus resulting in the CDF of $m_{f_i \to v_j}$. Given the CDF of $m_{f_i \to v_j}$, the PDF of $m_{f_i \to v_j}$ is obtained by taking the derivative of the CDF. The PDF of $m_{f_i \to v_j}$ is denoted $\mathcal{Q}_0$.

The next message sent up the computation tree, from check nodes to variable nodes, by the min-sum decoder is

$$m_{v_i \to f_j} = -y_i + \sum_{k \in N(v_i) \backslash j} m_{f_k \to v_i}.$$

The PDF of $m_{v_i \to f_j}$ is given by

$$\mathcal{P}_1 = \mathcal{P}_0 \star \mathcal{Q}_0^{\star(d_V - 1)},$$

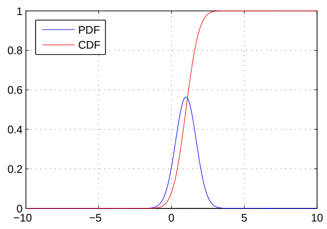
where $\star$ denotes a convolution. For any $1 \leq i < \ell$, where ℓ is the number of iterations, $\mathcal{Q}_i$ can be computed from equations (3.13) and (3.14) after substituting the CDF of $-y_i$ with the CDF corresponding to $\mathcal{P}_i$. For any $i \leq (\ell - 1)$, the PDF $\mathcal{P}_i$ can be computed using

$$\mathcal{P}_i = \mathcal{P}_0 \star \mathcal{Q}_{i-1}^{\star(d_V - 1)}.$$

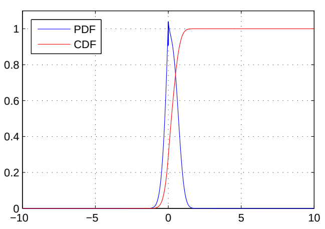
The PDF of $\mathcal{P}_\ell$, the final PDF of the output of the MS decoder, can be computed using

$$\mathcal{P}_\ell = \mathcal{P}_0 \star \mathcal{Q}_{\ell-1}^{\star(d_V)}.$$

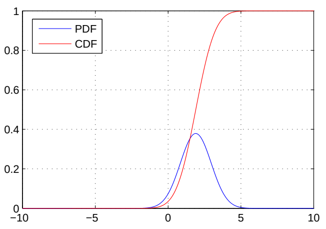
The exact probability of error at the root node can be found by integrating $\mathcal{P}_\ell$ over all negative costs.



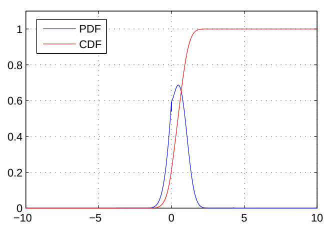
(a) $\mathcal{P}_0$



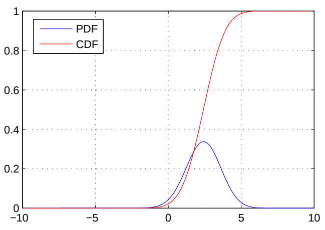
(b) $\mathcal{Q}_0$



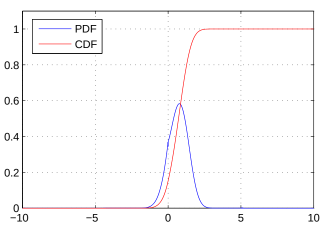
(c) $\mathcal{P}_1$



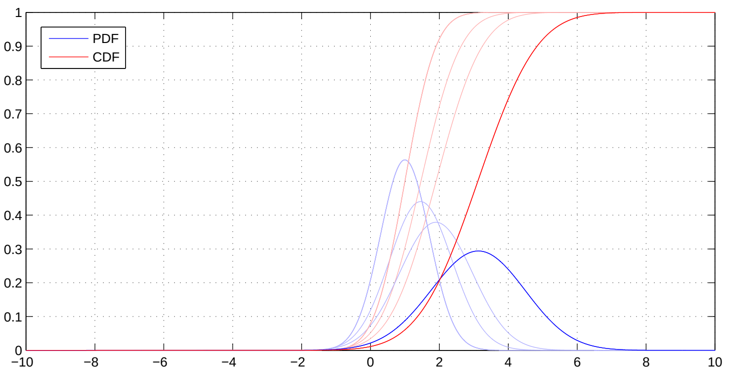
(d) $\mathcal{Q}_1$



(e) $\mathcal{P}_2$



(f) $\mathcal{Q}_2$



(g) PDF and CDF of the root node: $\mathcal{P}_3$

Figure 3.23: PDF and CDF of the messages passed during min-sum decoding of a $(3, 6)$-regular LDPC code at SNR $=$ 3dB for three iterations.

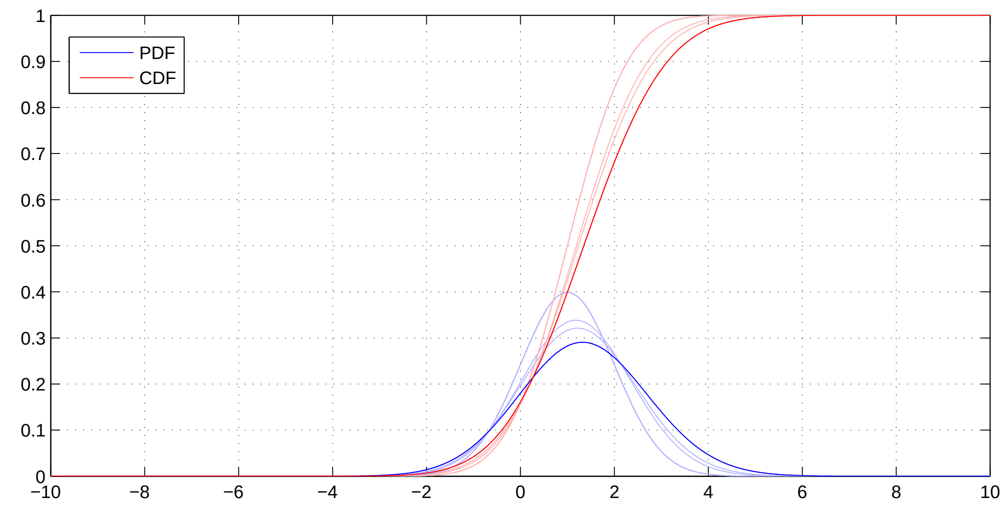


Figure 3.24: PDF and CDF of the messages passed during min-sum of a $(3, 6)$-regular LDPC code at SNR $= 0$dB for three iterations.

Example 3.6.1. *In this example, decoding of* $(3, 6)$*-regular low-density parity-check codes with min-sum is examined when the all-zeros codeword is transmitted over an additive white Gaussian noise channel with SNR $= 3$dB. Figure 3.23 shows the PDFs and CDFs of the messages passed by the MS decoder from the leaf nodes up to the root node during all three iterations. The full progression of the PDF's and CDF's is shown in Figure 3.23(g). At an SNR $= 3$dB, more iterations cause the PDF and CDF of $\mathcal{P}$ to move to the right and result in a decreased probability of error at the root node.*

While additional iterations decreased the probability of error at SNR $= 3$dB, this *is not always the case. Figure 3.24 shows the results of density evolution on the same code at SNR = 0dB. Here, the probability of error actually increases as more iterations are performed. After one iteration the probability of error is* $P_b = 0.1092$*. After three iterations the probability of error is* $P_b = 0.1117$*.*

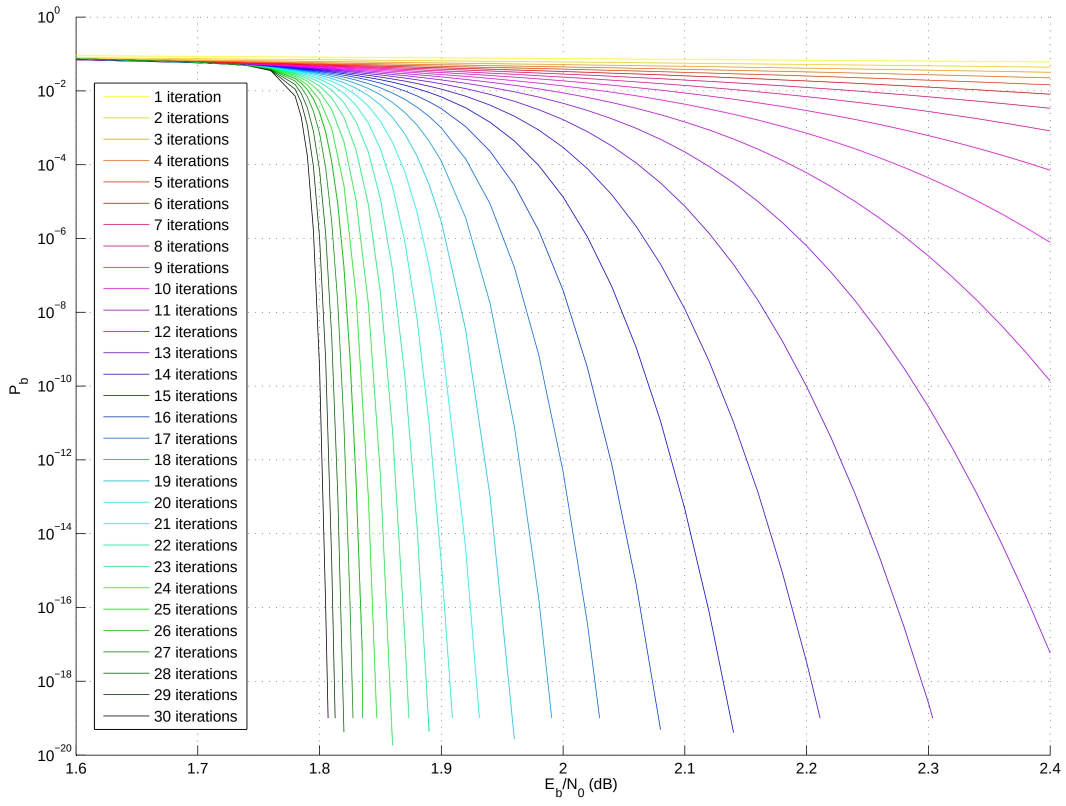


Figure 3.25: Density evolution results for $(3, 6)$-regular LDPC codes decoded with the min-sum decoder.

Figure 3.25 shows the probability of error after each iteration up to 30 for the *min-sum decoder operating on a* $(3,6)$*-regular low-density parity-check code. Each new iteration decreases the probability of error when the SNR is at least* $1.77dB$*. The capacity of MS decoding can be computed by performing density evolution for a large number of iterations while tracking the probability of bit error curves to see if the probability increases or decreases with increasing iterations. For* $(3,6)$*-regular LDPC codes the capacity of MS decoding is approximately* $1.77dB$*.*

When independent trees are constructed using Algorithm 3.2.1, density evolution can be used to compute the exact probability of bit error at the root node. This is because each node is only allowed to appear once in the independent tree. Therefore, all the messages passed up from the leaf nodes to the root nodes are independent. Figure 3.26 shows the independent tree rooted at v_0 for a rate $\frac{1}{2}$, length $N = 200$ LDPC code. In this example, 143 of the 200 variable nodes appear in the independent tree. Using this independent tree, simulation results of finite tree-based decoding are given along with the results of density evolution in Figure 3.27. The analytical results of density evolution accurately predict the performance of finite tree-based decoding on the independent tree for all simulated SNR. One advantage of density evolution, as seen in this example, is its ability to obtain the exact probability of bit error far beyond the reach of simulation. In Figure 3.27, a probability of bit error of nearly 10^{-15} is predicted for SNR = 11.0 dB.

It has been shown that density evolution can be used to predict the performance of

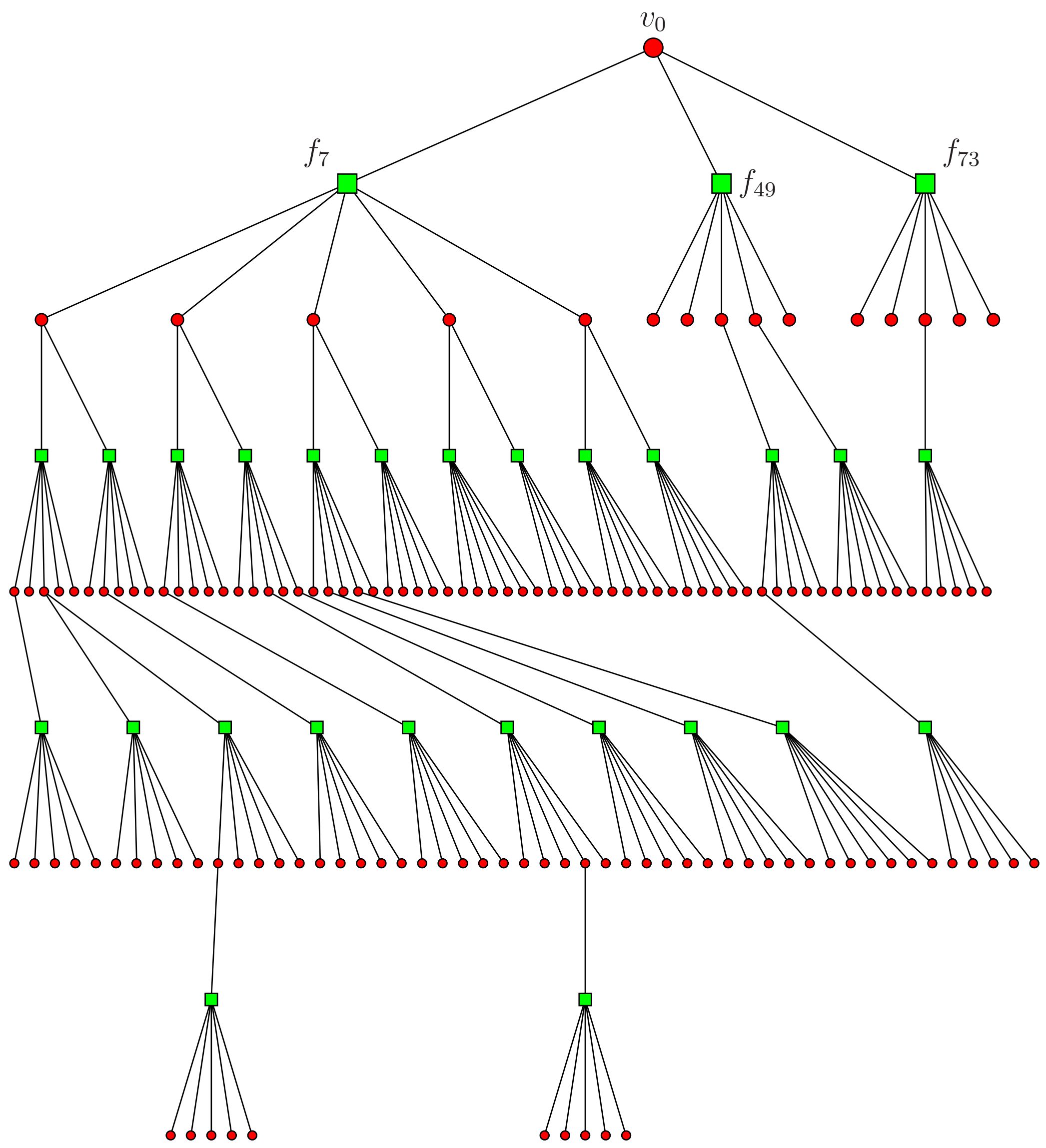


Figure 3.26: Finite tree-based decoding of an independent tree rooted at v_0 of a rate $\frac{1}{2}$, length $N = 200$ LDPC code.

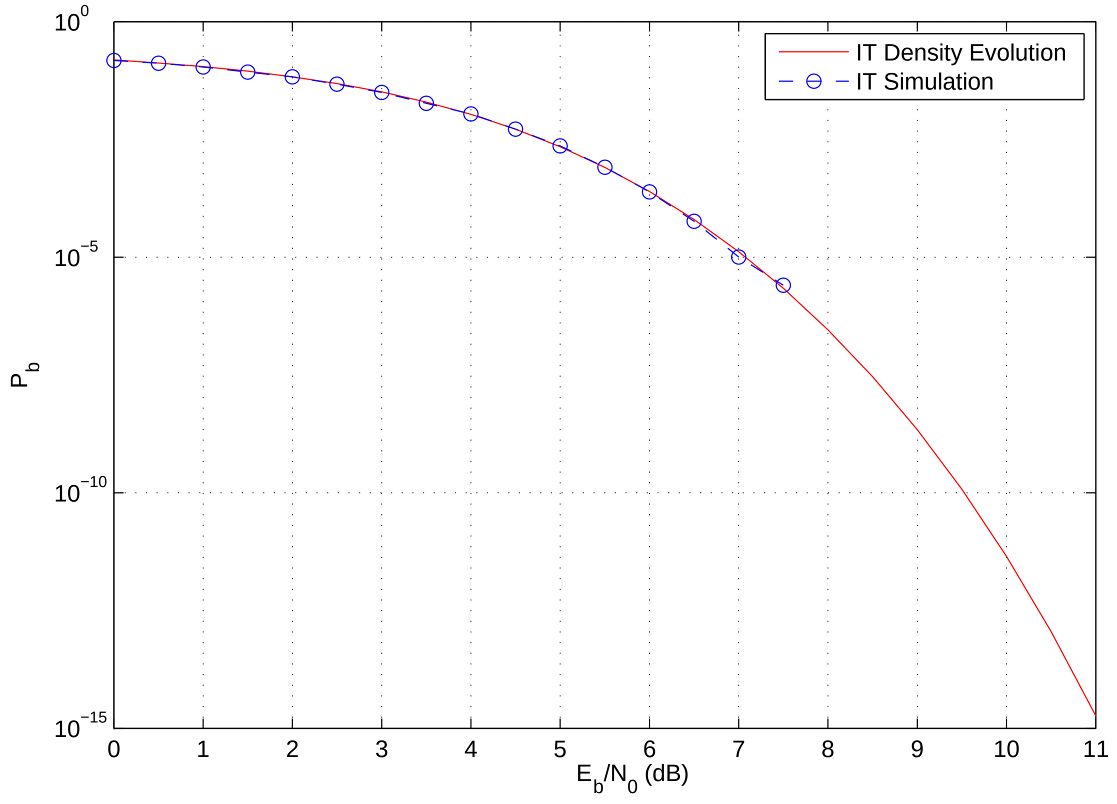


Figure 3.27: Density evolution and simulation results for finite tree-based decoding on the independent tree rooted at v_0 of a rate $\frac{1}{2}$, length $N = 200$ LDPC code.

finite tree decoding on independent trees. Unfortunately, as shown in Figure 3.13, the probability of bit error of finite tree decoding on independent trees is not competitive with MS decoding for codes up to length $N = 100{,}000$. Although independent trees may not be useful for decoding, it is shown in the next chapter that they can be used for creating bounds on the minimum distance and also as tools for constructing

LDPC codes with desirable properties.

3.7 Conclusion

In this chapter, finite tree decoding along with several new methods for constructing finite trees have been presented. Each of the finite tree construction methods presents a tradeoff between complexity and probability of bit error when decoded using finite tree decoding. Independent trees are very simple to construct, even for codes with block length as large as $N = 100{,}000$. However, they result in poor bit error rates when compared to MS decoding for much shorter block length.

Extrinsic tree construction enumerates each of the deviations on the extrinsic trees, and thus the complexity of construction grows exponentially with the number of variable nodes in the tree. For codes with block lengths $N \leq 50$, the bit error rates are comparable to, and sometimes exceed, those of MS decoding. Results of finite tree decoding on extrinsic trees shows that LDPC codes with smaller, regular node degrees appear to perform better than irregular LDPC codes.

Finally, deviation path-forcing tree construction produces trees that are very simple to analyze. This is because the weight of a deviation is equal to $\ell + 1$ if the lone leaf node in the deviation appears in level ℓ. The performance of finite tree decoding on DPFTs is in between that of independent trees and extrinsic tree in all the presented simulations. Unfortunately, the number of variable nodes in DPFTs grows exponentially with the minimum weight of its deviations. This makes DPFTs impractical for use with high minimum-distance LDPC codes.

After considering each of the finite tree construction methods, extrinsic tree construction appears to offer the greatest potential for decoding low-density parity-check codes as the block length increases beyond those presented in this chapter. This is because independent trees have unacceptable bit error rates compared to MS decoding, and DPFTs are not computationally tractable to construct for codes with high minimum distance.

The process of constructing extrinsic trees involves computing the set of all deviations and their corresponding deviation weights on each of the extrinsic trees. This makes it straightforward to compute Wiberg's deviation bound on the performance of finite tree decoding on the extrinsic trees. However, codes with typical variable and check node degrees, such as the $(3, 6)$-regular LDPC code, have a large number of deviations in the extrinsic trees of LDPC codes with block length $N = 40$. It is thus necessary to use LDPC codes of smaller variable and check node degree in order to allow for computationally tractable extrinsic tree construction for large block length codes.

Chapter 4 analyzes the properties of low-density parity-check codes with respect to variable and check node degrees, block length, minimum distance, and the complexity of extrinsic tree construction. It is shown that by carefully choosing the degrees of the variable nodes and check nodes, short-to-moderate block-length codes can be created that allow for finite tree decoding on extrinsic trees. A new method for constructing LDPC codes using independent trees is also introduced. Codes generated using this new construction offer performance advantages over those generated using an existing construction method.

Chapter 4

Code Design for Finite Tree-Based Decoding

Since the rediscovery of low-density parity-check codes, several different methods have been devised for constructing codes that perform well with iterative decoders. One property of LDPC codes considered to be important to the performance of iterative decoders is the girth [19]. The *girth* of an LDPC code, denoted by $\mathcal{G}$, is the length of the shortest cycle in the corresponding Tanner graph of the LDPC code. A large girth allows more iterations to be performed before the information propagated between nodes in the Tanner graph begins to reinforce itself. The phenomenon of self-reinforcement can be visualized as multiple copies of the same variable node appearing in a computation tree, thus making it possible for a valid configuration on the computation tree to have its LLR cost affected more by some variable nodes than others. One method of constructing LDPC codes, known as the progressive edge-growth (PEG) algorithm, takes a set of variable node degrees and check node degrees and attempts to maximize the girth of the resultant Tanner graph [19]. Modifications to the PEG algorithm have been developed to improve the probability of bit error of irregular LDPC codes at high SNR [34] [35].

This chapter begins by introducing the progressive edge-growth algorithm and examining properties of low-density parity-check codes in Section 4.1. In particular, the relationship between block length, minimum distance, and girth is examined in detail. Understanding the relationships between these different code parameters is necessary for constructing LDPC codes with desirable properties.

As shown in [19], the girth of the Tanner graph of a low-density parity-check code provides a lower bound on the minimum distance of the code. A new bound on minimum distance of LDPC codes, known as the independent tree-based (ITB) lower bound, is presented in Section 4.2. The ITB lower bound involves constructing independent trees rooted at each variable node in the LDPC code. The minimum-weight valid configuration on each independent tree can be easily determined, and serves as a lower bound on the minimum-weight codeword that each variable node is involved in. The set of lower bounds for each variable node easily extends to a lower bound on the minimum distance over the entire LDPC code. The ITB lower bound on minimum distance is proven to be greater than or equal to the lower bound on the minimum distance of LDPC codes derived from the girth of the code.

A new method is presented in Section 4.3 for constructing independent tree-based low-density parity-check (ITB LDPC) codes by iteratively improving the independent tree-based lower bound on the minimum distance. The new ITB LDPC codes have improved girth profiles and minimum distance bounds when compared to PEG LDPC codes. In addition, simulations indicate that the ITB LDPC codes have lower bit error rates at high SNR with iterative decoding methods.

In Section 4.4, the performance of finite tree decoding on extrinsic trees is examined on low-density parity-check codes with regular check node degree $d_F = 3$. Analysis shows that ITB LDPC codes with check node degree $d_F = 3$ have better minimum-distance properties than PEG LDPC codes with check node degree $d_F = 3$, thus resulting in lower bit error rates at high SNR. Examples are also given where finite tree decoding of extrinsic trees with $d_F = 3$ have bit error rates that approach ML decoding as the SNR grows large.

4.1 Code Construction and Code Properties from Computation Trees

One of the most effective methods for constructing low-density parity-check codes is the progressive edge-growth algorithm. The goal of PEG LDPC code construction is to create codes that have large girth [19]. Not only does large girth result in desirable minimum-distance properties, it also benefits iterative decoding by increasing the minimum weight of the deviations in the computation tree after a small number of iterations has been performed. The relationship between girth and minimum distance is examined in detail later in this section.

The procedure for constructing progressive edge-growth low-density parity-check codes involves sequentially adding edges to the Tanner graph for each of the variable nodes. The PEG construction begins with the block length N, the number of parity checks M, and the set of variable node degrees $\{d_{v_1}, d_{v_2}, \ldots, d_{v_N}\}$. For each candidate edge connecting a variable node and a check node, the cycle that the edge induces on the Tanner graph is chosen to have maximum length. The method for constructing PEG LDPC codes is given in Algorithm 4.1.1. Note that $\overline{R}^{\ell}_{v_i}(F)$ and $\overline{R}^{\ell}_{v_i}(V)$ refer to the check nodes and variable nodes that do not appear in the computation tree $R^{\ell}_{v_i}$.

Algorithm 4.1.1 (Progressive Edge-Growth [19]).

```
for $i = 1, \dots, N$
    for $j = 1, \dots, d_{v_i}$
        if $j = 1$
         –Add an edge connecting $v_i$ to a random check node
           $c_k \in F$ such that $d_{c_k} = \min_{c \in F} d_c$. Set $d_{c_k} = d_{c_k} + 1$.
        else
         –Expand the computation tree from variable node $v_i$ down to depth $\ell$, such
           that $\overline{R}^{\ell}_{v_i}(F) \neq \emptyset$ and $\overline{R}^{\ell+1}_{v_i}(F) = \emptyset$, or $|R^{\ell}_{v_i}(F)| = |R^{\ell+1}_{v_i}(F)| < M$.
         –Add an edge connecting $v_i$ to a random check node $c_k \in R^{\ell}_{v_i}(F)$,
           such that $d_{c_k} = \min_{c_k \in R^{\ell}_{v_i}(F)} d_{c_k}$. Set $d_{c_k} = d_{c_k} + 1$.
    end
end
```

It should be noted that the code construction given in Algorithm 4.1.1 does not guarantee that each check node has degree d_F. However, if check node regularity is desired, a minor modification to the construction can force the check node degrees to be regular. If, at level ℓ, the minimum check node degree is greater than d_F, set $\ell = \ell - 1$. Then randomly select a check node $c_k \in \overline{R}^{\ell}_{v_i}(F)$ with degree $d_{c_k} = \min_{c_k \in \overline{R}^{\ell}_{v_i}(F)} d_{c_k}$.

This process of reducing ℓ may be done recursively until $\min_{c_k \in \overline{R}^{\ell}_{v_i}(F)} d_{c_k} \leq d_F$. It is shown in Section 4.3 that this modification has very little effect on the girth and performance of PEG LDPC codes.

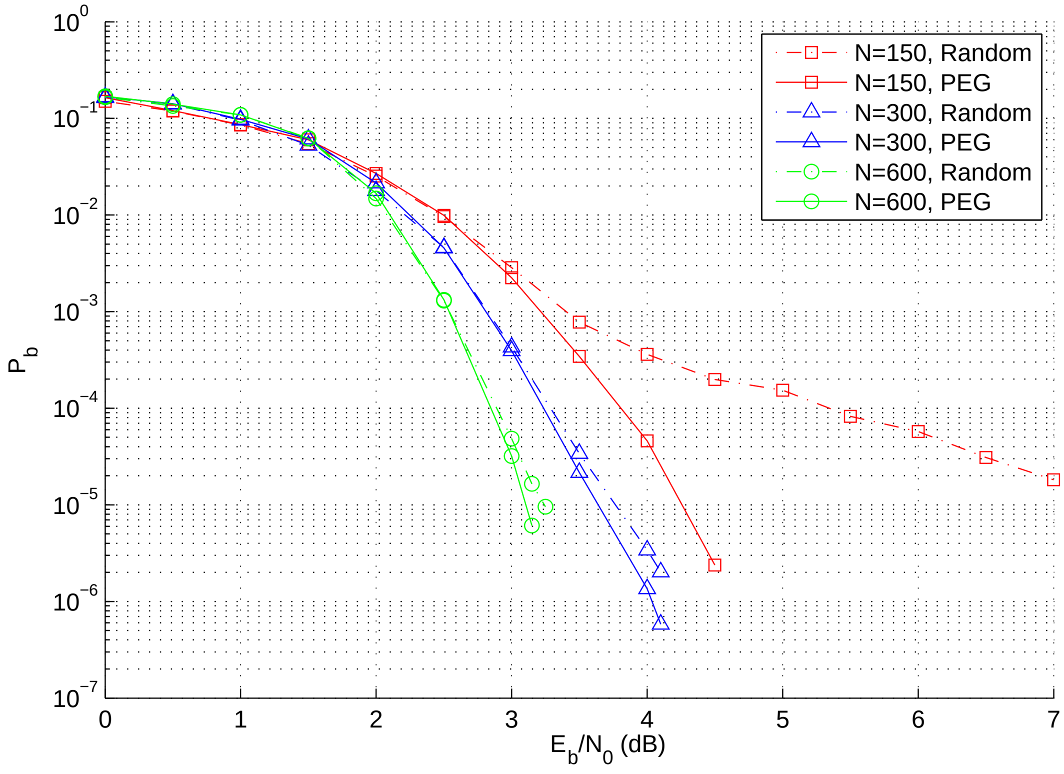


Figure 4.1: Probability of bit error for MS decoding of (3,6)-regular LDPC codes constructed randomly and with the PEG algorithm.

The benefits of progressive edge-growth are most noticeable at short-to-moderate block lengths ($N \leq 1500$), since randomly constructed low-density parity-check codes are capable of achieving near-capacity performance as the block length gets large.

Figure 4.1 shows the performance of several LDPC codes, generated both randomly and using the PEG algorithm, with MS decoding. The improvement in the probability of bit error is most pronounced at block length $N = 150$, and less significant at block lengths $N = 300$ and $N = 600$. In addition, the advantage of using PEG LDPC codes over random LDPC codes is most noticeable at low bit error rates.

In order to understand the reason for the performance gains, it is necessary to explore the effects of large girth on LDPC codes. An examination of the relationship between girth and block length shows that, after fixing the rate of the code and the variable node and check node degrees, longer LDPC codes have potential for larger girth than shorter LDPC codes. Furthermore, girth has a known relationship with the minimum distance of LDPC codes. This section provides a detailed examination of the relationship between girth, block length, and minimum distance for LDPC codes with regular variable node degrees and check node degrees.

Large girth in the Tanner graph of low-density parity-check codes results in lower bit error rates during the first few iterations of min-sum and sum-product decoding. This is because large girth allows the computation trees to have more levels below the root node, and hence more iterations, before encountering the same variable node more than once. To understand how girth restricts multiple copies of a single variable node from appearing on a computation tree, first consider the set of all non-backtracking walks on a computation tree of depth ℓ. The maximum length of any non-backtracking walk in the set is 4ℓ; it starts at one of the leaf nodes of the tree, makes its way through the root node, and finishes at a different leaf node. If the girth of a Tanner graph is $\mathcal{G} > 4\ell$, it is impossible to have more than one copy of the same variable node on the computation tree after ℓ iterations, since any two nodes on the computation tree are connected via a non-backtracking walk. Gallager [5] used an idea similar to this to derive a lower bound on the required block length N necessary to achieve girth $\mathcal{G}$ for (d_V, d_F)-regular LDPC codes. Since girth determines the maximum number of iterations such that no variable node appears more than once on the computation tree, the number of variable nodes in that computation tree is the minimum block length necessary to achieve that girth.

Proposition 4.1.2 (Gallager, [5]). *Given a (d_V, d_F)-regular low-density parity-check code, a block length of*

$$N \geq 1 + d_V(d_F - 1)\left(\sum_{i=0}^{b-1}((d_V - 1)(d_F - 1))^i\right) \tag{4.1}$$

is required to achieve girth $\mathcal{G} = 4b + 2$, where b is an integer.

A modification of Proposition 4.1.2 is now given to take into account girths of length $\mathcal{G} = 4b$, where b is an integer.

Proposition 4.1.3 (Gallager, [5]). *Given a (d_V, d_F)-regular low-density parity-check code, a block length*

$$N \geq d_F\left(\sum_{i=0}^{b-1}((d_V - 1)(d_F - 1))^i\right) \tag{4.2}$$

is required to achieve girth $\mathcal{G} = 4b$, where b is an integer.

Figure 4.2 shows the lower bounds given by (4.1) and (4.2) on the required block length for a given girth for three different degree profiles for rate $\frac{K}{N} = \frac{1}{2}$ codes. The minimum required block length grows exponentially with the girth for all given variable node and check node degrees. Codes with higher variable node and check node degrees require longer block lengths to achieve the same girth as codes with lower node degrees. However, while girth is an important property of LDPC codes, it is its relationship to low bit error rates that is most important to the performance of iterative decoders. Therefore, it is not sufficient to limit the examination to the relationship between girth and block length.

Girth can also be used to lower bound the minimum distance of a low-density parity-check code. To do so, it is first necessary to examine deviations on the computation tree. Given a computation tree that contains at most one copy of each variable node, the weight of the deviations can be easily determined. Unlike the total number of variable nodes in the computation tree, the deviation weight is dependent only on the degrees of the variable nodes. It is possible to bound the deviation weights of irregular LDPC codes, but for LDPC codes with regular variable node degree, it is possible to derive exact expressions for deviation weights. The deviation weight on the computation trees of LDPC codes with regular variable node degree d_V is given by Proposition 4.1.5.

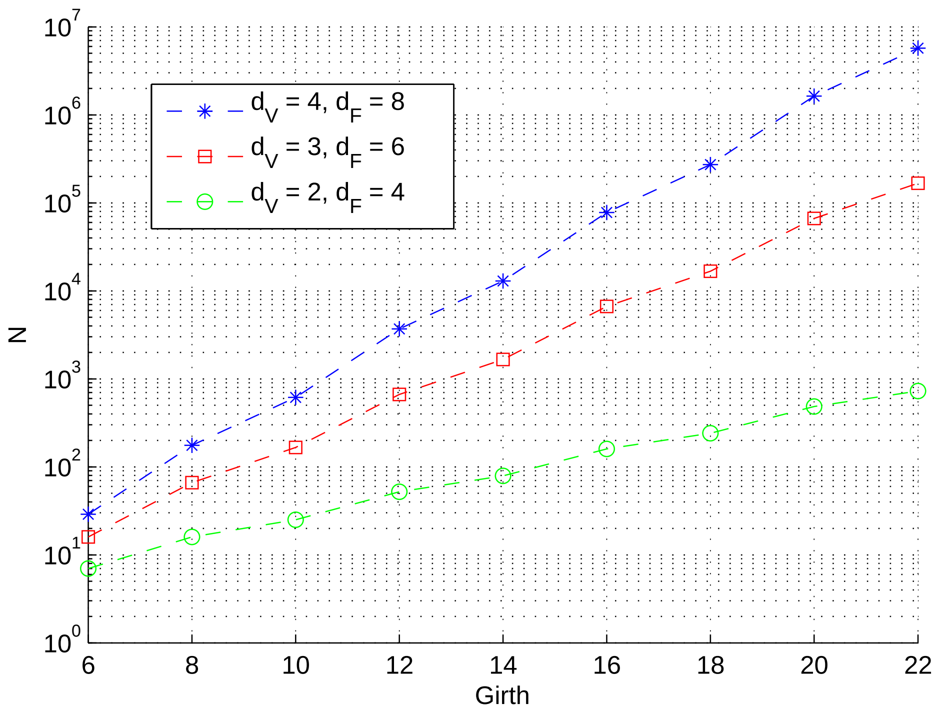


Figure 4.2: Lower bound on block length necessary to achieve a given girth for different variable and check node degrees.

Proposition 4.1.4 (Tanner, [12]). *After* $\ell \leq \frac{\mathcal{G}-2}{4}$ *iterations, the weight of each deviation* δ_ℓ *on the computation tree of a* (d_V, d_F)*-regular low-density parity-check code is*

$$w(\delta_\ell) = 1 + d_V \left(\sum_{i=1}^{\ell} (d_V - 1)^{i-1} \right). \tag{4.3}$$

The minimum-weight deviation on a computation tree rooted at variable node v_i serves as a lower bound on the minimum-weight codeword involving v_i. Thus, since girth can be used to determine the weight of the deviations when $\ell \leq \frac{\mathcal{G}-2}{4}$, girth also provides a lower bound on the minimum-weight codeword involving v_i. The set of minimum-weight codewords involving each of the variable nodes extends to a lower bound on the minimum distance of the code.

Proposition 4.1.5 (Tanner, [12])**.** *The minimum distance $d_{\min}$ of a (d_V, d_F)-regular low-density parity-check code satisfies*

$$d_{\min} \geq w(\delta_{\ell_{\max}}) \tag{4.4}$$

where $\ell_{\max} = \frac{\mathcal{G}-2}{4}$, and $\delta_{\ell_{\max}}$ is a deviation on the computation tree after $\ell_{\max}$ iterations.

A similar result is given to compute a lower bound on the minimum-weight codeword when $\frac{\mathcal{G}}{2}$ is even.

Proposition 4.1.6 (Tanner, [12])**.** *If $\frac{\mathcal{G}}{2}$ is even, then the minimum distance $d_{\min}$ of a (d_V, d_F)-regular low-density parity-check code satisfies*

$$d_{min} \geq 1 + d_V \left(\sum_{i=1}^{\lfloor \frac{\mathcal{G}-2}{4} \rfloor} (d_V - 1)^{i-1} \right) + (d_V - 1)^{\lfloor \frac{\mathcal{G}-2}{4} \rfloor}. \tag{4.5}$$

Figure 4.3 shows the lower bounds given by (4.4) and (4.5) on the minimum distance of low-density parity-check codes with variable node degrees of $d_V = 2$, 3, and 4. Codes with $d_V = 2$ are cycle codes, and have minimum distance lower bounds that grow linearly with girth. Alternatively, codes with $d_V = 3$ and 4 have minimum distance lower bounds that grow exponentially with girth. Comparing

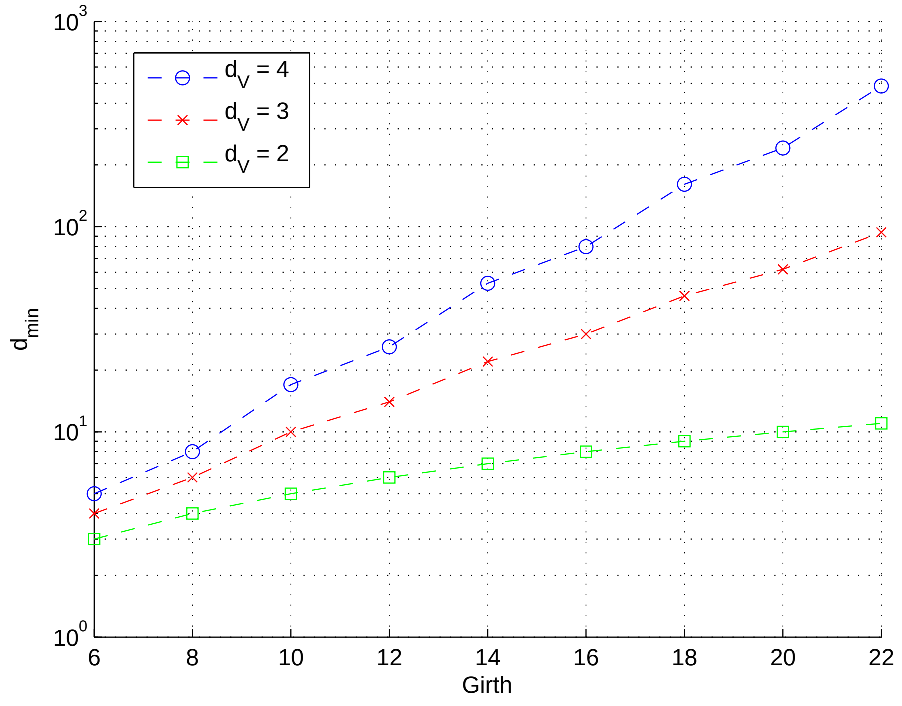


Figure 4.3: Lower bound on the minimum-weight codeword for a given girth for different variable node degrees.

Figure 4.2 and Figure 4.3, the relationships between girth, block length, and minimum distance can be further understood. For example, in order to guarantee a minimum distance of $d_{\min} = 100$ for a rate $\frac{K}{N} = \frac{1}{2}$, $(3, 6)$-regular LDPC code, the girth needs to be at least $\mathcal{G} = 22$. To achieve this girth requires a block length of at least $N = 200,000$. If a $(4, 8)$ regular LDPC code is used, it takes a girth of at least $\mathcal{G} = 18$ to guarantee $d_{\min} \geq 100$. This girth requires the block length of the code to be at least $N = 300,000$. These results indicate that codes of a given block length have similar minimum distance limitations regardless of their variable node degrees and check node degrees. Therefore, since the complexity of iterative decoders grows linearly with the degree of the variable/check nodes, LDPC codes with lower degrees are preferable to those with higher degrees.

In this section, it has been shown that girth can be used to bound the block length and minimum distance of low-density parity-check codes with specific variable node and check node degrees. However, it is possible to derive tighter bounds on the minimum distance of specific LDPC codes. In Section 4.2, a new method is presented for bounding the minimum distance of LDPC codes using the notion of independent trees introduced in Section 3.2.

4.2 Independent Tree-Based Lower Bounds on the Minimum Distance

A new method for lower bounding the minimum distance of low-density parity-check codes is presented in this section. In contrast to the bounds presented in Section 4.1, independent trees are used instead of computation trees. However, the method of bounding the minimum-weight codewords via deviations is still utilized.

The independent tree-based lower bounding method begins with the construction of independent trees rooted at each variable node in the code using Algorithm

3.2.1. To obtain a lower bound on the minimum-weight codeword that the root node of each independent tree is involved in, the minimum-weight deviation on the independent trees is found. As seen in Section 4.1 the minimum-weight deviation on a regular computation tree can be found from the girth of a (d_V, d_F)-regular LDPC code using the bounds of (4.4) and (4.5). However, independent trees constructed using Algorithm 3.2.1 have a more complex structure than computation trees, and the minimum-weight deviation can not be computed using simply the girth and node degrees. A computationally efficient way to determine the minimum-weight deviation on the independent tree is to assign a cost of $\lambda = +1.0$ to all the variable nodes, and then to use the finite tree decoder of Algorithm 3.1.2 to obtain an output at the root node. The resulting cost output at the root node will be the Hamming weight of the minimum-weight deviation on the independent tree, denoted $d_{\text{ITB-min}}$. In summary, there are four steps to obtaining the ITB lower bound on the minimum-weight codeword that each variable node is involved in.

1. Construct N independent trees using Algorithm 3.2.1.

2. Assign each variable node in each independent tree a cost of $\lambda = +1.0$.

3. Perform finite tree decoding using Algorithm 3.1.2.

4. For each independent tree, the decoder output at the root node is a lower bound $d_{\text{ITB-min}}$ on the minimum-weight codeword in which the root node is involved.

Proposition 4.2.1 establishes that the minimum-weight deviation on the independent tree has a Hamming weight that is greater than or equal to the minimum-weight deviation determined from the girth and node degrees of the LDPC code.

Proposition 4.2.1. *Given a parity-check matrix H with girth $\mathcal{G}$, the independent tree-based lower bound $d_{ITB\text{-}min}$ for any variable node v_i is greater than or equal to the lower bounds given by (4.4) and (4.5).*

Proof. Assume that the independent tree-based lower bound $d_{\text{ITB-min}}$ is lower than the bound obtained from (4.4). If $\frac{\mathcal{G}}{2}$ is odd, this implies that a variable node appears more than once in the independent tree during the construction of some level $\ell \leq \frac{\mathcal{G}-2}{4}$. Since the length of any path from one node to another in the independent tree after $\ell \leq \frac{\mathcal{G}-2}{4}$ levels is less than or equal to $\mathcal{G} - 2$, no variable nodes could have been eliminated, because they would appear on the independent tree for the first time. Thus, the minimum distance bound $d_{\text{ITB-min}}$ must be greater than or equal to that given by (4.4) when $\frac{\mathcal{G}}{2}$ is odd.

Now assume that the $\frac{\mathcal{G}}{2}$ is even, and $d_{\text{ITB-min}}$ is less than the bound obtained from (4.5). This implies that one node appears more than once before or during the consideration of the first $((d_F - 1)(d_V - 1))^{\lfloor \frac{\mathcal{G}-2}{4} \rfloor}$ check nodes of level $\ell = \lfloor \frac{\mathcal{G}-2}{4} \rfloor + 1$ in the independent tree. This requires that there is a path from a variable node to a copy of itself of length $\mathcal{G} - 2$ or less, which is not possible since that implies that the girth is less than $\mathcal{G}$. Thus, the minimum distance bound $d_{\text{ITB-min}}$ must also be greater than or equal to that given by (4.5) when $\frac{\mathcal{G}}{2}$ is even.

$\square$

Recall that progressive edge-growth low-density parity-check codes are constructed using girth as a metric for determining the connections between variable nodes and check nodes, and the bounds given by (4.4) and (4.5) show that improving girth can improve minimum distance. Proposition 4.2.1 shows that the ITB lower bound is as tight or tighter than the bounds given by (4.4) and (4.5). Therefore,it is reasonable to suggest that the ITB lower bounds can be used to create LDPC codes with improved minimum-distance properties when compared to PEG LDPC codes. In Section 4.3, the ITB lower bound is used as a metric for iteratively constructing LDPC codes with increasing lower bounds on the minimum distance.

4.3 Independent Tree-Based Low-Density Parity-Check Code Construction

This section presents a new method of code construction called the independent tree-based low-density parity-check (ITB LDPC) construction. Current methods for constructing LDPC codes, such as the PEG algorithm, build codes by creating a set of edges one-by-one and then keeping the edges fixed. A potential problem with using this approach is that edges created early on in the process are created for a different graph than edges created later in the process. For this reason, the ITB LDPC code construction begins with a randomly constructed LDPC code that has the desired number of variable nodes N, check nodes M, and variable and check node degrees. The ITB LDPC code construction then proceeds by shuffling edge connections between variable nodes and check nodes, while maintaining their respective degrees. The benefit of using this approach is that the effect of changing edge connections can be observed over the entire graph at each step.

Algorithm 4.3.1 presents independent tree-based low-density parity-check code construction. The construction makes use of the ITB lower bound presented in Section 4.2.

Algorithm 4.3.1 (ITB LDPC Code Construction). *Begin with a randomly constructed LDPC code that has the desired number of variable nodes N, check nodes M, and variable and check node degrees.*

```
while at least one edge has been switched
    for all pairs of edges $e_i, e_j \in E$ where the variable nodes connected to $e_j$ and
        $e_i$ are not connected to any of the same check nodes
      – Exchange the check nodes that edge $e_i$ and $e_j$ are connected to,
        keeping the variable node connections the same.
      – Using Algorithm 3.2.1, build $N$ independent trees and determine
        the minimum-weight deviation on each.
```

```
            if        the minimum-weight deviation of the set of all the
                      independent trees increases
                    – Keep the switch.
            else if   the minimum-weight deviation of the set of all the
                      independent trees is the same with decreased multiplicity
                    – Keep the switch.
            else if   the minimum-weight deviation of the independent trees is
                      the same with the same multiplicity, yet the mean weight
                      of the minimum-weight deviations has increased
                    – Keep the switch.
            else
                    – Undo the switch.
            end
        end
end
```

While this construction uses the independent tree construction given in Algorithm 3.2.1, any of the tree constructions presented in Chapter 3 could be used as long as it is possible to compute the minimum-weight deviation on each tree. The independent tree construction was chosen because it is simple to compute the minimum-weight deviation when there is a maximum of one copy of each variable node in the tree.

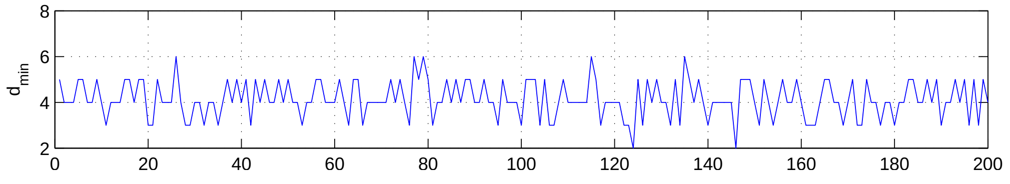
In practice, the independent tree-based low-density parity-check code construction has a much higher computational complexity than progressive edge-growth. For each set of edges that are switched, an independent tree rooted at each variable node is constructed, and the minimum-weight valid configuration on that independent tree is found. Assuming there are no 4-cycles, there are $d_V N$ edges that can be switched with $d_V(N - (1 + d_V(d_F - 1)))$ other edges. For each edge switch, N trees are built with a maximum of N nodes per tree. The computational complexity of the independent tree-based construction method scales linearly with the number of variable nodes in the trees, as does performing finite tree decoding on the independent trees to determine the weight of the minimum-weight valid configuration. The overall complexity of the independent tree-based code construction algorithm is $O(d_V^2 N^4)$. Note that parallel operations could reduce the complexity by a factor of N since it is not necessary for the independent trees to be constructed sequentially.

In the following simulation results, the sum-product decoder is used to decode all random, progressive edge-growth, and independent tree-based low-density parity-check codes. The SP decoder operates on real-valued (double floating point precision) channel outputs for a maximum of 80 iterations. The SP decoder with 80 iterations was chosen for this example to allow for direct comparisons with the PEG LDPC code results given in [19].

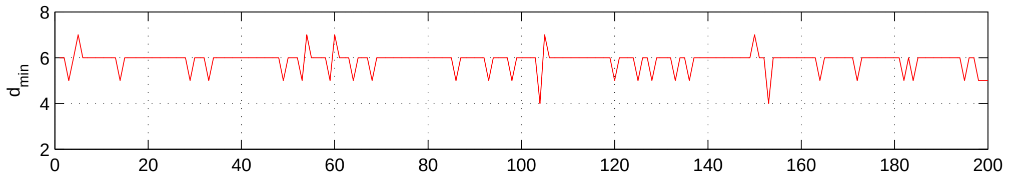
Block Length	Code Type	Min $\mathcal{G}$	Avg. $\mathcal{G}$	Min ITB Lower Bound	Avg. ITB Lower Bound
$N = 200$	Random	4	5.32	2	4.15
	PEG	6	7.79	4	5.88
	ITB	6	7.84	6	6.24
$N = 504$	Random	4	5.84	2	5.27
	PEG	8	8.00	6	7.50
	ITB	8	8.01	7	8.07
$N = 1008$	Random	4	6.77	3	6.69
	PEG	8	9.34	7	9.58
	ITB	8	9.96	9	9.98

Table 4.1: Girth and ITB lower bound statistics for $(3, 6)$-regular LDPC codes with block lengths of $N = 200$, $N = 504$, and $N = 1008$.

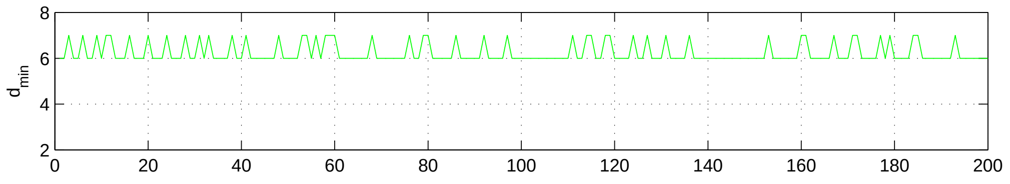
Table 4.1 shows both the girth and independent tree-based lower bound statistics for each variable node in random, progressive edge-growth, and independent tree-based $(3, 6)$-regular low-density parity-check codes with block lengths of $N = 200$, $N = 504$, and $N = 1008$. The minimum girth among all the variable nodes is given along with the average girth over all the variable nodes. Table 4.1 also shows the minimum ITB lower bound among all the variable nodes along with the average ITB lower bound over all the variable nodes.



(a) Random LDPC



(b) PEG LDPC



(c) ITB LDPC

Figure 4.4: Independent tree-based lower bound on the minimum distance for each bit of a random code, PEG LDPC code, and ITB LDPC code with $N = 200$, $M = 100$, $d_V = 3$, and $d_F = 6$.

Figure 4.4 gives a graphical representation of the independent tree-based minimum distance bound for each variable node of random, progressive edge-growth, and independent tree-based low-density parity-check codes with parameters $N = 200$, $M = 100$, $d_V = 3$, $d_F = 6$. The results of SP decoding of each type of LDPC code are shown in Figure 4.5. Simulations indicate that both PEG and ITB codes perform

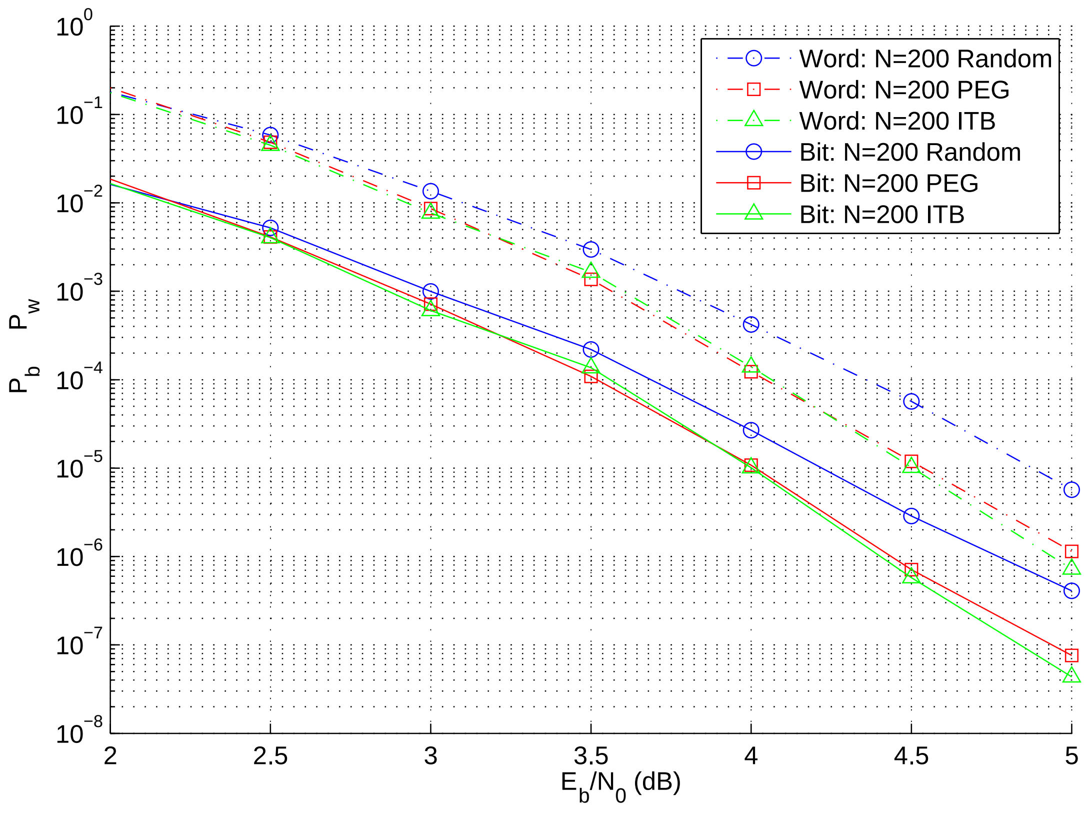
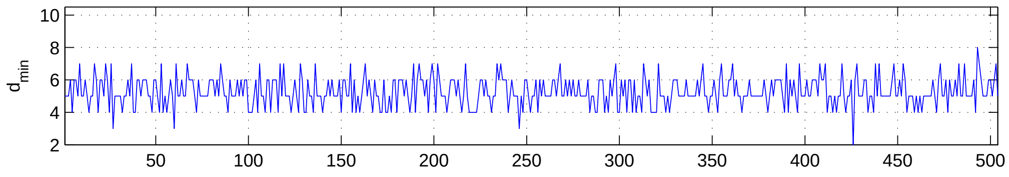


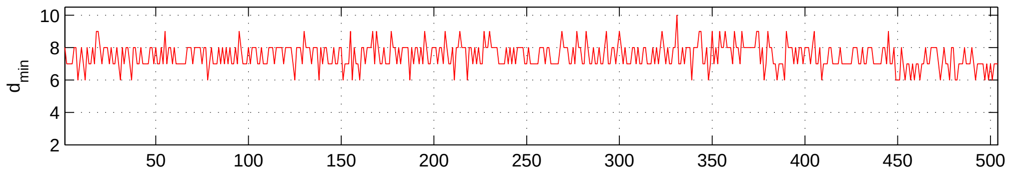
Figure 4.5: Performance of a random LDPC code, PEG LDPC code, and ITB LDPC code with parameters $N = 200$, $M = 100$, $d_V = 3$, and $d_F = 6$ with SP decoding for 80 iterations.

significantly better than the random LDPC code at high SNR. This is expected, since the random code has minimum ITB lower bounds of just $d_{\text{ITB-min}} \geq 2$, while the PEG LDPC codes and ITB LDPC codes have minimum ITB lower bounds of $d_{\text{ITB-min}} \geq 4$ and $d_{\text{ITB-min}} \geq 6$, respectively. The PEG and ITB simulations show similar performance until 4.5 dB, at which point the ITB LDPC code begins to outperform the

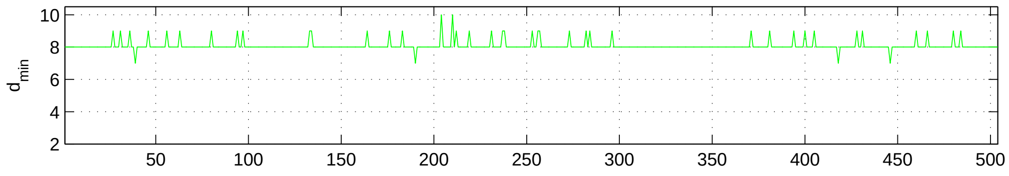
PEG LDPC code. The results indicate that there is a correlation between a code's ITB lower bound and the bit error rates at high SNR when using the SP decoder.



(a) Random LDPC



(b) PEG LDPC



(c) ITB LDPC

Figure 4.6: Independent tree-based lower bound on the minimum distance for each bit of a random LDPC code, PEG LDPC code, and ITB LDPC code with $N = 504$, $M = 252$, $d_V = 3$, and $d_F = 6$.

Table 4.1, Figure 4.6, and Figure 4.7 show similar results for low-density parity-check codes with $N = 504$, $M = 252$, $d_V = 3$, and $d_F = 6$. At SNRs greater than 3.0 dB the ITB LDPC code with minimum ITB lower bounds of $d_{\text{ITB-min}} \geq 7$ begins to outperform the random and PEG LDPC codes with $d_{\text{ITB-min}} \geq 2$ and $d_{\text{ITB-min}} \geq 6$,

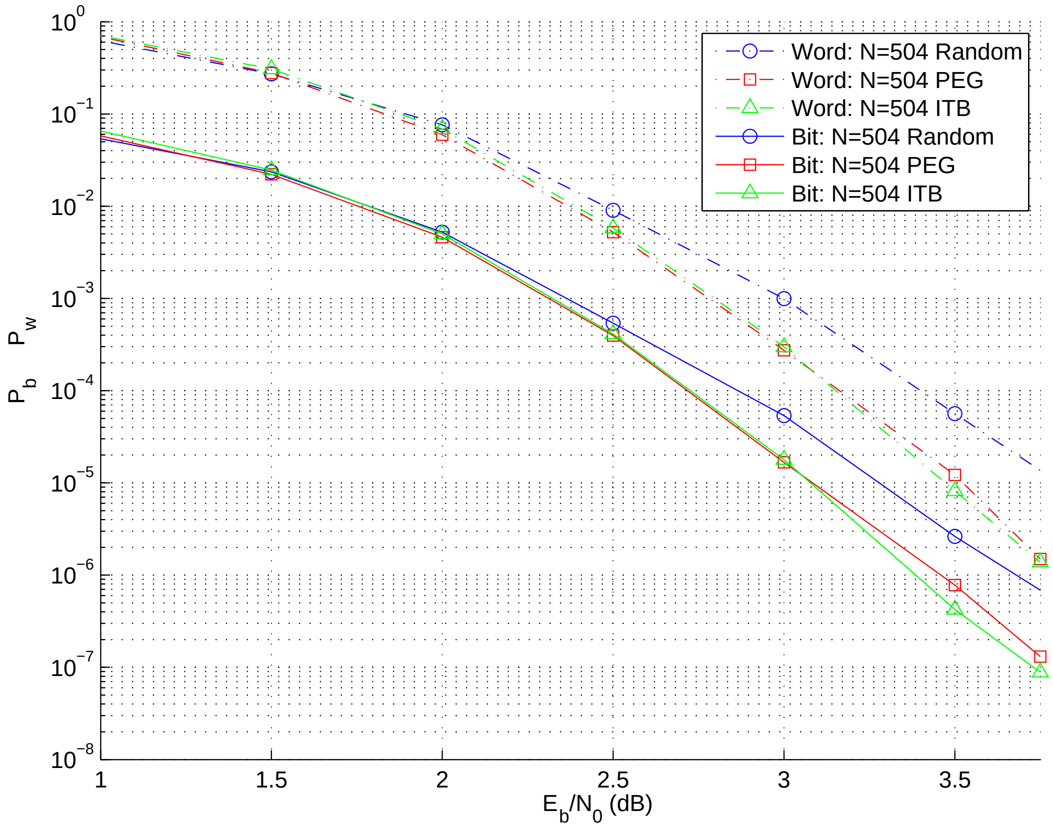
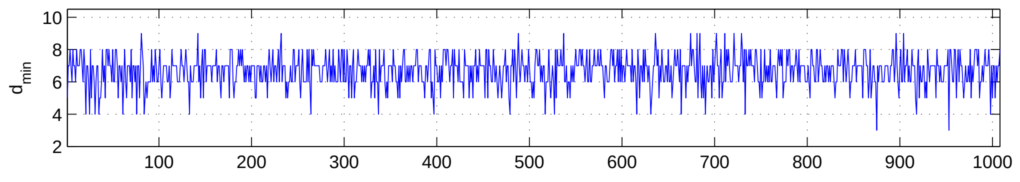


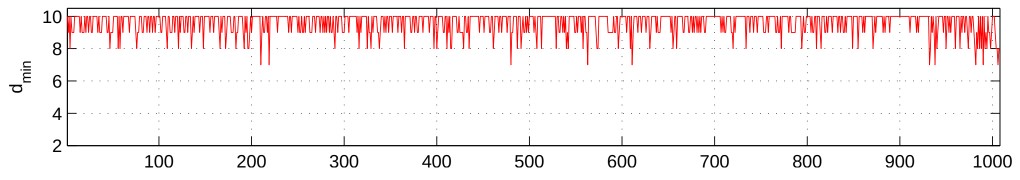
Figure 4.7: Performance of a random LDPC code, PEG LDPC code, and ITB LDPC code with parameters $N = 504$, $M = 252$, $d_V = 3$, and $d_F = 6$ with SP decoding for 80 iterations.

respectively.

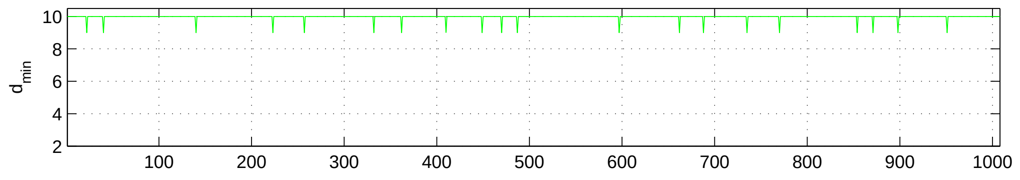
Finally, Table 4.1, Figure 4.8, and Figure 4.9 once again show similar results for low-density parity-check codes with $N = 1008$, $M = 504$, $d_V = 3$, and $d_F = 6$. At SNRs greater than 2.5 dB the ITB LDPC code with $d_{\text{ITB-min}} \geq 9$ begins to noticeably



(a) Random LDPC



(b) PEG LDPC



(c) ITB LDPC

Figure 4.8: Independent tree-based lower bound on minimum distance for each bit of a random LDPC code, PEG LDPC code, and ITB LDPC code with $N = 1008$, $M = 504$, $d_V = 3$, and $d_F = 6$.

outperform the random and PEG LDPC codes with $d_{\text{ITB-min}} \geq 3$ and $d_{\text{ITB-min}} \geq 7$, respectively.

The simulation results given in this section indicate that there is a strong correlation between higher independent tree-based lower bounds and improved performance at high SNR. Each of the ITB LDPC codes was constructed to have large ITB lower bounds. An unintended side-effect of the construction was the creation of LDPC

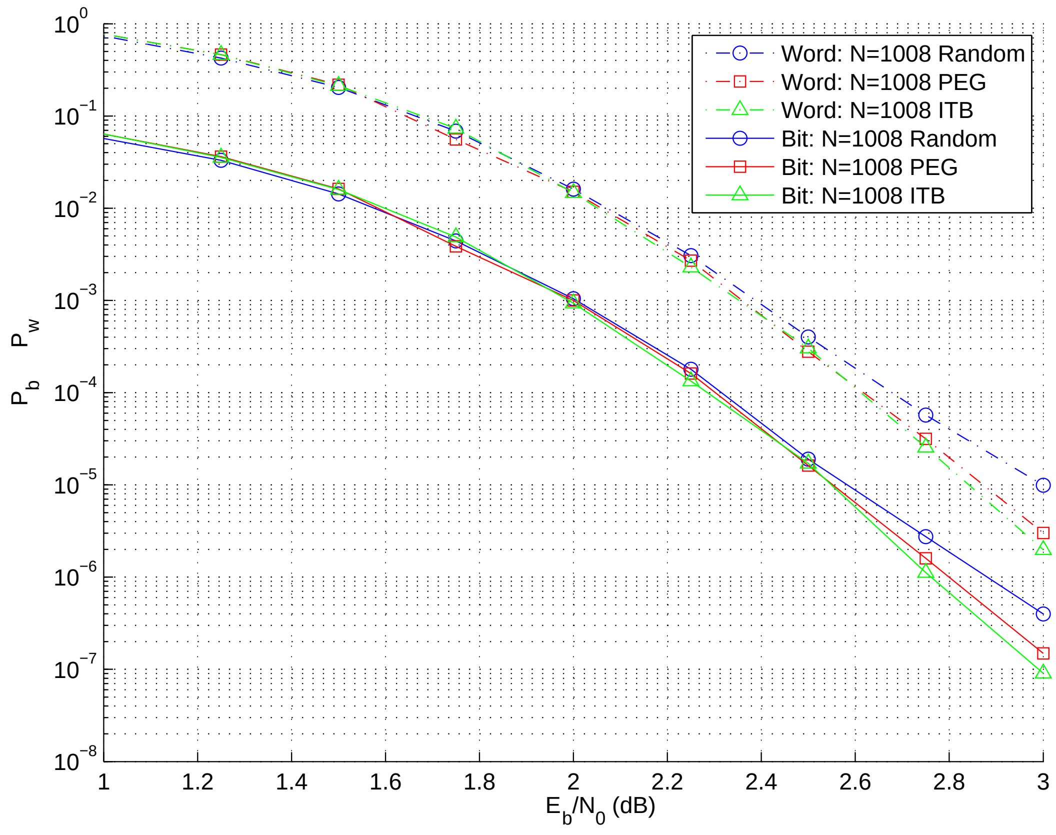


Figure 4.9: Performance of a random LDPC code, PEG LDPC code, and ITB LDPC code with parameters $N = 1008$, $M = 504$, $d_V = 3$ and $d_F = 6$ using SP decoding for 80 iterations.

codes with superior girth characteristics. While none of the ITB LDPC codes has higher minimum girth than the PEG LDPC codes, improvements are seen in the mean girths among all variable nodes for each of the codes in Table 4.1.

In Section 4.2, a new independent tree-based method for lower bounding the minimum distance of low-density parity-check code was presented. The ITB lower bound was utilized in the iterative construction of ITB LDPC codes. Simulations show that the new codes outperform random and PEG LDPC codes with identical variable and check node degree profiles using the SP decoder. In Section 4.4, LDPC codes with check node degree $d_F = 3$ are examined, and the benefits of ITB LDPC codes over PEG LDPC codes with $d_F = 3$ are shown using finite tree decoding on extrinsic trees.

4.4 LDPC Codes with Check Degree Three

The set of valid configurations on finite trees can be used to bound the probability of bit error at the root node. Consider a finite tree where the set of valid configurations coincides exactly with the set of codewords of an LDPC code C. More precisely, suppose there is a bijective function mapping the set of codewords C to the set of valid configurations on the finite tree, such that the codewords consist of the same variable nodes and have the same weight as the valid configurations to which they are mapped. Under these conditions, the bit error rate of finite tree decoding will be identical to that of ML decoding.

Unfortunately, it is often not possible for all configurations on the finite tree to coincide with codewords in C, since the variable nodes close to and including the leaf nodes of finite trees have a high degree of freedom. For example, on any set of k leaf nodes connected to the same check node are 2^{k-1} configurations that will satisfy the check node above it, regardless of the assignments to the rest of the variable nodes in the finite tree. A finite tree that has two or more leaf nodes connected to the same check node has a valid configuration with Hamming weight less than or equal to two. Thus, for a typical LDPC code with $d_{\min} > 2$, it is likely impossible to make the valid configurations on a finite tree coincide with the set of codewords C.

While it is likely not possible to get all valid configurations on finite trees to match those of the corresponding code, it might instead be possible to design a finite tree with correspondences between codewords in C that involve a variable node v_i and the deviations in a finite tree rooted at v_i.

Proposition 4.4.1. *Consider a minimum-weight codeword $\mathbf{c}_{\min}$ in the code C. Without loss of generality, let $\{v_1, \ldots, v_j\}$ be the support of $\mathbf{c}_{\min}$. Also, let the set $\{f_1, \ldots, f_k\}$ include all check nodes in the Tanner graph with at least one edge connected to the set of variable nodes $\{v_1, \ldots, v_j\}$. Assume that a single check node $f_m \in \{f_1, \ldots, f_k\}$ is connected to more than two variable nodes in the set $\{v_1, \ldots, v_j\}$. If a finite tree includes check node f_m, it is not possible for $\mathbf{c}_{\min}$ to project itself as a deviation in the finite tree.*

Proof. In order to project the codeword $\mathbf{c}_{\min}$ onto a finite tree, each variable node in the set $\{v_1, \ldots, v_j\}$ is set to one on the finite tree, and each variable node not in the set $\{v_1, \ldots, v_j\}$ is set to zero on the finite tree. Check node f_m is in the finite tree, which means that all of its neighboring variable nodes in the Tanner graph are also neighbors in the finite tree. Since f_m is connected to the set $\{v_1, \ldots, v_j\}$ more than two times, there will be more than two variable nodes connected to f_m in the finite tree that are set to one. Since deviations have a maximum of two variable nodes set to a binary one connected to a given check node in the finite tree, it is not possible for $\mathbf{c}_{\min}$ to project itself as a deviation in the finite tree. $\square$

Proposition 4.4.1 provides reason to believe that parity-check matrices limited to check node degree $d_F = 3$ have inherent advantages with respect to matching deviations on finite trees with codewords in the code C. Proposition 4.4.1 also highlights the difficulty in matching up deviations with codewords when there are check nodes with degree $d_f \geq 4$. When a check node has degree $d_f \geq 4$, it is possible that a codeword in the code has support from four or more of the variable nodes connected to the check node, and it is thus impossible to project the codeword as a deviation on the finite tree. One method for avoiding this scenario is to require that all the check nodes in the code have degree $d_f \leq 3$.

All finite trees have exactly two kinds of configurations that assign the root node to a binary 1: deviations, and disjoint configurations where exactly one of the configurations is a deviation. A *disjoint configuration* is a valid configuration on the finite tree where exactly the set of variable nodes V_i on the finite tree are assigned to a binary 1, such that there exist valid configurations V_j and V_k where $V_j \bigcup V_k = V_i$ and

$V_j \bigcap V_k = \emptyset$. On a finite tree with all degree-three check nodes, no check node on the finite tree can be connected to variable nodes from both V_j and V_k. The significance of this is that when codewords are projected onto finite trees with all degree-three check nodes they only appear as disjoint configurations on the finite tree if they are disjoint configurations on the Tanner graph. Thus, minimum-weight codewords in C can always be projected to deviations on finite trees with only degree-three check nodes. If the minimum-weight codeword in C is projected to the minimum-weight deviation on the finite tree, the bit error rate of finite tree decoding will approach that of ML decoding as the SNR grows large.

Extrinsic tree construction requires that the weight of each deviation is found at each step. As shown in Section 2.9, the number of deviations grows exponentially with the size of the computation trees. Since extrinsic trees are identical to computation trees at levels $0 \leq \ell \leq \frac{\mathcal{G}-2}{4}$, (2.9) and (2.11) can be used to analyze the growth rate in the number of deviations for trees that include a given number of variable nodes. Figure 4.10 shows the number of deviations given the number of variable nodes in computation trees of depth ℓ for rate $\frac{K}{N} = \frac{1}{4}$ LDPC codes. Results are given for codes with regular check node degrees $d_F = 3$, 4, 5, 6, and 7. The results indicate that the code with check node degree $d_F = 3$ has the smallest number of deviations per variable node. Note that even LDPC codes with check node degree $d_F = 3$ have computation trees with an exponential increase in the number of deviations as the

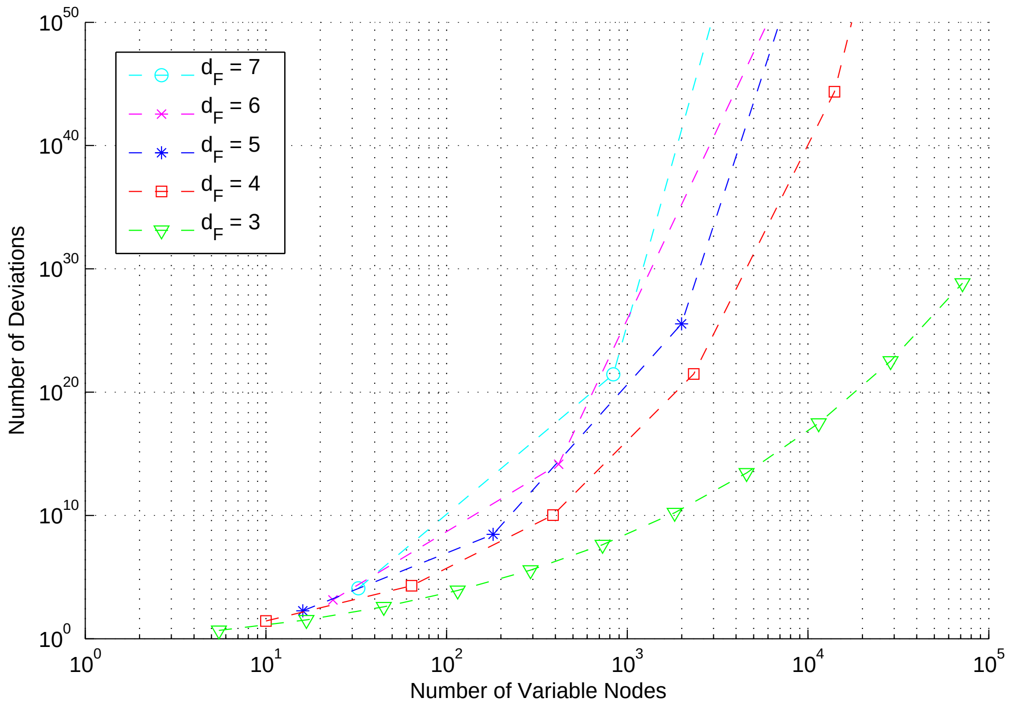


Figure 4.10: The number of deviations versus the number of variable nodes in a computation tree of depth ℓ for rate $\frac{K}{N} = \frac{1}{4}$ LDPC codes with varying check node degrees.

number of variable nodes increases. Therefore, due to the necessity of brute-force deviation enumeration, even codes with small check node degrees must have short-to-moderate block lengths to allow for extrinsic tree construction.

Figure 4.10 provides a good indication of the block lengths for which extrinsic tree construction is computationally feasible. While the structure of extrinsic trees is not the same as the structure of computation trees, given their similarities it is not unreasonable to assume a similar growth rate in the number of deviations per variable node. Under this assumption, if the number of deviation weights computed during extrinsic tree construction were limited to 10^9, it may not be possible to construct extrinsic trees for rate $\frac{K}{N} = \frac{1}{4}$ LDPC codes with $d_F = 3$ and block lengths $N \geq 1500$. For rate $\frac{1}{4}$ LDPC codes with $d_F = 4$ the block length would be limited to $N \leq 350$.

Recall that the goal of finite-tree decoding on extrinsic trees is to perform well at high signal-to-noise ratios by creating finite trees with large minimum-weight deviations. Codes with all degree-three check nodes have a natural correspondence between deviations and valid configurations on finite trees. Additionally, as seen in Figure 4.10, the number of deviation weights that need to be computed is significantly less for codes with $d_F = 3$ compared to codes with higher check node degree. For these two reasons, the performance of finite tree decoding on extrinsic trees using LDPC codes with $d_F = 3$ is further examined.

To create a full-rank parity-check matrix with check node degree $d_F = 3$, it is necessary to have an average variable node degree $\frac{1}{N}\sum_{i=1}^{N} d_{v_i} < 3$, so that there are less rows than columns. Another important restriction on the variable node degree d_{v_i} of any variable node v_i is that the degree satisfies the condition $d_{v_i} \geq 2$. A variable node with degree one is incapable of passing extrinsic information in the graph, since it will always appears as a leaf in the computation tree. Therefore, only parity-check matrices with $d_F = 3$, $d_{v_i} \geq 2$ for all $i = 1, \ldots, N$, and $\frac{1}{N}\sum_{i=1}^{N} d_{v_i} < 3$

are considered. Given these constraints, it is possible to have variable node degrees where $d_{v_i} > 3$. However, an extensive search would be required to find optimal degree profiles. The results given in Section 3.3 indicate that a high degree of irregularity among the variable node degrees results in poor finite tree decoding performance. For this reason, LDPC codes in this section are chosen to have variable node degrees as close to regular as possible, with $2 \leq d_{v_i} \leq 3$ for all $i = 1, \ldots, N$.

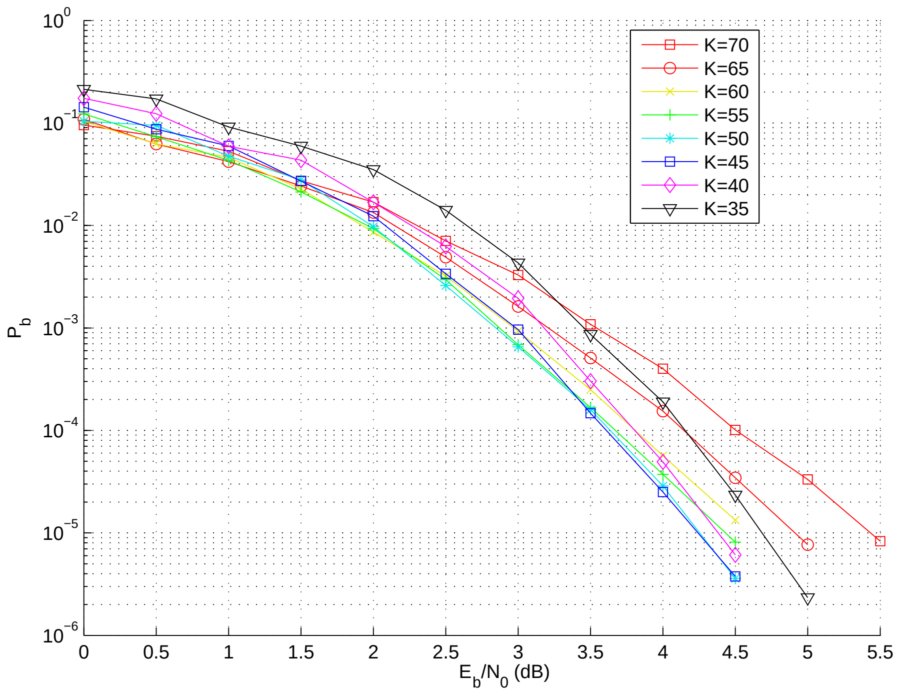


Figure 4.11: Probability of bit error of length N=210 PEG LDPC codes with dimension K=70, 65, 60, 55, 50, 45, 40, and 35 and MS decoding.

The restriction of $d_F = 3$, and $2 \leq d_{v_i} \leq 3$ for all $i = 1, \ldots, N$ allows for a range of rates given by $\frac{2}{3} \leq \frac{K}{N} < 1$. Given these constraints, Figure 4.11 shows the performance of various PEG LDPC codes with length $N = 210$ and MS decoding. The length was chosen arbitrarily, since the rate of the code is the variable of interest in this example. Notice that the dimension $K = 70$ PEG LDPC code has the highest probability of bit error. This can be attributed to the fact that all variable nodes have degree $d_F = 2$, making the $K = 70$ code a cycle code. Cycle codes are known to have poor minimum distance properties [36]. As the dimension of the code decreases to $K = 50$, there are sixty variable nodes with degree three and the performance is approximately 1.25 dB better than the cycle code at $P_b = 10^{-5}$. As the dimension continues to decrease below $K = 50$, the performance gets worse again. This is not surprising since, as the dimension of the code approaches $K = 1$, the capacity of the LDPC code will approach that of an uncoded transmission. As seen in Figure 4.13, rate $\frac{K}{N} = \frac{50}{210} = 0.2381$ codes perform as good as or better than codes with other rates for this block length. For this reason, codes with check node degree $d_F = 3$ and rates close to $\frac{K}{N} = 0.2381$ are further examined.

Recall that Proposition 4.4.1 suggests that check node degree $d_F = 3$ codes should have better performance with finite tree decoding on extrinsic trees at high SNR than codes with higher check node degree. Figure 4.13 shows that the choice of $d_F = 3$ is also preferable to $d_F = 4$ and $d_F = 5$ for MS decoding of rate $\frac{K}{N} = 0.2381$ PEG LDPC codes with block lengths $N = 210$ and $N = 1050$. At $P_b = 10^{-4}$, the check

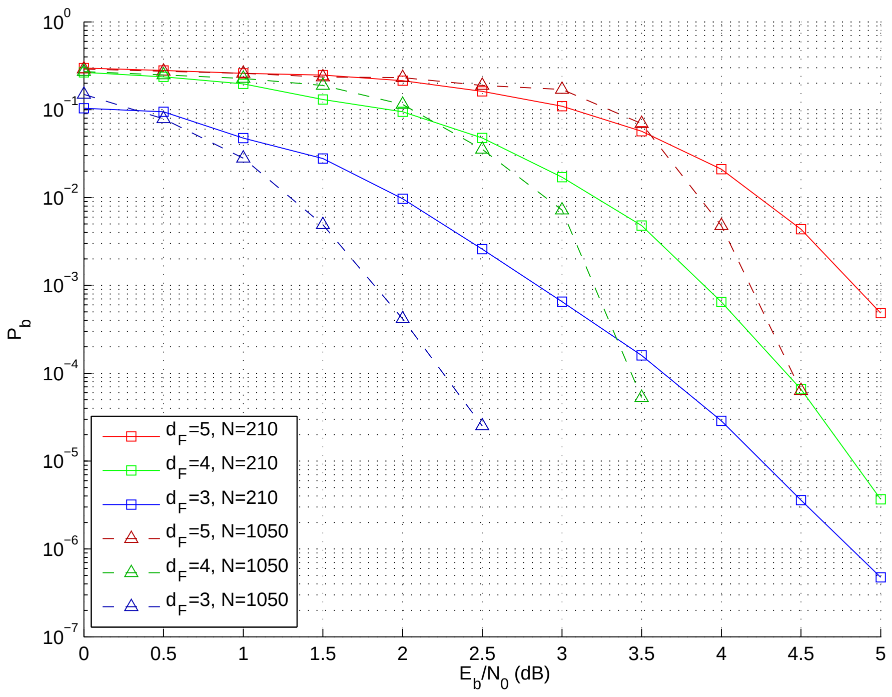


Figure 4.12: Probability of bit error of rate $\frac{K}{N} = 0.2381$ PEG LDPC codes with check node degrees $d_F = 3$, $d_F = 4$, and $d_F = 5$ with block lengths of N=210 and N=1050 and MS decoding.

node degree $d_F = 3$, length $N = 1050$ PEG LDPC code is over 1.0 dB better than the $d_F = 4$, $N = 1050$ code and 2.0 dB better than the $d_F = 5$, $N = 1050$ code.

Theoretical analysis can also be used to compare the performance capabilities of rate $\frac{K}{N} = 0.2381$ codes with check node degree $d_F = 3$, $d_F = 4$, and $d_F = 5$ as the SNR grows large. Figure 4.13 plots of the lower bound on block length N given by (4.1) versus the lower bound on the minimum distance $d_{\min}$ given by (4.4). The

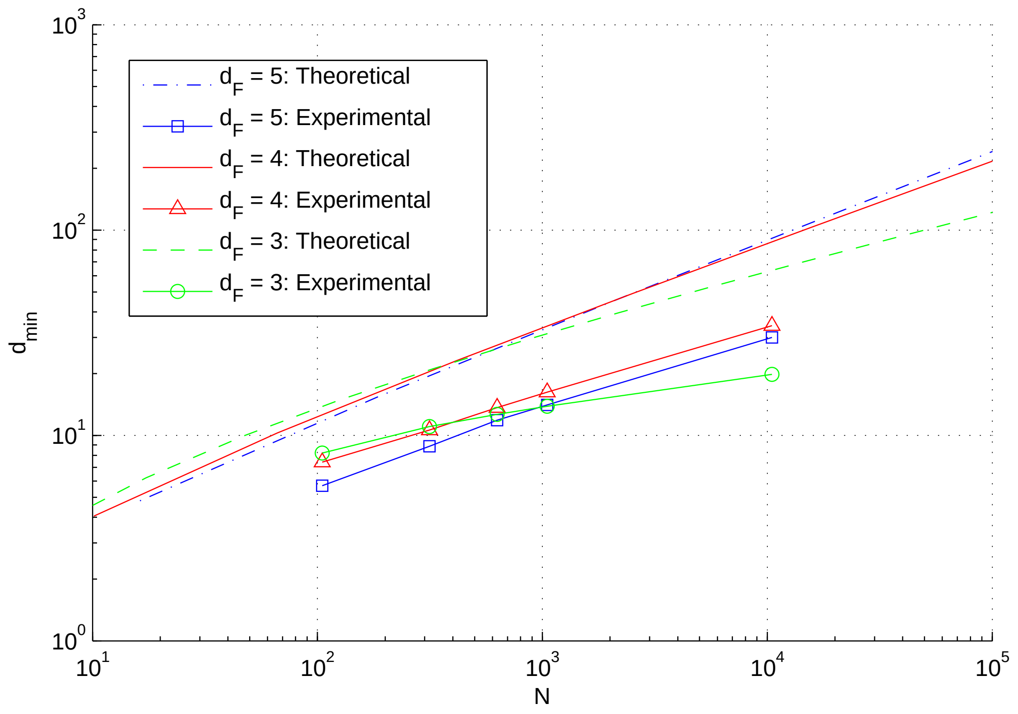


Figure 4.13: Theoretical and experimental lower bounds on the minimum distance as a function of block length for $d_F = 3$, $d_F = 4$, and $d_F = 5$ rate $\frac{K}{N} = 0.2381$ codes.

theoretical results suggest that using $d_F = 3$ codes is preferable for block lengths of $N \leq 300$. At block lengths of $300 < N \leq 2000$, using $d_F = 4$ codes is preferable, and for $N > 2000$ the $d_F = 5$ codes should have better performance as the SNR grows large.

Figure 4.13 also shows the parameters of real codes generated using the progressive edge-growth algorithm. The PEG LDPC codes with $d_F = 3$, 4, and 5 were generated for various block lengths. Then, a lower bound on the minimum distance is obtained for each variable node in the code. Instead of simply using the girth of the code, the ITB lower bound is computed for each variable node. The results in Figure 4.13 show the mean $d_{\text{ITB-min}}$ over all variable nodes.

The experimental results shown in Figure 4.13 are similar to the theoretical results, in terms of the relative parameters of the three code types. They suggest that using $d_F = 3$ codes is preferable for block lengths of $N \leq 400$. At block lengths of $400 < N \leq 1000$, using a $d_F = 4$ codes is preferable, and for $N > 1000$ the $d_F = 5$ codes should have better performance as the SNR grows large.

Note that the theoretical and experimental results presented in Figure 4.13 indicate that $d_F = 4$ codes are preferable to $d_F = 3$ codes at block length $N = 1050$ as the SNR grows large. Despite the theoretical and experimental results given in Figure 4.13, the simulations shown in Figure 4.4.1 indicate that the $d_F = 3$, $N = 1050$ code performs much better than the $d_F = 4$ and $d_F = 5$ codes for $P_b \geq 10^{-4}$. It is possible that the performance of the $d_F = 4$ and $d_F = 5$ LDPC codes have better performance at higher SNR, since the simulation results presented in Figure 4.4.1 are limited to $P_b \geq 10^{-5}$.

This section has examined the properties of low-density parity-check codes with check nodes of degree three. Both theoretical and experimental results show that short-to-moderate block length ($N \leq 1500$) LDPC codes with check degree $d_F = 3$ have advantages over similar codes with higher check node degrees. One of the most important advantages, with regards to finite tree decoding, is the natural correspondence between codewords in C and deviations on finite trees with $d_F = 3$. Other advantages include a slower growth rate in the number of deviations per variable node in the finite tree, and higher ITB lower bounds on the minimum distance for fixed block length LDPC codes. Section 4.5 presents simulation results and extrinsic tree properties of various LDPC codes with check node degree $d_F = 3$. The advantages of using ITB LDPC codes over PEG LDPC codes are also shown.

4.5 Finite Tree Decoding on Extrinsic Trees using LDPC Codes with Check Degree Three

Finite tree decoding on extrinsic trees has been used to decode a block length $N = 210$, dimension $K = 50$ progressive edge-growth low-density parity-check code with regular check node degree $d_F = 3$. Since the goal of finite tree decoding on extrinsic trees is to perform well at high SNR, it is sufficient to obtain the minimum-weight deviation on the extrinsic trees in order to predict the probability of bit error of finite tree decoding on extrinsic trees as the SNR grows large.

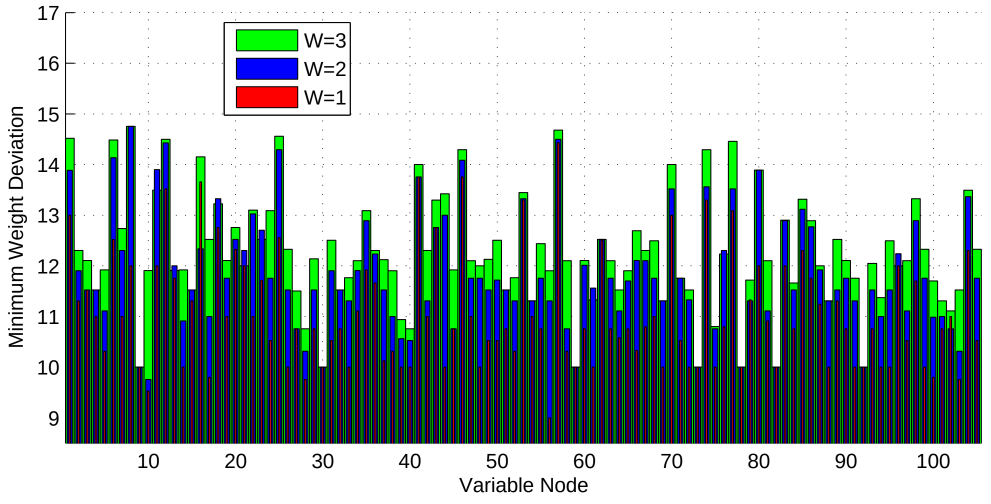
Table 4.2 shows the mean of the minimum-weight deviations on extrinsic trees rooted at variable nodes of degree $d_V = 2$ and 3. The extrinsic tree construction was performed using $\mathcal{W} = 1$, 2, and 3. The overall mean of the minimum-weight

Weights Checked	Mean Min-Weight Deviation ($d_v = 2$)	Mean Min-Weight Deviation ($d_v = 3$)	Mean Min-Weight Deviation (All)	# of Best Trees
$\mathcal{W} = 1$	10.9815	13.7930	11.7848	4
$\mathcal{W} = 2$	11.8340	14.2348	12.5199	30
$\mathcal{W} = 3$	12.2157	14.4794	12.8625	159

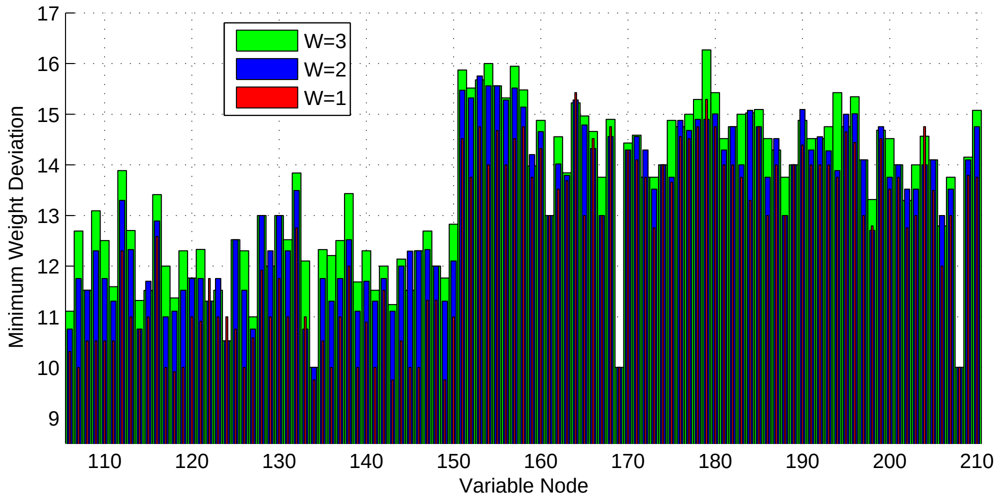
Table 4.2: Mean minimum-weight deviations on the extrinsic tree for the length $N = 210$, dimension $K = 50$, check node degree $d_F = 3$ LDPC code.

deviations for extrinsic trees rooted at all variable nodes is also given, along with the number of times that each of the three methods resulted in the extrinsic tree rooted at a particular variable node with the highest minimum-weight deviation, given by "# of Best Trees". Two key observations can be made from the results given in Table 4.2. First, extrinsic trees rooted at variable nodes with degree two have lower minimum-weight deviations than extrinsic trees rooted at variable nodes of degree three. Second, increasing $\mathcal{W}$, the number of deviation weights tracked, also increases the minimum-weight of deviations on the extrinsic trees.

Two graphical depictions of the minimum-weight deviations are given in Figures 4.14 and 4.15. The bar plot in Figure 4.14 shows each individual minimum-weight deviation on the extrinsic trees. An important observation from Figure 4.14 is that there are exactly ten extrinsic trees that have a minimum-weight deviation with



(a) Variable Nodes: 1-150



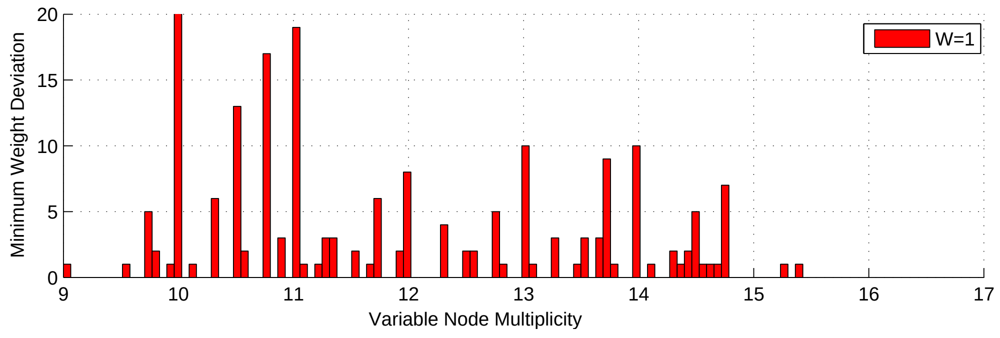
(b) Variable Nodes: 106-210

Figure 4.14: Bar graph of the minimum deviation weights on the extrinsic tree for the length $N = 210$, dimension $K = 50$, check node degree $d_F = 3$ LDPC code.

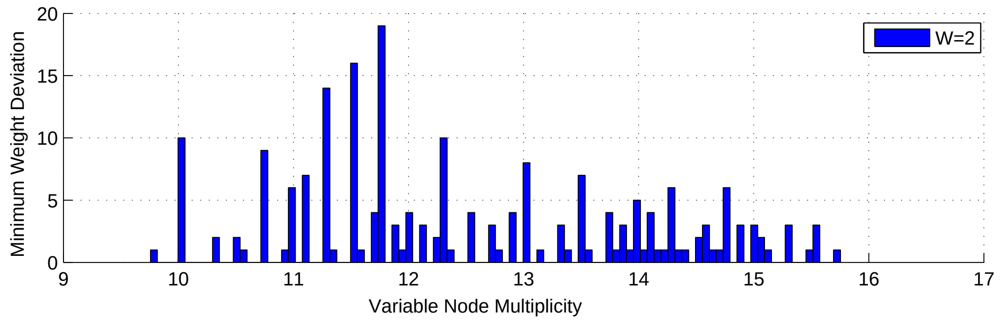
weight equal to ten when using extrinsic tree construction with $\mathcal{W} = 3$. After further investigation, it has been determined that the minimum-weight deviations on each of these trees corresponds with a codeword of Hamming weight ten in the code. This is easily determined by checking the parity of the bits involved in each of the ten deviations. From this observation come two important conclusions: First, the minimum-weight codeword in the code has Hamming weight equal to ten, and second the probability of bit error of finite tree decoding on extrinsic trees with $\mathcal{W} = 3$ approaches that of ML decoding as the SNR grows large.

The histograms given in Figure 4.15 show that finite tree decoding of extrinsic trees with $\mathcal{W} = 1$ and $\mathcal{W} = 2$ does not approach maximum-likelihood performance as the signal-to-noise ratio gets large, like finite tree decoding of extrinsic trees with $\mathcal{W} = 3$. This is because there are extrinsic trees with minimum-weight deviations that have weight less than ten when $\mathcal{W} = 1$ and when $\mathcal{W} = 2$. It should also be noted that, in Figure 4.15, the entire distribution of minimum-weight deviations appears to shift to the right with each additional deviation weight that is tracked during extrinsic tree construction.

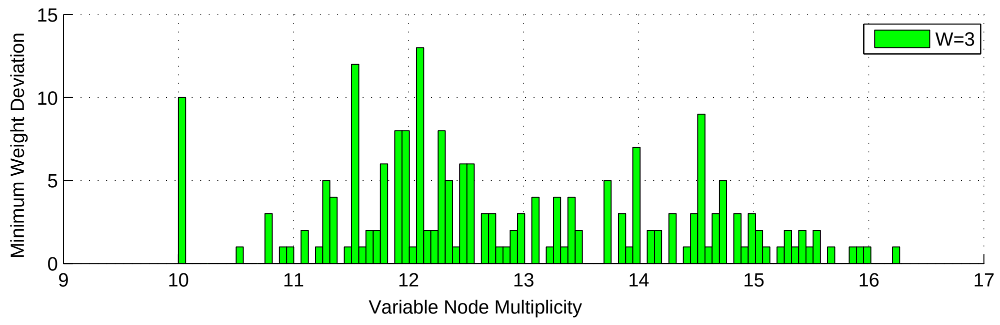
Minimum-weight deviations obtained from extrinsic tree construction of an independent tree-based low-density parity-check code are given in Table 4.3. Compared to the PEG LDPC results, the ITB LDPC results demonstrate a 0.5082 improvement in the mean minimum-weight deviations over all extrinsic trees rooted at degree $d_V = 2$



(a) $\mathcal{W} = 1$



(b) $\mathcal{W} = 2$



(c) $\mathcal{W} = 3$

Figure 4.15: Histograms of the minimum deviation weights on the extrinsic trees of a length $N = 210$, dimension $K = 50$, check node degree $d_F = 3$ PEG LDPC code.

Code: Weights Checked	Mean Min-Weight Deviation ($d_v = 2$)	Mean Min-Weight Deviation ($d_v = 3$)	Mean Min-Weight Deviation (All)
PEG: $\mathcal{W} = 3$	12.2157	14.4794	12.8625
ITB: $\mathcal{W} = 3$	12.7239	14.6544	13.2754

Table 4.3: Mean minimum-weight deviations on the extrinsic tree for the length $N = 210$, dimension $K = 50$, check node degree $d_F = 3$ LDPC code generated with ITB LDPC code construction.

variable nodes. On the other hand, the extrinsic trees rooted at degree $d_V = 3$ variable nodes only had a 0.1750 improvement in the mean minimum-weight deviations. Figure 4.16 shows a histogram of the minimum-weight deviations for ET decoding of the ITB LDPC code. Notice that the overall minimum-weight deviation over all the extrinsic trees has weight 10.7561.

Figure 4.17 shows the bit error rates of finite tree decoding on extrinsic trees for both the progressive edge-growth and independent tree-based low-density parity-check codes. The performance improvement gained by increasing $\mathcal{W}$ for the PEG LDPC code is evident. The performance gain obtained by using the ITB LDPC code is also clear in Figure 4.17. As expected, the performance gains are greatest at high SNR due to the increase in the overall minimum-weight deviation from 10.0 to 10.7561. It is worth noting that the weight 10.7561 deviation does not correspond to

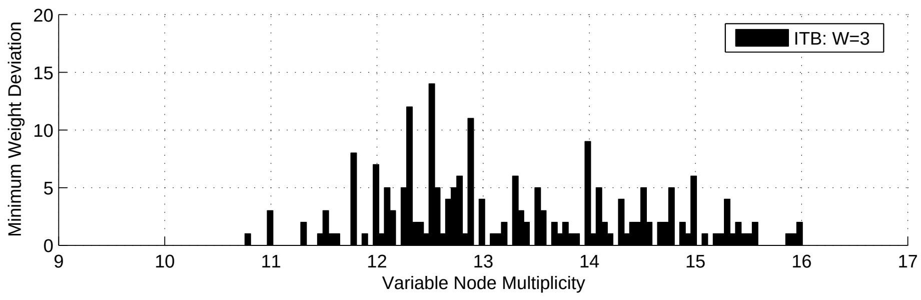


Figure 4.16: Histogram of the minimum deviation weights on the extrinsic trees with $\mathcal{W} = 3$ of a length $N = 210$, dimension $K = 50$, check node degree $d_F = 3$ ITB and PEG LDPC code.

a codeword in the original code. Therefore, finite tree decoding of extrinsic trees with $\mathcal{W} = 3$ does not achieve ML decoding performance as the SNR grows large. However, by increasing $\mathcal{W}$, the minimum-weight deviation would likely increase much like it was shown to in Figures 4.14 and 4.15.

The results of finite tree decoding on extrinsic trees and min-sum decoding of length $N = 1050$ ITB LDPC codes with $K = 325$, $K = 300$, and $K = 275$ are given in Figures 4.18, 4.19, and 4.20. The minimum-weight deviation among the extrinsic trees of both dimension $K = 325$ and $K = 300$ ITB LDPC codes is twelve. Each of these codes contains codewords of weight twelve. Therefore, the performance of finite tree decoding on the extrinsic trees approaches ML performance as the SNR gets large. The minimum-weight deviation among the extrinsic trees for the dimension $K = 275$ ITB LDPC codes is fourteen. This code contains a single codeword of weight fourteen.

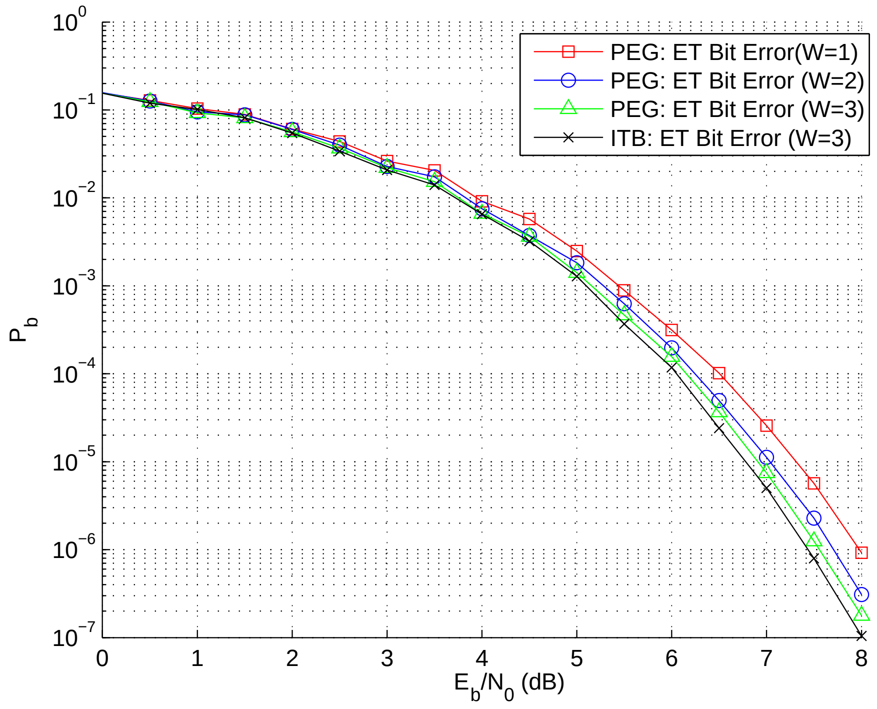


Figure 4.17: Probability of bit error of finite tree decoding on extrinsic trees of a length $N = 210$, dimension $K = 150$, check node degree $d_F = 3$ PEG LDPC code and a ITB LDPC code.

Therefore, the performance of finite tree decoding on the extrinsic trees of the $K = 275$ code also approaches ML performance as the SNR gets large. Unlike the $K = 325$ and $K = 300$ codes, which only need $\mathcal{W} = 3$, the code with dimension $K = 275$ requires $\mathcal{W} = 6$ to achieve ML performance as SNR gets large. This is likely due to the higher minimum weight deviation on the extrinsic trees. While the performance of finite tree decoding on extrinsic trees is worse than that of MS decoding over all simulated SNR, the simulated bit error rate of MS decoding decreases at a slower rate at SNRs greater than 6.0 dB, suggesting that the performance of MS decoding might be surpassed at higher SNR.

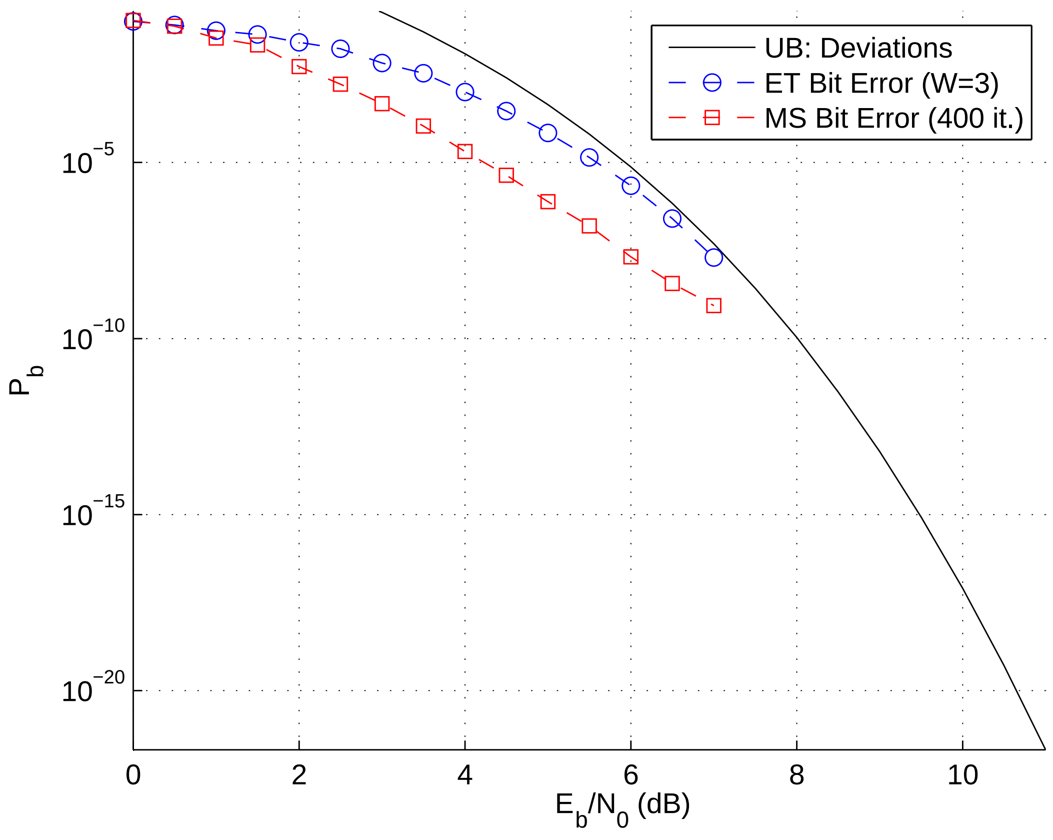


Figure 4.18: Probability of bit error of finite tree decoding on extrinsic trees with $\mathcal{W} = 3$ and MS decoding of a length $N = 1050$, dimension $K = 325$, check node degree $d_F = 3$ ITB LDPC code.

This section examined the properties and bit error rates of low-density parity-

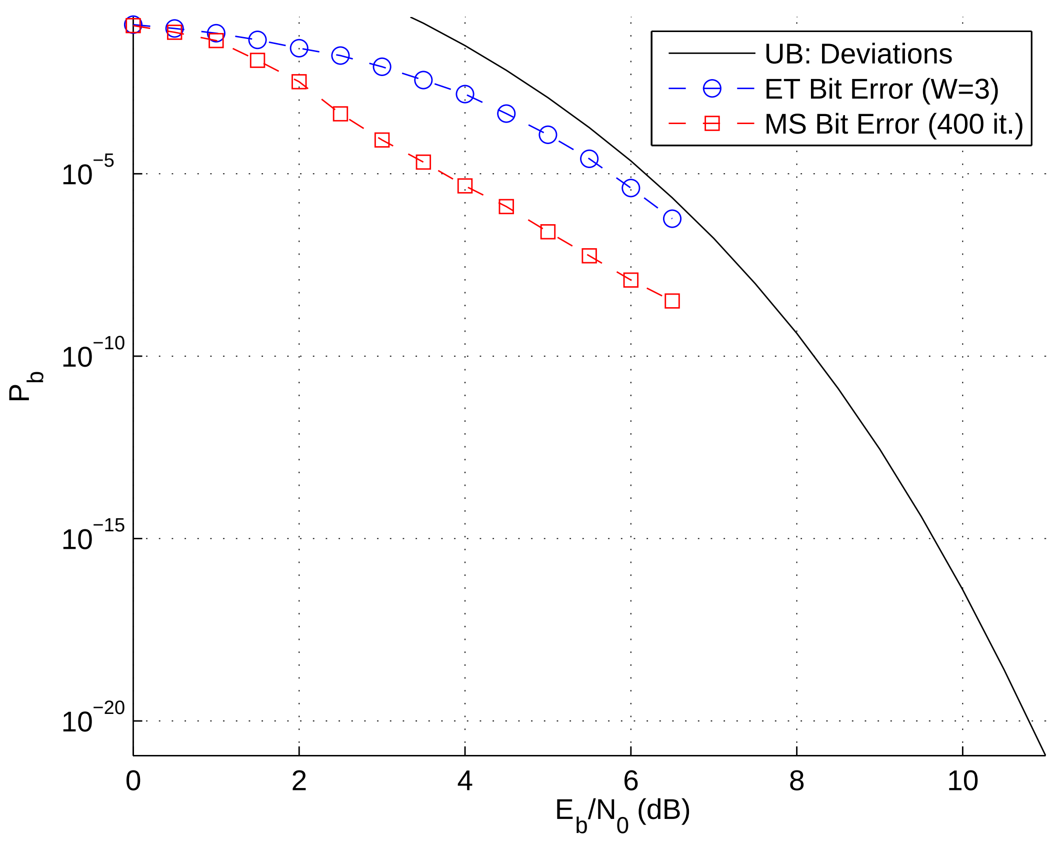


Figure 4.19: Probability of bit error of finite tree decoding on extrinsic trees with $\mathcal{W} = 3$ and MS decoding of a length $N = 1050$, dimension $K = 300$, check node degree $d_F = 3$ ITB LDPC code.

check codes with check node degree $d_F = 3$. Analysis of extrinsic tree construction of LDPC codes shows that, in some cases, the construction makes it possible to identify minimum-weight codewords in the code. In general, identifying the minimum-weight of LDPC codes is a difficult problem, since brute-force methods require that the

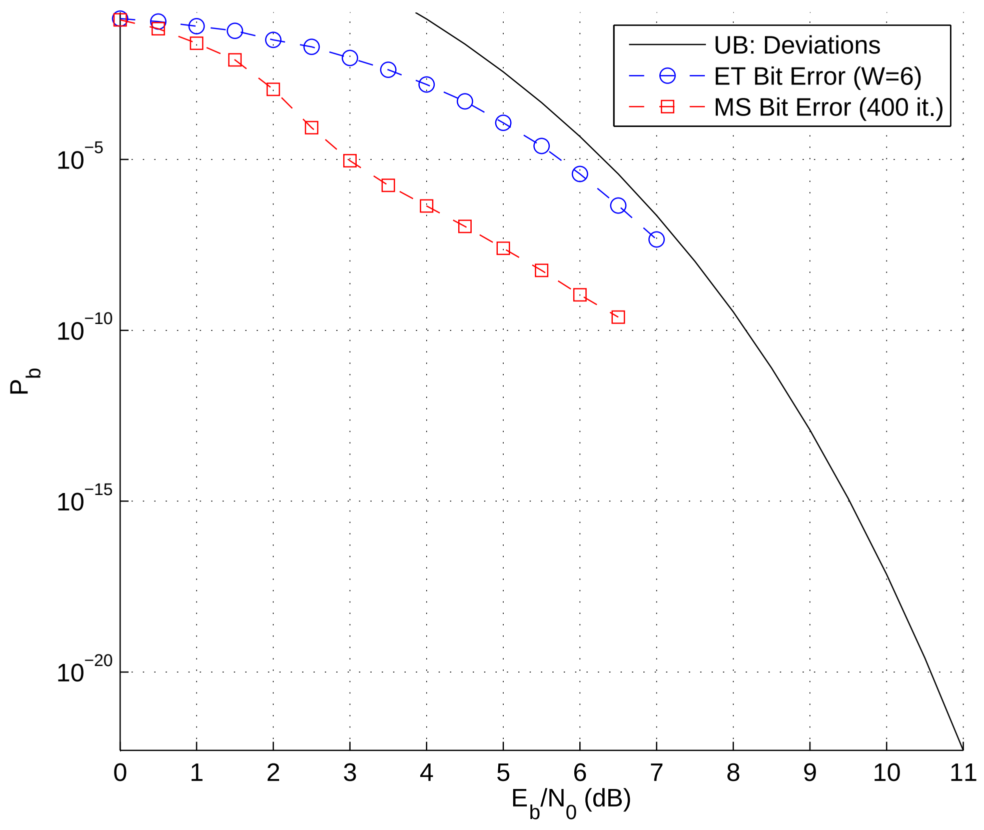


Figure 4.20: Probability of bit error of finite tree decoding on extrinsic trees with $\mathcal{W} = 3$ and MS decoding of a length $N = 1050$, dimension $K = 275$, check node degree $d_F = 3$ ITB LDPC code.

weight of each of the 2^K codewords is checked. After extrinsic tree construction, if the minimum-weight deviation over all extrinsic trees is equal to the minimum distance of the LDPC code, the bit error rate of finite tree decoding on the extrinsic trees approaches that of ML decoding as the SNR grows large.

Recall the two primary goals of finite tree decoding on extrinsic trees: 1) Allow for tractable performance analysis, and 2) achieve bit error rates comparable to iterative decoders like MS and SP at high SNR. Since ML decoding is optimal, it is not possible for iterative decoders to achieve better performance than ML decoding. Therefore, when finite tree decoding on extrinsic trees has bit error rates approaching that of ML decoding at high SNR, it is certain that this performance also approaches or exceeds that of iterative decoders like the MS decoder at high SNR. Therefore, by combining the use of deviation-based upper bounds and a resulting bit error rates that approaches that of ML decoding as the SNR gets large, finite tree decoding on extrinsic trees is capable of achieving both goals.

Chapter 5

Conclusion

Wiberg laid the foundation for future analysis of the min-sum and sum-product decoders by introducing the concept of computation trees and deviations[13]. He proved that the probability of bit error at the root node of a computation tree can be accurately predicted at high SNR using only the weight of the deviations. Unfortunately, as shown in Section 2.9, the use of deviations to bound the performance of the MS and SP decoders becomes computationally intractable after just a small number of iterations. While several methods have since been developed to estimate the error rates of the MS and SP decoder [25, 26, 28, 29, 17, 18], none are capable of producing bounds on their performance for finite codes used on the BIAWGN channel. This lack of performance bounds on MS and SP decoding makes it impossible to guarantee bit error rates beyond the reach of simulations. This uncertainty has inhibited the use of LDPC codes in applications requiring exceptionally low bit error rates.

In this dissertation, finite tree-based decoding is introduced as an alternative method for decoding low-density parity-check codes. While the computation trees modeling MS and SP decoding quickly grow too large to allow for performance bounds, the number of nodes used in finite tree-based decoding is kept small enough to allow for performance bounds. The two goals of finite tree-based decoding are to allow for tractable performance bounds, and to have bit error rates comparable to that of current iterative decoders, such as MS and SP. The two steps of finite tree-based decoding are the construction of the finite trees and the decoding on the finite trees.

In Chapter 3, several new methods for constructing finite trees were given, along with a single method for decoding on the finite trees. The following four methods for constructing finite trees for LDPC codes were given: iterative tree construction, independent tree construction, extrinsic tree construction, and deviation path-forcing tree construction. Iterative tree construction builds trees based upon the simulated error rate at a specific SNR. Finite tree decoding on iterative trees shows improvement over MS decoding after a small number of iterations, and demonstrates that decoding on smaller trees can achieve similar error rates compared to much larger computation trees. However, the method for constructing iterative trees relies on the assumption that the simulated probability of bit error at the root node is accurate. Even when using an adaptive method for establishing statistical confidence in the construction, the iterative trees have noticeably worse error rates when compared to MS after a large number of iterations.

Independent trees are the simplest among the four to construct, and the exact performance can be determined using density evolution. Despite these two advantages, simulations show that finite tree decoding on independent trees has much higher error rates than MS decoding. Extrinsic tree construction requires brute-force deviation enumeration for each set of nodes added to the tree. Therefore, the complexity of extrinsic tree construction is much higher than independent tree construction. However, simulations show that the performance of finite tree decoding on extrinsic trees is capable of competing with MS decoding at high SNR. Finally, DPFT construction provides a simplified method of construction when compared to extrinsic tree construction. In addition, the enumeration of all the deviations on the DPFTs is very straightforward once the trees are constructed. Unfortunately, the size of the DPFTs is such that they consume an exponentially increasing amount of computer memory as the minimum weight of the deviations increases.

In Chapter 4, low-density parity-check code properties are examined and a new method for bounding the minimum distance is introduced. This new lower bound on the minimum distance is used as a cost criteria for ITB LDPC code construction. The ITB LDPC codes are shown to perform better than PEG LDPC codes when using existing iterative decoders. Codes with check node degree $d_F = 3$ are examined in detail, and are shown to have theoretical benefits with regards to finite tree-based decoding, due to the natural correspondence between deviations and codewords on finite trees with maximum check node degree $d_F = 3$. Simulations of PEG LDPC codes and ITB LDPC codes are given for finite tree decoding using codes of length $N = 210$ and $N = 1050$. The simulations show improved bit error rates for ITB LDPC codes when compared to PEG LDPC codes. Additionally it is shown that, with some codes, the performance of finite tree decoding on extrinsic trees will approach that of ML decoding as SNR increases. This property is a result of the minimum deviation weight among all the extrinsic trees being equal to the weight of the minimum-weight codeword in the LDPC code.

Therefore, finite tree decoding on extrinsic trees is capable of accomplishing both of the original goals of this dissertation: 1) Performance that competes with iterative decoders, and 2) performance that allows for tractable analysis. Tractable analysis comes from the fact that the set of deviations, computed during extrinsic tree construction, is used to upper bound the bit error rates. Additionally, ML performance at high SNR ensures that the performance of finite tree decoding on extrinsic trees is comparable to existing decoders like MS and SP at high SNR.

There are several avenues of research that might lead to improvements to the finite tree-based decoders presented in this dissertation. For example, the addition of redundant rows to the parity check matrix has been shown to result in lower bit error rates for finite tree decoding on extrinsic trees [16]. Extrinsic tree construction could be modified to use the redundant nodes only in situations where low-weight deviations would otherwise be created. The problem with using redundant rows is that a systematic way of creating redundant rows that benefit finite tree-based decoders is not yet known.

The brute-force method of enumerating deviations and their weights contributes heavily to the complexity of extrinsic tree construction. Therefore, more efficient methods for enumerating the deviations on finite trees would greatly reduce the complexity of construction. Memory storage can also be a concern with finite tree-based decoders. However, portions of the N finite trees are likely to be identical. Thus, if there were a way of grouping together portions of finite trees that are identical, the memory required to store the finite trees could be significantly reduced. It might also be possible to improve upon the method of scaling the LLR cost among variable nodes presented in Section 3.3. Instead of scaling the cost by the total number of copies in the tree, it might be better to scale by the maximum number of copies that can appear in the same deviation.

Simulations have shown that the min-sum decoder often outperforms finite tree-based decoders at low SNR. Therefore, the creation of a hybrid decoder that uses both MS decoding and finite tree decoding on extrinsic trees may result in a decoder with excellent performance at both low and high SNRs. Unlike MS or SP decoding by themselves, this hybrid decoder would have guaranteed performance beyond the reach of simulations. One way of creating the hybrid decoder might be to use finite tree decoding on extrinsic trees only if MS or SP decoding fails to output a codeword. However, this relies on the assumption that MS and SP decoding never output a codeword unless it is the ML codeword, which is not always true. Another way of creating the hybrid decoder might be to decode with finite tree decoding on extrinsic trees first, and if this fails to produce a codeword, use the MS and SP decoders as a backup. In this way, the bounded performance of finite tree decoding on extrinsic trees would be preserved.

Bibliography

[1] C. E. Shannon, “A mathematical theory of communication,” *Bell System Technical Journal*, vol. 27, pp. 379–423 and 623–656, July and October 1948.

[2] R. W. Hamming, “Error detecting and error correcting codes,” *Bell System Technical Journal*, vol. 29, pp. 147–160, April 1948.

[3] Peter Elias, “Coding for noisy channels,” *IRE Convention Record*, vol. 4, pp. 37–47, 1955.

[4] A. Viterbi, “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm,” *IEEE Transactions on Information Theory*, vol. 13, pp. 260–269, April 1967.

[5] R. G. Gallager, “Low-density parity-check codes,” *IEEE Transactions on Information Theory*, vol. 8, pp. 21–28, January 1962.

[6] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding," in *Proceedings of the 1993 IEEE International Conference on Communications*, Geneva, Switzerland, 1993, pp. 1064–1070.

[7] D. J. C. MacKay and R. M. Neal, "Near Shannon limit performance of low-density parity check codes," *IEE Electronic Letters*, vol. 32, no. 18, pp. 1645–1646, August 1996.

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

[9] A. Morello and V. Mignone, "DVB-S2: The second generation standard for satellite broad-band services," in *Proceedings of the IEEE*, January 2006, vol. 94, pp. 210–227.

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

[11] L. C. Pérez, J. Seghers, and D. J. Costello, "A distance spectrum interpretation of turbo codes," *IEEE Transactions on Information Theory*, vol. 42, pp. 1698–1709, November 1996.

[12] R. Tanner, "A recursive approach to low complexity codes," *IEEE Transactions on Information Theory*, vol. IT-27, no. 5, pp. 533–547, September 1981.

[13] N. Wiberg, *Codes and Decoding on General Graphs*, Ph.D. thesis, Linköping University, Linköping, Sweden, 1996.

[14] N. Axvig, K. Morrison, E. Psota, D. Turk, L. C. Pérez, and J. L. Walker, "Average min-sum decoding of LDPC codes," *2008 International Symposium on Turbo Codes and Related Topics*, September 2008.

[15] E. Psota and L. C. Pérez, "Extrinsic tree decoding," in *Proceedings of the 2009 Conference on Information Sciences and Systems*, March 2009.

[16] E. Psota and L. C. Pérez, "LDPC decoding and code design on extrinsic trees," in *Proceedings of the 2009 International Symposium on Information Theory*, June-July 2009.

[17] T. Richardson and R. Urbanke, "The capacity of low-density parity check codes under message-passing decoding," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 599–618, February 2001.

[18] T. Richardson, A. Shokrollahi, and R. Urbanke, "Design of capacity-approaching irregular low-density parity check codes," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 619–637, February 2001.

[19] Xiao-Yu Hu, E. Eleftheriou, and D.-M. Arnold, "Progressive edge-growth Tanner graphs," in *Proceedings of the 2001 Global Telecommunications Conference*, November 2001, vol. 2, pp. 995–1001.

[20] J. Feldman, *Decoding Error-Correcting Codes via Linear Programming*, Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, September 2003.

[21] G. B. Dantzig, *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963.

[22] P. Vontobel and R. Koetter, "Graph-cover decoding and finite-length analysis of message-passing iterative decoding of LDPC codes," To appear in *IEEE Transactions on Information Theory*.

[23] N. Axvig, K. Morrison, E. Psota, D. Dreher, L.C. Pérez, and J. L. Walker, "Analysis of connections between pseudocodewords," *IEEE Transactions on Information Theory*, vol. IT-55, no. 9, pp. 4099–4107, September 2009.

[24] P. O. Vontobel, "Symbolwise graph-cover decoding: Connecting sum-product algorithm decoding and bethe free energy minimization," in *Proceedings of the 46th Annual Allerton Conference on Communication, Control, and Computing*, September 2008.

[25] G. D. Forney, Jr., R. Koetter, F. R. Kschischang, and A. Reznik, "On the effective weights of pseudocodewords for codes defined on graphs with cycles," in *Codes, systems, and graphical models (Minneapolis, MN, 1999)*, vol. 123 of *IMA Vol. Math. Appl.*, pp. 101–112. Springer, New York, 2001.

[26] D. Changyan, D. Proetti, I. E. Telatar, T. J. Richardson, and R. L. Urbanke, "Finite length analysis of low-density parity-check codes on the binary erasure channel," *IEEE Transactions on Information Theory*, vol. 48, pp. 1570–1579, June 2002.

[27] C. Kelley, D. Sridhara, J. Xu, and J. Rosenthal, "Pseudocodeword weights and stopping sets," in *Proceedings of the 2004 IEEE International Symposium on Information Theory*, Chicago, IL, Jun.-Jul. 27-3 2004, p. 150.

[28] D. J. C. MacKay and M. Postol, "Weaknesses of Margulis and Ramanujan-Margulis low-density parity-check codes," *Electronic Notes in Theoretical Computer Science*, 2003.

[29] T. Richardson, "Error floors of LDPC codes," in *Proceedings of the 41^{st} Allerton Conference on Communications, Control, and Computing*, Monticello, Illinois, October 2003.

[30] Y. Lu, C. Méasson, and A. Montanari, "TP decoding," in *Proceedings of the 45th Annual Allerton Conference*, September 2007.

[31] D. Weitz, “Counting independent sets up to the tree threshold,” in *Proceedings of the 38th ACM Symposium on Theory of Compuing*, May 2006.

[32] V. Savin, “Self-correcting min-sum decoding of LDPC codes,” in *Proceedings of the 2008 IEEE ISIT*, Toronto, Canada, Jul. 6-11 2008, pp. 146–150.

[33] X. Wei and A. N. Akansu, “Density evolution for low-density parity-check codes under Max-Log-MAP decoding,” *IEEE Electronic Letters*, vol. 37, pp. 1125–1126, August 2001.

[34] H. Xiao and A. H. Banihashemi, “Improved progressive-edge-growth construction of irregular LDPC codes,” in *Proceedings of the 2004 Global Telecommunications Conference*, November 2004, vol. 1, pp. 489–492.

[35] H. Chen and Z. Cao, “A measure of the girth histogram for improved girth-conditioning construction of LDPC codes,” *IEEE Communications Letters*, vol. 11, no. 4, pp. 340–342, 2007.

[36] L. Decreusefond and G. Zémor, “On the error-correcting capabilities of cycle codes on graphs,” in *Proceedings of the 1994 IEEE International Symposium on Information Theory*, Trondeim, Norway, Jun.-Jul. 27-1 1994, pp. 307–.