Studies on Lowering the Error Floors of Finite Length LDPC codes

A dissertation presented to
the faculty of
the Russ College of Engineering and Technology of Ohio University

In partial fulfillment
of the requirements for the degree
Doctor of Philosophy

Huanlin Li
June 2011

© 2011 Huanlin Li. All Rights Reserved.
This dissertation titled
Studies on Lowering the Error Floors of Finite Length LDPC codes

by

HUANLIN LI

has been approved for
the School of Electrical Engineering and Computer Science
and the Russ College of Engineering and Technology by

__________________________________________

Jeffrey C. Dill
Professor of Electrical Engineering and Computer Science

__________________________________________

Dennis Irwin
Dean, Russ College of Engineering and Technology
ABSTRACT

LI, HUANLIN, Ph.D., June 2011, Electrical Engineering

Studies on Lowering the Error Floors of Finite Length LDPC codes (156 pp.)

Director of Dissertation: Jeffrey C. Dill

Low-density parity-check (LDPC) codes can approach the Shannon limit performance closely, and are becoming one of the most promising channel codes in the Error Control Coding area. The performance of an LDPC code with given length is mainly affected by the sizes of some combinatorial characteristics of its corresponding bipartite graph and the distribution of variable degrees and check node degrees. With the help of the density evolution algorithm, randomly designed LDPC codes, with carefully chosen degree distribution pairs, have been shown to achieve better Shannon capacity performance than their regular counterparts when decoded using the iterative belief propagation (BP) decoding algorithm. However, regular LDPC codes outperform their irregular counterparts in term of their error floors. Therefore, construction of LDPC codes which have both attractive error rate performance and low error floor performance has become an attractive topic.

Currently, optimization of variable node degree distribution and check node degree distribution for an LDPC code with given length, which could help the code to achieve good Shannon limit performance, has been extensively studied. In this dissertation, we demonstrate some approaches of improving the error floor performance for a given LDPC code with desired degree distribution pair. The contribution mainly includes three parts. In the first part, the relationship between cycles and unfavorable combinatorial
characteristics (or trapping set for codes over AWGN channels and stopping set for codes over BEC channels) of LDPC codes is analyzed. The analysis indicates that large girths of LDPC codes could lead to low error floors. Based on this analysis, an approach of designing any individual irregular LDPC code with girth of six is proposed. The second part presents a novel algorithm of enumerating the worst unfavorable combinatorial characteristics with the help of building a searching tree. With the help of the enumeration results of this novel algorithm, the worst unfavorable combinatorial characteristics can be eliminated by refining the parity-check matrices of LDPC codes, which results in the improvement of the error floor performance of LDPC codes based on our simulation results. The above mentioned methods focus on designing LDPC codes with low error floors from the encoder side of the LDPC codes. Another possible approach of lowering the error floors is from the decoder side, which is studied in the third part of this dissertation. In the third part, the trapping sets of LDPC codes are extensively analyzed and a concept of pseudo-cycle is proposed. Based on the analysis, we present an improved decoder, a two-stage decoder, for LDPC codes to enable dealing with the negative influence caused by those unfavorable combinatorial properties of LDPC codes. The simulation results show that the error floors of LDPC codes can be lowered by more than one order of magnitude.

Unlike the current methods for lowering the error floors of LDPC codes, all the approaches proposed in this dissertation can be applied to any individual LDPC code. Based on our simulation results, these approaches can effectively decrease the error floors for any specific LDPC code.
Approved: _____________________________________________________________

Jeffrey C. Dill

Professor of Electrical Engineering and Computer Science
ACKNOWLEDGMENTS

First, I would like to specially thank my dissertation advisor, Dr. Jeffery C. Dill for his dedicated instructions throughout the six years of my PhD study, and for the encouragement, support, patience, and understanding he has been providing to me. Successful completion of this study would not have been possible without his professional guidance.

I would also thank my committee members: Dr. David Matolak, Dr. Savas Kaya, Dr. Maarten Uijt De Haag, Dr. Sergiu Aizicovici, and Dr. Wei Lin for their valuable time and effort on getting my research and graduate training in progress. Their support is precious for my academic and career preparation. Especially, I thank Dr. Matolak and Dr. Kaya for their advice and guidance throughout my research and Dr. Aizicovici for mathematical discussions.

I would like to express my sincere gratitude to Dr. Kevin Crist, my employer who is working as a professor in the Department of Chemical Engineering of Ohio University, for his continuous graduate assistantship for five years. His professional advice and support are valuable.

With this chance, I would like to thank Dr. Zhen Zhu for his advice and help with my study. I am also grateful for my helpful colleagues and friends: Weizheng Huang, Frank Alex, Kamalakar Ganti, Shan Jiang, and Kai-jen Cheng for their suggestions for my research. Without these teammates, I would not have made such progress.
Last, but not least, I would express my deep gratitude to my family: my wife, my daughter and my son for their everlasting love and support; my parents in-law and my parents for their understanding, selfless love, care, encouragement and support.
TABLE OF CONTENTS

Abstract ............................................................................................................................... 3
Acknowledgments ............................................................................................................... 6
List of Tables .................................................................................................................... 12
List of Figures ................................................................................................................... 13
List of Acronyms and Abbreviations ................................................................................ 15
Chapter 1: Introduction ..................................................................................................... 17
  1.1 The Background of LDPC Codes ........................................................................... 17
  1.2 Contributions of This Dissertation .......................................................................... 22
  1.3 Overview of This Dissertation ................................................................................ 24
Chapter 2: Combinatorial Characteristics of LDPC Codes .............................................. 27
  2.1 The Representations of LDPC Codes ..................................................................... 27
    2.1.1 Matrix Representation ...................................................................................... 28
    2.1.1 Bipartite Representation ................................................................................... 29
    2.1.3 Degree Distribution Polynomials ..................................................................... 32
    2.1.4 Set Diagrams .................................................................................................... 32
  2.2 The Message Passing Decoding Algorithm ............................................................ 35
  2.3 The Combinatorial Characteristics of LDPC Codes ............................................... 38
  2.4 Summary ................................................................................................................. 41
Chapter 3: Designing Irregular LDPC Codes with Large Girths ...................................... 43
  3.1 The Conventional Methods of Constructing LDPC Codes ................................. 44
3.1.1 Gallager LDPC Codes ................................................................. 44
3.1.2 MacKay-Neal (MN) LDPC Codes .................................................. 47
3.1.3 Finite Geometry LDPC Codes and Combinatorial LDPC Codes .......... 48
3.1.4 Array Codes .............................................................................. 51
3.1.5 Irregular LDPC Codes ................................................................. 53
3.2 The Filling-and-Splitting Technique .................................................. 54
3.3 The Notation and Background of Finite Geometries ......................... 56
  3.3.1 Galois Field $GF(2^m)$ and Its Construction ................................. 56
  3.3.2 Some Important Properties of Galois Field $GF(2^m)$ ...................... 59
  3.3.3 The Basic Concepts of Euclidian Geometry .................................. 61
3.4 Constructing an Structured Irregular LDPC Code Based on Euclidean Geometry 64
  3.4.1 Generating a Regular Raw Set Diagram Based on Euclidean Geometries ...... 64
  3.4.2 The Wisely Chosen Degree Distribution Pair for an Irregular LDPC Code .... 68
  3.4.3 Refine the Raw Set Diagram ....................................................... 69
  3.4.4 Simulation Results ....................................................................... 72
3.4 Summary ......................................................................................... 77

Chapter 4: A New Approach for Enumerating the Error Patterns in LDPC Codes ........ 78
4.1 The Background of Enumerating the Error Patterns for LDPC Codes .......... 78
4.2 Exhaustively Searching the Error Patterns in LDPC Codes ..................... 80
  4.2.1 Enumerating Stopping Sets for LDPC Codes .................................. 80
  4.2.2 Enumerating Trapping Sets for LDPC Codes ................................. 87
4.3 Computational Complexity of the Searching Algorithm ................. 88
6.1 Conclusions ........................................................................................................... 131

6.2 Future Studies ....................................................................................................... 133

References ....................................................................................................................... 134

Appendix 1: Statistical Results for the Improved Decoder ............................................ 144
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Two presentations of $GF(2^4)$</td>
<td>58</td>
</tr>
<tr>
<td>4.1</td>
<td>Stopping sets in code with length 2304</td>
<td>92</td>
</tr>
<tr>
<td>4.2</td>
<td>Stopping sets with size six in code $H$</td>
<td>94</td>
</tr>
<tr>
<td>4.3</td>
<td>Variable node sets in Stopping Set 1 in Table 4.1</td>
<td>94</td>
</tr>
<tr>
<td>4.4</td>
<td>Variable node sets in Stopping Set 1 after optimizing</td>
<td>95</td>
</tr>
<tr>
<td>4.5</td>
<td>The variable node sets in the original “stopping sets with size 6” in code</td>
<td>98</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Steps block diagram of channel coding process</td>
<td>17</td>
</tr>
<tr>
<td>1.2</td>
<td>The classification of LDPC codes</td>
<td>20</td>
</tr>
<tr>
<td>2.1</td>
<td>The Tanner graph of the matrix $H$</td>
<td>31</td>
</tr>
<tr>
<td>2.2</td>
<td>The set diagram of matrix $H$</td>
<td>34</td>
</tr>
<tr>
<td>2.3</td>
<td>One round of MPA in Tanner graph</td>
<td>37</td>
</tr>
<tr>
<td>3.1</td>
<td>The performance of a Gallager code with length 258 and rate 1/2</td>
<td>45</td>
</tr>
<tr>
<td>3.2</td>
<td>The performance of a Gallager code with length 6000 and rate 1/2</td>
<td>46</td>
</tr>
<tr>
<td>3.3</td>
<td>The performance of a Finite geometry code with length 258</td>
<td>49</td>
</tr>
<tr>
<td>3.4</td>
<td>The performance of an array code with length 2304 and rate 1/2</td>
<td>50</td>
</tr>
<tr>
<td>3.5</td>
<td>The performance of an irregular code with length 258 and rate 1/2</td>
<td>52</td>
</tr>
<tr>
<td>3.6</td>
<td>The performance of an irregular code with length 6000 and rate 1/2</td>
<td>53</td>
</tr>
<tr>
<td>3.7</td>
<td>Splitting-and-filling on a variable node in variable node set graph</td>
<td>55</td>
</tr>
<tr>
<td>3.8</td>
<td>The performance of a regular Geometry LDPC code with length 255</td>
<td>68</td>
</tr>
<tr>
<td>3.9</td>
<td>The performance of a rate 1/2 irregular LDPC code with length 1024</td>
<td>74</td>
</tr>
<tr>
<td>3.10</td>
<td>The performance comparison between a raw irregular LDPC code and its</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>optimized counterpart</td>
<td></td>
</tr>
<tr>
<td>3.11</td>
<td>The performance of a rate 1/2 irregular LDPC code with length 5999</td>
<td>76</td>
</tr>
<tr>
<td>4.1</td>
<td>The searching diagram for codes in Figure 2.1</td>
<td>84</td>
</tr>
<tr>
<td>4.2</td>
<td>The enumerating time cost</td>
<td>85</td>
</tr>
<tr>
<td>4.3</td>
<td>Full splitting-and-filling on variable node of $v_7^l$ in $S'_G$</td>
<td>96</td>
</tr>
<tr>
<td>4.4</td>
<td>Re-apply full splitting-and-filling on variable node of $v_7^l$ in $S'_G$</td>
<td>97</td>
</tr>
</tbody>
</table>
Figure 4.5: The performance for an optimized code $H_2$ compared with other codes introduced in previous chapters .................................................................99

Figure 5.1: Combinatorial characteristics of out-check-nodes ..................115

Figure 5.2: Trapping Set (12, 4) by SPA decoders ..................................119

Figure 5.3: The collected error events .....................................................128

Figure 5.4: The performance of the improved decoder .........................129
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACND</td>
<td>Average Check Node Degree</td>
</tr>
<tr>
<td>APP</td>
<td>A Posteriori Probability</td>
</tr>
<tr>
<td>AVND</td>
<td>Average Variable Node Degree</td>
</tr>
<tr>
<td>AWGN</td>
<td>Additive White Gaussian Noise</td>
</tr>
<tr>
<td>BAWGN</td>
<td>Binary Additive White Gaussian Noise</td>
</tr>
<tr>
<td>BEC</td>
<td>Binary Erasure Channel</td>
</tr>
<tr>
<td>BP</td>
<td>Belief Propagation</td>
</tr>
<tr>
<td>BSC</td>
<td>Binary Symmetric Channel</td>
</tr>
<tr>
<td>CN</td>
<td>Check Node</td>
</tr>
<tr>
<td>ECC</td>
<td>Error Control Codes</td>
</tr>
<tr>
<td>EG</td>
<td>Euclidean Geometry</td>
</tr>
<tr>
<td>EXIT</td>
<td>Extrinsic Information Transfer</td>
</tr>
<tr>
<td>DVB</td>
<td>Digital Video Broadcasting</td>
</tr>
<tr>
<td>GF</td>
<td>Galois Field</td>
</tr>
<tr>
<td>ICNs</td>
<td>In- Check Nodes</td>
</tr>
<tr>
<td>LDPC</td>
<td>Low-Density Parity-Check</td>
</tr>
<tr>
<td>LLR</td>
<td>Log Likelihood Ration</td>
</tr>
<tr>
<td>LR</td>
<td>Likelihood Ration</td>
</tr>
<tr>
<td>MAP</td>
<td>Maximum a Posteriori</td>
</tr>
<tr>
<td>MPA</td>
<td>Message Passing Algorithm</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>NCC</td>
<td>Noisy Channel Coding</td>
</tr>
<tr>
<td>OCNs</td>
<td>Out-Check Nodes</td>
</tr>
<tr>
<td>PAM</td>
<td>Pulse Amplitude Modulation</td>
</tr>
<tr>
<td>SPA</td>
<td>Sum-Product Algorithm</td>
</tr>
<tr>
<td>VN</td>
<td>Variable Node</td>
</tr>
</tbody>
</table>
CHAPTER 1: INTRODUCTION

1.1 The Background of LDPC Codes

Information transmission and storage systems transmit data from information sources to destinations through wired or wireless channels. An important issue regarding these systems is reliability, that is to say, the received data might be corrupted because of the noisy channel. In 1946, Shannon proposed a Noisy Channel Coding (NCC) theorem in his landmark paper [1]. In this paper, Shannon proved that the errors caused by a noisy channel could be decreased to any wanted level if the data are properly encoded at the transmitter side and the received data are wisely decoded at the receiver side. The general idea of this channel coding procedure is shown in Figure 1.1. Error control codes (ECC) are an important class of these channel codes. Since the proof of the NCC theorem by Shannon, many researchers have been looking for good error control codes, which include efficient encoding and decoding procedures, to realize reliable information transmissions.

![Figure 1.1. Block diagram of channel coding process.](image)

Currently, there are several classes of good error control codes which can provide near Shannon limit performance, including RM codes, turbo codes, Low-density Parity-
check (LDPC) codes, etc. Among those near capacity performance channel codes, the class of low-density parity-check codes is the most promising. LDPC codes are also attractive because of their low complexity iterative decoders. The LDPC codes are a class of linear block codes and were first proposed by Gallager in the 1960s [2]. Unfortunately, they were almost ignored in the following 35 years because the hardware technology could not provide effective support for the codes’ complex computation at that time. Until the mid-1990s, MacKay and some other researchers [3] [4] [5] independently rediscovered the LDPC codes and proved their near Shannon limit performance [3] [6].

Like other linear block codes which can be expressed by parity-check matrices, the LDPC codes can be represented by using low-density parity check matrices. In [2], an LDPC code was written as \((n, d_v, d_c)\) where \(n\) is block length of this code, \(d_v\) and \(d_c\) are the numbers of ones in each column and row in the parity check matrix, respectively. From this representation, we can see that for any matrices, the degrees of all the columns are a fixed number, same for all the rows. This kind of LDPC code is conventionally referred to as a regular LDPC code because of its regular row degrees and column degrees of the parity-check matrices. The concept of irregular codes was first proposed in [7] and further investigated in [8] by Luby. In an irregular LDPC code, the column weights and row weights of the parity-check matrix can vary widely. The column weights and row weights of an irregular parity check matrix can be well described by the variable and check node degree distribution polynomials in an irregular code [7]. With the help of density evolution tools [9], Richardson et al. showed that randomly designed LDPC codes
[10], with wisely chosen degree distribution pairs, could achieve much better near Shannon capacity performance at low $E_b/N_0$ region than their regular counterparts when decoded using the iterative belief propagation (BP) decoding algorithm [8] [9]. However, regular LDPC codes outweigh their irregular counterparts in error floor performance in the high $E_b/N_0$ area [11].

Based on the construction methods, the LDPC codes can also be grouped into two other categories: random-like LDPC codes and structured LDPC codes. The random-like codes are usually generated using random computer searching methods according to the requirements of the parity-check codes [2] [3] [4] [7] [8] [9]. These requirements usually focus on the properties of the corresponding Tanner graphs of the LDPC codes [12], which include the girths of Tanner graphs, degree distribution polynomials, and some other combinatorial characteristics. Structured codes might be generated by using finite algebraic [13] [14] [15] or other combinatorial methods [16] [17] [18] [19] [20] [21]. These kinds of codes are called structured LDPC codes because their structures are fixed when their dimensions and lengths are fixed. The structured codes usually can perform better than random-like codes for short block lengths [13] while the random-like codes often perform better in the case of long block lengths. On the other hand, it is hard for random-like codes to predict their minimum distances and short cycles (especially for the possible shortest cycles with length four) in the parity check matrices, which might result in high error floor, even though they can achieve good near capacity performance before the error floor phenomena show up. The general classification of LDPC codes are shown in Figure 1.2.
The LDPC codes can be encoded using their generator matrices, which can be obtained from the corresponding parity-check matrices using the Gaussian elimination method. However, the procedure would become complicated and time-inefficient if the code length is large. In [22], Richardson proposed an efficient approach for encoding the LDPC codes by exploiting the low-density property of their parity-check codes. The encoding procedure can also be completed by using a shift register if an LDPC code has some special structures, such as cyclic or quasi-cyclic [14] [23] [24] [25] [23] [26] [27] [28].

Figure 1.2. The classification of LDPC codes.

In order to achieve good error floor performance, the LDPC codes are usually decoded by using message passing algorithm (MPA) [2] [3] [4] [29]. The MPA decoding is an iterative decoding algorithm with soft input and soft output. If the messages passed in a message passing decoder are continuous reliable probabilities of variable node values, these probabilities can be called beliefs and this decoding algorithm is referred to as the belief propagation algorithm. The BP decoding algorithm is an optimal class of message passing decoding algorithms. The concept of belief propagation is often used in
the Artificial Intelligence area, and it was applied to the decoding of LDPC codes in
1990s [3] [5] [7]. Gallager’s decoding algorithm in [2] is actually a simple version of
the BP algorithm even though it is based on hard decisions. There are also several other
decoding methods [25], such as majority-logic decoding, bit flipping decoding and a
posteriori probability (APP) decoding. Although some of these decoding algorithms have
low decoding complexity compared with MPA, they cannot provide performance as well
as MPA. Therefore, we can find the trade-off among those decoding methods between
error performance and decoding complexity.

For an LDPC code with finite length, an error floor exists [30] [31] [32], which
may be either observable or non-observable [32]. The phenomena of error floors are
mainly attributed to some unfavorable combinatorial characteristics in LDPC codes.
These unfavorable combinatorial characteristics were first called low-weight near
codewords by MacKay in [31], or trapping sets (called stopping sets if codes are over
BEC channels [9] [30] [32]). The existence of such combinatorial characteristics might
severely degrade the error floor performances of LDPC codes [32]. Di, et al. [30] studied
the average error probability of LDPC codes over the binary erasure channels (BECs)
under iterative decoding and demonstrated that the error floor performance of an LDPC
code is largely dependent on nonempty stopping sets, a combinatorial property of its
parity-check matrix. But for LDPC codes over other binary channels, these kinds of
combinatorial characteristics are collected only using the experimental searching
strategies and Important Sampling method [32] [33] [34] [35] [36] [37]. Moreover,
 exhaustively enumerating all such possible combinatorial characteristics is profoundly
unpleasant and sometime infeasible. Therefore, a systematic analysis of such kind of combinatorial characteristics is needed in order to analyze the performance of any specific code or to lower their error floors.

Presently, many researchers are studying improving the performance of LDPC codes, especially the error floor performance. The error floor phenomenon of LDPC codes restrains their application in some systems, such as deep space communication systems and storage systems, which require very low error rates. Lowering the error floors of LDPC codes is extremely attractive for those systems and becoming a hot topic in the field of LDPC codes. This issue will be investigated from several aspects in this dissertation.

1.2 Contributions of This Dissertation

The main subject of this dissertation is to explore approaches for improving both the good near Shannon capacity performance and the error floor performance of LDPC codes. From Figure 1.1, we can see that there are two possible ways to improve the performance of LDPC codes: design LDPC codes with good combinatorial properties at the encoder side and improve the decoders of LDPC codes at the decoder side. This dissertation addresses the approaches of optimizing the performance of LDPC codes on both the encoder and the decoder sides. The main contributions are summarized in the following aspects:

1. A new representation of LDPC codes which is named set diagram is proposed.

The set diagram is convenient for visualizing the structure of an LDPC code and
applying some operations on an LDPC code. Based on the set diagrams of LDPC codes, the combinatorial characteristics of LDPC codes, including cycles, trapping sets, and stopping sets, are investigated.

2. Based on the set diagrams of LDPC codes and the analysis of the combinatorial characteristics, a technique of designing irregular LDPC codes with large girth, named splitting-and-filling, is proposed based on the Euclidean Geometry. These kinds of irregular LDPC codes have both good capacity performance and low error floor performance because they have girths no less than six and wisely chosen degree [10] distribution pairs.

3. A novel and effective algorithm of enumerating the smallest unfavorable combinatorial characteristics (or error patterns) for any individual LDPC code is presented by building a searching diagram similar to a searching tree. At the same time, the computational complexity of this algorithm is analyzed with the help of average degree of the variable (check) node. These error patterns that are enumerated by using this algorithm may greatly degrade the performance of the LDPC codes in the error floor region. Therefore, enumerating all possible error patterns for an individual LDPC codes is extremely useful for analyzing and further improving the codes’ performance.

4. A mathematical analysis method that enables the identification of some significant features of the possible error patterns of LDPC codes over binary additive white Gaussian noise (BAWGN) channels is presented with the help of the beliefs passed in their decoders. Based on the analysis of beliefs passed among
error patterns, an improved decoder is proposed, which can be applied to any individual LDPC code over an AWGN channel. The proposed decoder can effectively deal with the traversable trapping sets and achieve significantly improved error floor performance compared with the current decoders. More importantly, this proposed decoder does not need to have all the possible trapping sets of an LDPC code when dealing with its error floor problem caused by trapping sets.

1.3 Overview of This Dissertation

Many researchers are studying the distributions of unfavorable combinatorial characteristics of LDPC codes and the smallest sizes these characteristics, aiming at getting better error floor performances for the random-like codes while keeping their good capacity performance. By analyzing the stopping sets, the exact expressions for the average error probability of finite-length codes can be obtained [9] [32] [37]. Moreover, tight bounds for the error probabilities of long LDPC codes are achieved by analyzing the asymptotic behaviors of the stopping sets [38]. However, up to now, there are no systematic methods to generate a deterministic irregular parity-check matrix which has given degree distribution pair and is free of some small bad combinatorial characteristics. Targeting at improving the capacity performance and error floor performance, we investigate approaches for designing irregular parity-check matrices with good combinatorial properties by borrowing the merits of the structured parity check matrices which have the determined minimum distances, but no cycles of length four. After
generating good parity-check matrices, two other approaches, from both the encoding side and the decoding side, are proposed to further improve the error floor performance of LDPC codes. The organization of this dissertation is as follows.

Chapter 2 introduces the conventional representation methods of LDPC codes and the current LDPC codes’ classification based on the structures of the parity-check matrices. At the same time, a new representation method, the set diagram, for LDPC codes is proposed. Based on these representations, some combinatorial concepts and their relationships are discussed. These combinatorial characteristics might severely degrade the error floor performance of LDPC codes.

Chapter 3 first states some conventional methods of generating LDPC codes. After that, the Euclidean geometry is briefly introduced. Based on the set diagram presented in Chapter 2, a new technique, called splitting-and-filling, is described. With the help of the Euclidean geometry and this new technique, an approach of designing irregular LDPC codes is proposed. The LDPC codes generated using this method can have both good capacity performance and good error floor performance.

Chapter 4 focuses on searching the combinatorial characteristics of LDPC codes. By building a search diagram similar to a searching tree, a novel method of exhaustively enumerating some combinatorial characteristics is explored. With the help of enumerating results, the parity-check matrices are optimized in their combinatorial properties. At the same time, the simulation results of the optimized LDPC codes are disclosed and discussed.
Chapter 5 investigates the widely used decoding algorithm of LDPC codes, the belief propagation decoding algorithm. By analyzing the beliefs passed in the decoding process, the possible reasons causing the LDPC codes’ error floors are explored. Based on the analysis of the belief propagation decoding process, an improved decoder is developed, which can effectively lower the error floors of LDPC codes. The decoding performance of this improved decoder is shown and also compared with that of the conventional decoders.

Finally, Chapter 6 gives the conclusions of this dissertation and proposes future research relevant to the presented ideas.
CHAPTER 2: COMBINATORIAL CHARACTERISTICS OF LDPC CODES

Low-density parity check codes, originally proposed Gallager in 1962 in [2], have re-attracted researchers’ interest recently in coding theory because of their near capacity performance [3] and their parallel and low complexity decoders [4] [5] [7]. In this chapter, several commonly used methods of representing LDPC codes and the advantages of LDPC Codes are presented. A new representation method proposed by our team is also described, which is convenient for visualizing and analyzing the combinatorial characteristics of LDPC codes. Because the combinatorial characteristics (or error patterns) of LDPC codes are related to the decoding algorithms and transmitting channels, a popular decoding algorithm of LDPC codes, the message passing decoding algorithm, is described in detail. After this, the relationship between different error patterns and their influence on the performance of LDPC codes are discussed.

2.1 The Representations of LDPC Codes

In this section, several conventional representation methods of LDPC codes and the advantages of these codes are introduced. At the same time, a new representation method will be proposed. This new representation method is very useful for visualizing the combinatorial properties of LDPC codes and applying some operations, including splitting a parity-check matrix and searching the combinatorial characteristics of an LDPC code, to a code.
2.1.1 Matrix Representation

As a sub-class of linear error control codes, any individual LDPC code can be expressed by either a generator matrix $G$ or a parity check matrix $H$, where $H$ should be very sparse, i.e., the number of nonzero elements in each matrix should be much smaller than the total elements in this matrix (usually less than 3%). This is the reason behind the name of LDPC codes. For an LDPC code whose parity-check matrix $H$ has $n$ columns and $m$ rows, it can be written as $(n, n-j)$ [9] and its code rate is $(n - m)/n$, where $j$ the number of parity-check equations and equals to $n-m$. In this dissertation, only binary LDPC codes are considered. Therefore, the elements in a $G$ or an $H$ are either 0’s or 1’s.

For any individual LDPC code, its codeword $c$ can be generated by using $\bar{c} = \bar{u}G$ and $\bar{c}$ must satisfy $\bar{c}H^T = 0$, where $\bar{c} = [c_1, c_2, \cdots, c_n]$ is an $n$-tuple codeword, $\bar{u} = [u_1, u_2, \cdots, u_k]$ is the message vector, and where $k = n - m$, and $u_i$ and $c_i \in \{0,1\}$. That is to say, the code bits of each codeword must satisfy a group of parity check equations specified by the nonzero elements in the parity check matrix $H$. Therefore, all codewords form the $k$-dimensional subspace $C$ of vector space $F_2^n$ over the Galois field $GF(2)$, and the subspace $C$ is the null space of $H$. If we apply the row operations to the parity-check matrix $H$ by means of Gaussian Elimination rule, we can get $H$’s systematic format, written as follows.

$$H = [I_{n-k} : P^T]$$

And $G$ can be correspondingly written as

$$G = [P : I_k]$$
where $P$ is the parity array portion of $H$, $I_k$ and $I_{n-k}$ are identity matrices. Because all LDPC codes belong to the class of linear block codes, their minimum distances can be determined and are equal to the minimum column weight in $H$ whose all columns are added to zero [25].

In every Gallager LDPC code, its parity check matrix $H$ has the following structural properties: (1) each row has $\rho$ 1’s and each column has $\gamma$ 1’s, which are also known as row weight and column weight, respectively; (2) no two columns or two rows can have more than one 1’s in common; (3) $\rho$ and $\gamma$ are small compared with the column number and the row number of $H$ respectively, which ensures the sparseness of $H$ [3] [25]. Gallager’s LDPC codes are referred to as regular LDPC codes because of their regular structures in the parity-check matrices. For a regular code, $\rho$ and $\gamma$ are constants and satisfy

$$\rho = \gamma \cdot (n/(n - k)) \quad (2.1)$$

Therefore, its code rate can be written as

$$R = k/n = 1 - \gamma/\rho.$$ 

If the number of 1’s in each row or each column is not constant, this code is called an irregular code. Therefore, the LDPC codes can be classified into two categories based on their column and row weights: regular LDPC codes and irregular LDPC Codes.

### 2.1.1 Bipartite Representation

In 1981, Tanner first represented an LDPC code by using a bipartite graph, now called Tanner graph, in order to demonstrate the iterative decoding process [12]. A
Tanner graph is composed of a set of nodes (or vertices) and a set of edges. The nodes are grouped into two types: variable nodes and check nodes. Each edge can only connect two nodes of different types in the Tanner graph. When the two nodes are connected by an edge, we can also say that this edge is incident with these two nodes. The node's degree is defined as the number of edges that are connected to (or incident to) this node. A cycle is referred to as a closed loop in the Tanner graph and its length is the number of edges in this loop. We will give the accurate definition of cycles later in this chapter. The length of the shortest cycle(s) is called the graph’s girth.

A Tanner graph can be induced from a parity check matrix with \( m = n - k \) rows and \( n \) columns by using the following rules: (1) the \( m \) rows corresponding to the set of parity check constraints form the \( m \) check nodes (or check sum vertices), denoted by \( c_1, c_2, \ldots, c_m \), while the \( n \) columns corresponding to the codeword bits form \( n \) variable nodes (or code bit vertices), denoted by \( v_1, v_2, \ldots, v_n \); (2) there is an edge between a check node \( c_j \) (1 ≤ \( j \) ≤ \( m \)) and a variable node \( v_i \) (1 ≤ \( i \) ≤ \( n \)) if and only if the element \( h_{ji} \) in \( H \) is equal to 1. Therefore, two conclusions can be obtained by investigating the above rules: (1) the degree of a check node (or variable node) is equal to its corresponding row (or column) weight; (2) there is at most one edge between any two nodes. Based on these rules, a Tanner Graph of the following matrix \( H \) can be induced, which is shown in Figure 2.1:

\[
H = \begin{bmatrix}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1
\end{bmatrix}
\]
In this example, the LDPC code is regular because the row weight and column weight are constant, equal to four and two respectively. From the check node degree and the variable node degree in its Tanner graph, we can also conclude that this code is regular. In Figure 2.1, the six bold edges indicate a cycle. In fact, this cycle is one of the shortest ones. Therefore, the girth of the graph is six. The girth is an important combinatorial characteristic of an LDPC code because it may severely affect error floor performance in the iterative decoding process. When constructing LDPC codes, large girths are always desired. The role that the girths play in the LDPC codes will be discussed in detail when we describe the message passing decoding.

![The Tanner graph of the matrix H.](image)
2.1.3 Degree Distribution Polynomials

The degree distribution polynomials were introduced by Dr. Richardson to represent an ensemble of LDPC codes combined with the Tanner graph [9], and they can be used to specify the degree distributions of the variable nodes and check nodes in Tanner graph by using the following format [8] [10]

\[ y(x) := \sum_{i=2}^{d_v} y_i \cdot x^{i-1} \quad \text{for variable nodes} \]  
\[ \rho(x):= \sum_{j=2}^{d_c} \rho_j \cdot x^{j-1} \quad \text{for check nodes} \]

where \( d_v \) and \( d_c \) are the maximum degrees of the variable nodes and check nodes respectively; \( y_i \) and \( \rho_j \) denote the fraction of all edges incident to variable nodes with degree \( i \) and check nodes with degree \( j \) respectively. Based on a pair of degree distribution polynomials and a given code length, we can calculate some parameters of this given LDPC code. Some of parameters that are necessary for generating an irregular LDPC code will be given in Chapter 3.

From this notation, we can see that the degree distribution polynomials describe an ensemble of LDPC codes, but not a specific LDPC. However, this definition is very helpful in expressing a code’s structure and in generating an irregular LDPC code, which will be demonstrated in the following chapter.

2.1.4 Set Diagrams

The set diagram was proposed to represent the LDPC codes in [39] in order to help to analyze and enumerate some combinatorial characteristics of LDPC codes. As described in [39], each set diagram consists of two parts: a variable node set diagram and
a check node set diagram. The variable (check) node set diagram part is composed of a variable (check) node Venn diagram and all variable (check) node sets. Generally speaking, the variable (check) node Venn diagram in a variable (check) node set diagram is mainly used to visualize the operations applied to an LDPC code or to describe the code’s combinatorial characteristics, such as trapping sets, stopping sets and cycles, while the variable (check) node sets are very helpful and convenient for searching the combinatorial characteristic of an LDPC code. Therefore, a variable (check) node Venn diagram and all variable (check) node sets can be separated and either one can be neglected for different purposes. For instance, a variable (check) node Venn diagram can be neglected if we only focus on the real operations or combinatorial characteristic searching. When generating a set diagram for an LDPC code, it is assumed that the girth of this LDPC code is no less than 6 and there is no variable node or check node with degree less than 2. These two assumptions are reasonable for any LDPC codes. These assumptions demonstrate that the intersection of any two variable (check) node sets has at most one element, i.e., there is at most one common element for any two variable (check) node sets in the variable (check) node set diagram [39].

\[ H = \begin{bmatrix}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\end{bmatrix} \begin{bmatrix}
\nu_0 \\
\nu_1 \\
\nu_2 \\
\nu_3 \\
\nu_4 \\
\nu_5 \\
\nu_6 \\
\nu_7 \\
\nu_8 \\
\nu_9 \\
\end{bmatrix} \begin{bmatrix}
c_0 \\
c_1 \\
c_2 \\
c_3 \\
c_4 \\
\end{bmatrix} \]
In order to explain the set diagram clearly, an example of a set diagram [39] is shown in Figure 2.1 which describes an LDPC code represented by the above matrix $H$. With the help the above definition of set diagrams, this example demonstrates that a set diagram is actually composed of two groups of sets: variable node sets and check node sets. Each group of node sets can be represented by a Venn diagram. The group of variable (check) node sets and its corresponding Venn diagrams form the variable (check) node set graph.

\[ v_0 = \{c_0, c_1\}; \quad v_1 = \{c_0, c_2\}; \quad v_2 = \{c_0, c_3\}; \]
\[ v_3 = \{c_0, c_4\}; \quad v_4 = \{c_1, c_2\}; \quad v_5 = \{c_1, c_3\}; \]
\[ v_6 = \{c_1, c_4\}; \quad v_7 = \{c_2, c_3\}; \quad v_8 = \{c_2, c_4\}; \]
\[ v_9 = \{c_3, c_4\} \]

(a) Variable node set graph.

\[ c_0 = \{v_0, v_1, v_2, v_3\}; \quad c_1 = \{v_0, v_4, v_5, v_6\}; \]
\[ c_2 = \{v_1, v_4, v_7, v_8\}; \quad c_3 = \{v_2, v_5, v_7, v_9\}; \]
\[ c_4 = \{v_3, v_6, v_8, v_9\} \]

(b) Check node set graph.

*Figure 2.2* The set diagram of the matrix $H$. 
node set diagram. In the variable (check) node set diagram, each set represents a variable (check) node. The elements of each set are the neighbors of the corresponding variable (check) nodes in the Tanner graph. Equivalently, a variable (check) node set diagram is a series of sets of nonzero elements of corresponding columns (rows) in the parity-check matrix. Therefore, a set diagram can be easily induced from a Tanner graph or a parity-check matrix, and vice versa. To induce an LDPC code from a set diagram, the variable node set diagram is usually used. The check node set diagram is mainly used to assist the variable node set diagram to do some operations on parity-check matrix or search some special combinatorial characteristics for an LDPC code [39].

2.2 The Message Passing Decoding Algorithm

The message passing algorithm is an iterative low complexity decoding algorithm that is used to estimate the codeword sent from the transmitter. The messages passed in the MPA are usually obtained by calculating the a posterior probability (APP) of the received codeword at the receiver side. Assume \( \vec{x} = [x_1, x_2, \ldots, x_n] \) is a transmitted codeword and the corresponding received codeword at the receiver side is \( \vec{y} \), the APP can be written as

\[
P_r = p(x_i = 1 | \vec{y})
\]  

and \( x_i \) can be any bit from \( x_1 \) to \( x_n \). For easy computing, the MPA often uses the likelihood ratios (LR) or log-likelihood ratios (LLR) instead of APP. The likelihood ratios and log-likelihood ratios can be obtained using (2.5) and (2.6) respectively. Using
LLR can make the fixed-point implementation more accurate than using APP because the effect of finite codeword decreases a lot with LLR.

\[ l(x_i) = \frac{p(x_i=0/y)}{p(x_i=1/y)} \]  \hspace{1cm} (2.5)

\[ L(x_i) = \ln[l(x_i)] = \ln[p(x_i = 0/y)/p(x_i = 1/y)] \] \hspace{1cm} (2.6)

When decoding an LDPC code, the message passing algorithm proceeds in this code’s corresponding Tanner graph. As mentioned earlier, the MPA is composed of several rounds or iterations. At each round of this algorithm, there are messages exchanged between variable nodes (VNs) and check nodes (CNs) along the edges in the code’s Tanner Graph, i.e., at each round, each variable node \( v_i \) sends its associated message to each neighboring check node \( c_j \), then each \( c_j \) processes the received messages and sends its processed result back to its connected variable nodes [40]. The processing procedure in each check (or variable) node will be explained in detail in chapter 5. At iteration zero, all check nodes send nothing to their neighbors, and all variable nodes get their processed results only according to the observation (inputs of the decoder). An important restriction on the process of message sending is that a message passed along an edge \( e \) cannot depend on the message previously received along the same edge. Figure 2.3 demonstrates how the messages are exchanged between variable nodes and check nodes at each round of MPA [40].

The message passing decoding algorithm is also called the sum-product algorithm (SPA) because the decoding process is a sum and a product process at the check node side and the variable node side in the Tanner graph, respectively. If the message passing decoding proceeds in the Tanner graph and the messages passed in a message passing
decoder are continuous reliable probabilities of variable node values such as LLR, these probabilities can be called beliefs and this decoding algorithm is known as belief propagation algorithm. The BP decoding algorithms are an optimal or suboptimal class of message passing decoding algorithms. In [9], the BP algorithm was extensively analyzed over a large class of channels with the help of density evolution tools. In Chapter 5, the BP decoding algorithm is introduced in detail and an improved decoder for dealing with the error patterns of the LDPC codes is provided.

![Figure 2.3](image)

*Figure 2.3* One round of MPA in Tanner graph. Squares represent the check node, circles represent variable node, and arrow lines coming from squares are the extrinsic messages received by a variable node. The arrow lines coming from circles are extrinsic messages received by a check.
2.3 The Combinatorial Characteristics of LDPC Codes

From the message exchanging process in MPA, we can see that the MPA would be optimal for a tree-like graph [9]. However, in real applications, it is almost infeasible to build an LDPC code whose factor graph is a tree. The existence of a cycle (see Definition 2.1) with length \(2l\) in a code’s Tanner graph would break the independence restriction of MPA after \(l\) iterations. Therefore, cycles may seriously degrade the performance of an LDPC code and they are treated as bad combinatorial characteristics in Tanner graph. When designing an LDPC code, it would be desirous to enumerate all the cycles and make its girth as large as we can.

Besides the cycles, there are some other unfavorable combinatorial characteristics in LDPC codes which could seriously deteriorate the codes’ performance. These unfavorable combinatorial characteristics are conventionally called low-weight near code words [31], or trapping sets [32](or stopping sets [9] [30] if codes are over binary erasure channels). We generally call these bad combinatorial characteristics error patterns and we use these two terms interchangeably in this dissertation. Because the messages passed in the MPA decoding algorithm are dependent on the channels over which codes are transmitted, we need to investigate those combinatorial characteristics of LDPC codes according to the channel type. The definition of cycles, stopping sets and trapping set based on Tanner graph are given in Definition 2.1, Definition 2.2 and Definition 2.3, respectively.

**Definition 2.1:** [Cycle] In the Tanner Graph, a cycle with length \(2l\) is a closed path composed by \(l\) variable nodes and \(l\) check nodes such that this path can travel to
each of these $2l$ nodes exactly once along the edges, i.e., there are no repeated nodes or edges in this closed path.

**Definition 2.2:** [Stopping Set] If an LDPC code is transmitted over a binary erasure channel, its error pattern can be defined as a set of variable nodes whose neighbors are connected to this set twice or more in the subgraph induced by this set of variable nodes. Such an error pattern is called a *stopping set* [9] [30] [32].

**Definition 2.3:** [Trapping Set] a trapping set $T(v, c)$ is a set of $v$ variable nodes that are not successfully decoded in the decoder, and the subgraph induced by $T(v, c)$ has $c$ check nodes with odd-degrees [32].

Tian et al. proved that each stopping set is composed of several cycles [41] in a Tanner graph in which there is no more than one edge between any check node and variable node. This conclusion can be easily explained with the help of the definitions of cycles and stopping sets defined based on the set diagram [39]. Definition 2.4, Definition 2.5, and Definition 2.6 are the definitions for cycles, stopping set, and trapping set, respectively. Based on Definition 2.4 and the two previously mentioned assumptions of the set diagram, it can be concluded that a cycle of length $2d$ also means that $d$ different variable (check) node sets have exactly $d$ different common elements and each of these $d$ elements is only shared by two variable (check) node sets, which is a sufficient condition for $d$ variable (check) nodes to form a cycle. Definition 2.5 means that every element in these $d$ variable (check) nodes (or in the union of these $d$ variable node sets) is covered by at least two of these $d$ variable node sets. That is to say, there are $d$ different variable node sets and at least $d$ different elements shared by any two of the variable node sets in a
stopping set with size \(d\) (Only when all these \(d\) different variable nodes have degree 2, the number of different common elements is \(d\).) [39]. Finally, we can get the same conclusion as [39] [41]: there must be at least one cycle in any stopping set of an LDPC code [39].

**Definition 2.4:** [Cycle in the set diagram] In the variable (check) node set diagram, a cycle is an enclosed loop formed by a series of sets which meet the requirement that any two of them share exactly one common element.

**Definition 2.5:** [Stopping set in the set diagram] In the variable node set diagram, \(d\) \((d >1)\) variable node sets form a stopping set with size \(d\) if any element in the union of \(d\) variable node sets is an element of the intersection of at least two of these \(d\) variable node sets. \(d\) is called the *stopping distance* or *stopping number*.

**Definition 2.6:** [Trapping set in the set diagram] A trapping set \(T(v, c)\) is defined as a set of \(v\) variable nodes that are not successfully decoded in the decoder, and among all the elements of these \(v\) variable node sets in the set diagram, there are \(c\) elements that are covered by an odd-number of these \(v\) variable node sets.

From the definitions of the trapping sets and stopping sets, we can see that these two error patterns are defined based on the channels over which the codes are transmitted. In chapter 5, we will show that a trapping set usually contains several cycles or *pseudo-cycles*.

The unfavorable combinatorial characteristics in an LDPC code may cause a high error floor [31] [32]. In [30], those combinatorial characteristics which might lead to the error floors of LDPC codes over binary erasure channels were well analyzed. For LDPC
codes over binary erasure channels, their nonempty stopping sets, especially the nonempty smallest stopping sets, play a crucial role in the error floor region [32]. The negative influence of stopping set will be demonstrated in Chapter 4 and Chapter 5. The sizes of the nonempty smallest stopping sets, also named stopping distances or stopping numbers [30] [38] [42], are roughly proportional to the block length of an LDPC code [38] for either a regular code or an irregular code with a given degree distribution pair. The relationship between the performances of the LDPC code ensembles and the stopping set distributions has been deeply investigated in [39] [41] [43] [44].

For the LDPC codes on AWGN channels, the error patterns are referred to trapping sets. The definition of trapping sets are given in Definition 2.6 based on the set diagram. The trapping sets have the similar roles as the stopping sets for codes transmitted over BEC channels. The definition of trapping sets based on Tanner graph and the negative influence of trapping sets on the performance of LDPC codes will be discussed in detail in Chapter 5.

2.4 Summary

From the above analysis, we can see that the error floor performance of an LDPC code can be degraded significantly due to their unfavorable combinatorial characteristics. For a same LDPC code, its error patterns might be different if it is transmitted over a different channel. In chapter 3, we will show several examples of low error floors caused by these unfavorable combinatorial characteristics. Therefore, Eliminating or mitigating the negative influence of these error patterns is desirable for improving the performance,
especially the error floor performance of an LDPC code. In the following chapters, effective approaches will be provided to improve the performance of LDPC codes by decreasing the negative impact of these error patterns from three aspects: designing DPC codes with large girths, eliminating the smallest error patterns in LDPC codes, and designing improved decoders for LDPC codes.
CHAPTER 3: DESIGNING IRREGULAR LDPC CODES WITH LARGE GIRTHS

In [1], Shannon showed that reliable transmission can be built up between a transmitter and a receiver if the date transfer rate is less than the capacity of a channel. And maximum date transfer rate of a specified channel is referred to as the **Shannon limit** or **Shannon Capacity** of this channel for a certain amount of noise (or noise level). In other words, the Shannon limit or Shannon capacity specifies the minimum possible $E_b/N_0$ when reliably transferring data with rate of the channel capacity. LDPC codes are shown as one of the promising channel codes that can approach the Shannon capacity [3]. Using the density evolution algorithm, Richardson *et al.* [10] showed that randomly designed LDPC codes, with wisely chosen degree distribution pairs, could approach Shannon capacity performance closer their regular counterparts when decoded using the iterative belief propagation algorithm. However, regular LDPC codes generally have lower error floors compared with their irregular counterparts [11]. The higher error floor of an irregular LDPC code is mainly attributed to its unfavorable combinatorial characteristics, including the smallest cycles, stopping sets or trapping sets [10] [9] [11] [30] [45]. Based on the relation between stopping set (or trapping set) and cycles, we can deduce that the large girth of an LDPC code can help to increase the size of the smallest error patterns because each stopping sets (or trapping sets) is composed of one or several cycles [39] [41]. Therefore, the stopping (or trapping) distance might be increased and the code’s error floor performance can be improved if we can make the girth of an LDPC code larger.
In this chapter, we will first discuss the commonly used method of designing LDPC codes and show the performance differences between regular and irregular LDPC codes. Then we will introduce a method of generating low-density parity check matrices with girths at least six based on the knowledge of Euclidean geometry. After that, a technique, called splitting and filling, is proposed based on the set diagram. By applying this technique to the generated matrices, new LDPC codes with desired degree distribution pairs and girths of at least 6 can be obtained. These newly-generated LDPC codes can achieve both near Shannon limit performance and good error floor performance.

3.1 The Conventional Methods of Constructing LDPC Codes

In this section, the conventional construction methods for LDPC codes are briefly introduced. At the same time, their advantages and disadvantages will be discussed based on their performances simulated with MATLAB ® 2009b.

3.1.1 Gallager LDPC Codes

Gallager proposed the LDPC codes in the 1960s. In [2], he deeply investigated the performance of LDPC codes and gave us a general design approach for constructing random-like regular codes though not systematically. For a given parity-check matrix with row weight \( \rho \) and column weight \( \gamma \) which are both greater than 1, the parity-check matrix can be written using the following format which is composed of \( \gamma \) sub-matrices, \( H_1, H_2, \ldots, H_\gamma \) [2].
These $\gamma$ sub-matrices have the following structure properties: (1) every row of each sub-matrix has $\rho$ 1s and every column of each sub-matrix has a single 1; (2) each sub-matrix is $\mu \times \rho$, where $\mu = (n - k)/\gamma$; (3) in $H_1$, the $\rho$ 1s in the $i^{th}$ row of $H_1$ are located at

$$H = \begin{bmatrix} H_1 \\ H_2 \\ \vdots \\ H_\gamma \end{bmatrix}$$

Figure 3.1 The performance of a Gallager code with length 258 and rate $\frac{1}{2}$. Parity-check matrix $H$ is $129 \times 258$ with (3, 6). The number of iterations is 100.
columns from \((i - 1) \cdot \rho + 1\) to \(i \cdot \rho\) continuously for \(i = 1, 2, \ldots, \mu\); (4) the other sub-matrices are merely column permutations of \(H_1\). The parity-check matrix generated by the above rules is called Gallager parity-check matrix or Gallager code. The error performance of the Gallager code with length 258 and rate 1/2 is shown in Figure 3.1. In Figure 3.1 the code’s column weight and row weight is 3 and 6, respectively. The number of iterations running the BP decoding algorithm is 100. From this figure we can see that error performance is good though the block length is only 258.

**Figure 3.2** The performance of a Gallager code with length 6000 and rate \(\frac{1}{2}\). The parity-check matrix \(H\) is \(3000 \times 6000\) with \((3, 6)\). The number of iterations is 50.
We also generated a Gallager code with length 6000 with error probability vs. SNR results shown in Figure 3.2. By comparing Figure 3.2 and Figure 3.1, we can see that the near Shannon performance can be better when block length is longer. The Gallager construction method does not purposely avoid forming the cycles of length four. Therefore, the Gallager matrix may contain these kinds of cycles which will severely degrade the performance of iterative decoding. In order to improve the performance of a Gallager code, we should try to eliminate the cycles with length four, i.e., to avoid more than one 1s in any two rows or two columns in the parity-check matrix $H$ when constructing $H$.

3.1.2 MacKay-Neal (MN) LDPC Codes

In the mid 1990s, MacKay et al. independently rediscovered the LDPC codes and noticed the advantages of the sparse parity check matrix, including the codes’ near Shannon performance and low complexity decoding [3]. They generated the sparse parity check matrix pseudo-randomly using the computer searching method in 3 steps: (1) randomly generate one vector which has weight and length equal to $\gamma$ and the row number of $H$ respectively; (2) take the transpose of this vector as one column of the matrix $H$ if there are no cycles of length four after adding this column to the partially constructed matrix; discard this column if there are cycles of length four; (3) go to step (1) to keep generating columns that are different from all the previously generated vectors until all the columns needed for matrix $H$ are obtained. Many of his regular codes
generated in this way and their corresponding simulation results are shown at MacKay’s personal website [46].

3.1.3 Finite Geometry LDPC Codes and Combinatorial LDPC Codes

The Gallager LDPC codes and MN codes are generated randomly or pseudo-randomly by using computer searching, which means these two classes of LDPC codes do not have fixed structure and cannot be generated repeatedly. Therefore, they are referred to as random LDPC Codes. The finite geometry LDPC Codes and combinatorial LDPC codes which will be introduced in the current sub-section have fixed structures and can be generated repeatedly if code lengths and code rates are given. Because of their fixed structure of parity-check matrix, the finite geometry LDPC codes and combinatorial LDPC codes are called structured LDPC codes.

In [13], [14], [25], and [47], the authors proposed a construction method for regular LDPC codes by using the point, line, and hyper-plan concepts of finite Euclidean geometry. The matrices generated using this technique could have cyclic or quasi-cyclic structures, which allow them to be encoded simply via shift-register circuits [24] [25]. Another merit of these matrices is their good combinatorial characteristics, i.e., these matrices are free of cycles of length four. Therefore, the LDPC codes, especially for shorter length LDPC codes, designed using the knowledge of finite geometry can achieve much lower error floors than Gallager LDPC codes and MN LDPC codes of the same length. The simulation results for a finite geometry LDPC code are shown in Figure 3.3.
Note that the curve of finite geometry code in this figure is much steeper than the one in Figure 3.1.

Figure 3.3 The performance of a Finite geometry irregular code with length 258. The parity-check matrix $H$ is $128 \times 255$. And the number of iterations for decoding is 20.

However, the matrices generated based on finite geometry have two obvious disadvantages. One is that the column weights and row weights are much larger than those of the random LDPC codes. The large column weights and row weights can make the LDPC codes harder to meet the low-density (usually no larger than 0.01) requirement. The density of some finite geometry LDPC codes can reach 0.33. The high density is
unfavorable for the decoding complexity because the complexities of the message passing decoding algorithm are largely dependent on the densities of parity-check matrices. Another drawback is the inflexibility of code length. For a finite geometry LDPC code has a fixed code length once the finite geometry is chosen. The length inflexibility may restrain the application of finite geometry LDPC codes. However, this problem can be solved by applying splitting operations to the parity-check matrices. In Section 3.2, we will present one technique that helps to change code length easily.

Figure 3.4 The performance of an array code with length 2304 and rate $\frac{1}{2}$. Its parity-check matrix is $1152 \times 2305$. The number of iterations is 30.
Combinatorial LDPC codes are constructed using the combinatorial mathematics, such as balanced incomplete block designs [25], Steiner systems [16] [17] [48], and Kirkman system [49]. Similar to finite geometry LDPC codes, the combinatorial LDPC codes also have fixed structure. Therefore, they are also a class of structured LDPC codes. Combinatorial LDPC codes behave like other structured LDPC codes. That is, they have lower error floors than random codes, but their capacity performances are not as good as random ones.

3.1.4 Array Codes

The low density parity-check matrices can also be created by using an array of circular shifted identity matrices and zero matrices. This kind of parity-check code is known as an array code. Array codes were first proposed by Blaum [50] and were applied to generate LDPC codes by Fan [18] [51]. Fan’s array code was further modified by Eleftheriou to generate the LDPC codes whose matrices have the following format [52].

\[
H = \begin{bmatrix}
I & I & I & \cdots & I & I & \cdots & I \\
0 & I & \alpha & \cdots & \alpha^{j-2} & \alpha^{j-3} & \cdots & \alpha^{k-2} \\
0 & 0 & I & \cdots & \alpha^{2(j-3)} & \alpha^{2(j-2)} & \cdots & \alpha^{2(k-3)} \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & 0 & I & \alpha^{j-4} & \cdots & \alpha^{(j-1)(k-j)}
\end{bmatrix}
\]

In the above \(H\) matrix, \(k\) and \(j\) are integers satisfying \(k \leq p\) and \(j \leq p\), where \(p\) is a prime number. \(I\) is the \(p \times p\) identity matrix, and 0 is a zero matrix having the same dimension.
as $I$. $\alpha$ is a $p \times p$ permutation matrix whose index denotes a single left-cyclic shift or a single right-cyclic shift.

The array LDPC codes belong to structured LDPC codes because their structures are fixed. The array codes can achieve extremely good performance, including both capacity and error floor performances, if the parameters and code lengths are wisely chosen. The simulation results of one irregular array code with length 2304 are shown in Figure 3.4. However, the array codes have a drawback: the array codes only have selected code lengths and rates.

![Graph](image)

*Figure 3.5* The performance of an irregular code with length 258 and rate $\frac{1}{2}$. Its parity check matrix is $129 \times 258$, and the number of iterations is 100.
3.1.5 Irregular LDPC Codes

Irregular LDPC codes were proposed in [53] and were profoundly investigated in [10] [8] [11]. An LDPC code can be described as a code whose parity-check matrix $H$ has various row weights and column weights. In the Tanner graph of an irregular code, the variable nodes or check nodes have multiple degrees. The random-like irregular codes with wisely chosen degree distribution pairs can be constructed based on the Tanner graph and the degree distribution polynomials, as was introduced in [8] and [10]. The

*Figure 3.6* The performance of an irregular code with length 6000 and rate $\frac{1}{2}$. The parity-check matrix $H$ is $3000 \times 6000$. The number of iterations is 100.
performance of two irregular LDPC codes with lengths 258 and 6000 is shown in Figure 3.5 and Figure 3.6 respectively.

By comparing the LDPC codes shown from Figure 3.1 through Figure 3.3 and from Figure 3.4 to Figure 3.6, we can see that regular LDPC codes have better error floor performance compared with their counterparts while irregular LDPC codes have better Shannon Limit performance. Therefore, it is desirous to design an irregular LDPC code with low error-rate floor, because this kind of LDPC codes can achieve both near Shannon capacity performance and low error floor performance, which are desirous for almost all communication systems and storage systems. In the section 3.3, we will show how to generate such an irregular LDPC code, which has both desired degree distribution pair and large girth.

3.2 The Filling-and-Splitting Technique

Row splitting and column splitting of an LDPC code, which can lower the code’s density and break some of its cycles, were clearly described in [13] based on the matrix representation. Even though an irregular parity-check matrix can be generated by applying row splitting or column splitting to a regular or irregular matrix, it is difficult to construct an irregular code with a given degree distribution pair using rotational and/or random splitting introduced in [13].

In order to make the splitting method more flexible, we developed a new technique, called splitting-and-filling [45], with the help of the set diagram. “The general idea of the splitting-and-filling is as follows: in a set diagram, firstly split a variable
(check) node set, say $s$, into a desired number of variable (check) node sets; then purposely fill the newly-generated variable (check) node sets with the elements of the original set $s$ to make them have given degree distributions and favorite combinatorial property. In the filling step, it is not mandatory to use up all the elements in the original node set $s$. If all the elements in the original node set $s$ are assigned, the splitting-and-filling is called \textit{full splitting-and-filling}, otherwise called \textit{partial splitting-and-filling}.” [45] The full splitting-and-filling can generate a result similar to the splitting method proposed in [13]. Figure 3.7 demonstrates how the partial splitting-and-filling technique works [45], where the variable node $v_i$ is split into two new variable nodes, $v_{i1}$ and $v_{i2}$. In Figure 3.7, the element $c_k$ is never assigned to any newly-born variable node set. Therefore, it is partial splitting-and-filling.

Figure 3.7 Splitting-and-filling on a variable node in variable node set graph.
3.3 The Notation and Background of Finite Geometries

Finite geometry codes were initially studied by Rudolph [54]. After that, many good error control codes, including BCH codes and R-S codes, were generated using the tool of finite geometry. Similarly, LDPC codes can also be generated using the finite geometry tool [13] [55] [56]. The LDPC codes generated by means of finite geometry are conceptually referred to as structured codes because they have fixed structures and can be repeatedly generated. In this section, we will mainly focus on the fundamental knowledge and notations that are related to generating this kind of LDPC codes.

3.3.1 Galois Field $\mathbb{GF}(2^m)$ and Its Construction

In this dissertation, we will mainly focus on the binary coding. Therefore, we will only consider the special case of Galois Field $\mathbb{GF}(q)$, $\mathbb{GF}(2^m)$, where $q$ is a prime or the power of a prime. The field $\mathbb{GF}(2^m)$ is an extended form of the ground field $\mathbb{GF}(2)$, and its elements can be written as

$$F = \{0, 1, \alpha^1, \alpha^2, \cdots, \alpha^l, \cdots, \alpha^{2^m - 2}\}$$

(2.4)

For each element in a field $F$, $\alpha$ for instance, there must be a smallest positive integer $n$ satisfying $\alpha^n = 1$. The integer $n$ is referred to as the order of this element $\alpha$.

For any finite field, such as Galois Field $\mathbb{GF}(2^m)$, if the order of a nonzero element is $(2^m - 1)$, this element is defined as the primitive element of this Galois Field $\mathbb{GF}(2^m)$. Every finite field has a primitive element. For an irreducible polynomial $p(x)$ with degree $m$, if all its coefficients are from $\mathbb{GF}(2)$ and the smallest positive integer $n$ for which $p(x)$ divides $X^n + 1$ is $n = 2^m - 1$, $p(x)$ is called primitive polynomial, i.e.,
\[ X^n + 1 = p(x) \cdot q(x) \]

In the field \( F \) shown in Equation (2.4), \( \alpha \) is a primitive element of \( GF(2^m) \). Therefore the following equation can be achieved

\[ \alpha^{2^m-1} + 1 = p(\alpha) \cdot q(\alpha) = 0 \cdot q(\alpha) = 0 \]

And using modulo-2 addition, we can obtain Equation (2.5).

\[ p(\alpha) = 0 \quad (2.5) \]

Therefore, \( \alpha \) is a root of the primitive polynomial \( p(x) \) with degree \( m \) over \( GF(2) \) [55]. The Equation (2.4) also demonstrates that any nonzero element of \( F \) can be expressed by its primitive elements, which means that field \( GF(2^m) \) can be constructed by using the powers of its primitive element, where the element 1 can be expressed by \( \alpha^0 \).

Next, we will show that any elements in the field \( F \) can be presented by other elements in \( F \). Suppose \( p(X) \) is a primitive polynomial with degree \( m \) and all its coefficients are from \( GF(2) \). It can be written as in (2.6).

\[ p(X) = k_0 + k_1 \cdot X + k_2 \cdot X^2 + \cdots + k_{m-1} \cdot X^{m-1} + k_m \cdot X^m \quad (3.6) \]

where \( k_0, k_1, \ldots, k_{m-1} \) are either 0 or 1 and are from \( GF(2) \). From equation 2.5 we know that \( \alpha \) is a root of \( p(x) \), that is,

\[ p(\alpha) = k_0 + k_1 \cdot \alpha + k_2 \cdot \alpha^2 + \cdots + k_{m-1} \cdot \alpha^{m-1} + k_m \cdot \alpha^m = 0. \]

\( p(X) \) has degree \( m \). Therefore, \( k_m \neq 0 \) and we can get the following expression:

\[ \alpha^m = b_0 + b_1 \cdot \alpha + b_2 \cdot \alpha^2 + \cdots + b_{m-1} \cdot \alpha^{m-1} \]

where \( b_j = k_j / k_m \) is also from \( GF(2) \), and \( j \) is from 0 to \( m-1 \).

Because any element can be expressed by \( \alpha \), any element in \( F \) can also be denoted with the following format
\[ a^i = b_{i,0} + b_{i,1} \cdot \alpha + b_{i,2} \cdot \alpha^2 + \cdots + b_{i,m-1} \cdot \alpha^{m-1} \quad (3.7) \]

In (2.7), if \( i < m \) only \( b_{i,i} \) is 1 and other coefficients are zeros. If \( i \geq m \), \( a^i \) can be derived by \( \alpha^m \cdot \alpha^{i-m} \), where \( 0 \leq i - m < m \). Therefore, the \( b_{i,j} \) can be obtained by the polynomial multiplication of \( \alpha^m \) and \( \alpha^{i-m} \).

<table>
<thead>
<tr>
<th>( \alpha )</th>
<th>( \alpha )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha^0 = 1 )</td>
<td>(0000)</td>
</tr>
<tr>
<td>( \alpha^1 )</td>
<td>(0100)</td>
</tr>
<tr>
<td>( \alpha^2 )</td>
<td>(0010)</td>
</tr>
<tr>
<td>( \alpha^3 )</td>
<td>(0001)</td>
</tr>
<tr>
<td>( \alpha^4 = 1 + \alpha )</td>
<td>(1100)</td>
</tr>
<tr>
<td>( \alpha^5 = \alpha \cdot (1 + \alpha) = \alpha + \alpha^2 )</td>
<td>(0110)</td>
</tr>
<tr>
<td>( \alpha^6 = \alpha^2 \cdot (1 + \alpha) = \alpha^2 + \alpha^3 )</td>
<td>(0011)</td>
</tr>
<tr>
<td>( \alpha^7 = \alpha^3 \cdot (1 + \alpha) = \alpha^3 + \alpha^4 = 1 + \alpha + \alpha^3 )</td>
<td>(1101)</td>
</tr>
<tr>
<td>( \alpha^8 = \alpha \cdot \alpha^7 = \alpha \cdot (1 + \alpha + \alpha^3) = 1 + \alpha^2 )</td>
<td>(1010)</td>
</tr>
<tr>
<td>( \alpha^9 = \alpha \cdot \alpha^8 = \alpha \cdot (1 + \alpha^2) = \alpha + \alpha^3 )</td>
<td>(0101)</td>
</tr>
<tr>
<td>( \alpha^{10} = \alpha \cdot \alpha^9 = \alpha \cdot (\alpha + \alpha^3) = 1 + \alpha + \alpha^2 )</td>
<td>(1110)</td>
</tr>
<tr>
<td>( \alpha^{11} = \alpha \cdot \alpha^{10} = \alpha \cdot (1 + \alpha + \alpha^2) = \alpha + \alpha^2 + \alpha^3 )</td>
<td>(0111)</td>
</tr>
<tr>
<td>( \alpha^{12} = \alpha \cdot \alpha^{11} = \alpha \cdot (\alpha + \alpha^2 + \alpha^3) = 1 + \alpha + \alpha^2 + \alpha^3 )</td>
<td>(1111)</td>
</tr>
<tr>
<td>( \alpha^{13} = \alpha \cdot \alpha^{12} = \alpha \cdot (1 + \alpha + \alpha^2 + \alpha^3) = 1 + \alpha^2 + \alpha^3 )</td>
<td>(1011)</td>
</tr>
<tr>
<td>( \alpha^{14} = \alpha \cdot \alpha^{13} = \alpha \cdot (1 + \alpha^2 + \alpha^3) = 1 + \alpha^3 )</td>
<td>(1001)</td>
</tr>
</tbody>
</table>

Based on the expression (2.7), we can also use the following \( m \)-tuple to represent the elements \( a^i \), \( 0 \leq i \leq 2m - 1 \).

\[ (b_{i,0}, b_{i,1}, b_{i,2}, \cdots, b_{i,m-1}) \quad (3.8) \]
The elements of this $m$-tuple are induced from the coefficients of polynomial $p(X)$. Therefore, $p(X)$ is referred to as the generator polynomial of this $m$-tuple.

**Example 3.1** For $m=4$ and the primitive polynomial $p(X) = 1 + X + X^4$, we can get two other representations for the elements of Galois Field $GF(2^4)$: one is based on the polynomial representation shown in (2.7) and the other is based on its corresponding 4-tuple representation shown in (2.8). They are both shown in table 3.1. And this field can be expressed by its primitive element $\alpha$ as follows.

$$F = \{0, 1, \alpha^2, \cdots, \alpha^i, \cdots, \alpha^{14}\}$$

### 3.3.2 Some Important Properties of Galois Field $GF(2^m)$

If $\beta$ is a nonzero element of Galois field $GF(2^m)$, this field $GF(2^m)$ can also be expressed as

$$F = \{0, 1 \cdot \beta, \alpha^2 \cdot \beta, \cdots, \alpha^i \cdot \beta, \cdots, \alpha^{2^m-2} \cdot \beta\}$$

Therefore, we can get

$$(1 \cdot \beta) \cdot (\alpha \cdot \beta) \cdots (\alpha^{2^m-2} \cdot \beta) = 1 \cdot \alpha \cdots \alpha^{2^m-2}$$

So we finally obtain

$$\beta^{2^m-1} = 1, \text{ or } \beta^{2^m-1} + 1 = 0$$

That means $\beta$ is a root of polynomial $q(X)$ over $GF(2)$, where $q(X) = X^{2^m-1} + 1$. Because $\beta$ is chosen randomly from nonzero element of $GF(2^m)$, it can be concluded that any nonzero element of $GF(2^m)$ is a root of the above polynomial $q(X)$. Therefore, the following theorem is obtained [55].
**THEOREM 3.1** All the nonzero elements of Galois field $GF(2^m)$ form all the roots of the polynomial $q(X) = X^{2^m - 1} + 1$ over $GF(2)$. In other words, all the elements of $GF(2^m)$ form all the roots of polynomial $X^{2^m} + X$.

For any element $\beta$ in $GF(2^m)$, there may exist a polynomial $\varphi(X)$ with degree less than $2^m$ over $GF(2)$ satisfying $\varphi(\beta) = 0$. If $\varphi(X)$ is the polynomial with the smallest degree that satisfies $\varphi(\beta) = 0$, this polynomial is called the *minimum polynomial of $\beta$*. Apparently, the minimum polynomial is irreducible.

There are two other theorems [55] about the minimum polynomial. And they are useful for searching the generator polynomials of the cyclic codes and the structured LDPC codes. The proof of these theorems can be found in Chapter 2 of [25].

**THEOREM 3.2** If $e$ is the smallest integer no less than zero and it is satisfying $\beta^{2^e - 1} = 1$, where $\beta$ is an element of $GF(2^m)$, the minimum polynomial $\varphi(X)$ of $\beta$ has the following format

$$\varphi(X) = \prod_{i=0}^{e-1} (X + \beta^{2^i})$$  \hspace{1cm} (3.9)

**THEOREM 3.3** If $\varphi(X)$ is the minimum polynomial of $\beta$ in $GF(2^m)$ with degree $e$, $e$ is the smallest integer satisfying $\beta^{2^e - 1} = 1$ and $e \leq m$.

For the generator polynomial of cyclic code, we have the following theorem to obtain its degree and other code polynomials in this cyclic code [25]:

**THEOREM 3.4** For an $(n, k)$ cyclic code, the nonzero code polynomial with minimum degree has degree equal to the number of parity check bits, $n-k$. And this polynomial is also called *generator polynomial*, denoted as $g(X)$. All the other code polynomials are
the multiple of the generator polynomial. The generator polynomial has the format as the following

$$g(X) = 1 + g_1X + g_2X^2 + \cdots + g_{n-k-1}X^{n-k-1} + X^{n-k}$$

### 3.3.3 The Basic Concepts of Euclidian Geometry

In the combinatorial mathematic field, the $m$-dimension Euclidian geometry [16, 34], denoted by $EG(m, 2^s)$, can be built from $2^m$ $m$-tuples (or vectors) $\mathbf{v} = (b_0, b_1, \cdots, b_{m-1})$, where $b_0, b_1, \cdots, b_{m-1}$ are from the Galois field $GF(2^s)$. There is a one-to-one correspondence between each $m$-tuple and each point in Euclidian geometry $EG(m, 2^s)$. The origin of $EG(m, 2^s)$ is mapped by the $\theta$ $m$-tuple, $(0, 0, \cdots, 0)$. Some points in $EG(m, 2^s)$ can form lines, planes or hyperplanes, which are also named 1-flat, 2-flat, or $\mu$-flats ($2^2 < \mu < m$) [25] respectively.

A line that contains a point $V_0$ can be described as

$$\{\mathbf{v}_0 + \beta \mathbf{v} : \beta \in GF(2^s)\}$$  \hspace{1cm} (3.10)

where $\mathbf{v}_0$ and $\mathbf{v}$ are two points in $EG(m, 2^s)$ and linearly independent, plus $\mathbf{v} \neq 0$. If $\mathbf{v}_0$ and $\mathbf{v}$ are linearly dependent on each other, we can have $\beta' \mathbf{v}_0 + \beta \mathbf{v} = 0$, which means that it is a line passing through the origin. All the lines that pass through the origin can also be expressed as $\{\beta \mathbf{v} : \beta \in GF(2^s), \text{ and } \mathbf{v} \neq 0\}$, where $\mathbf{v}$ is in $EG(m, 2^s)$. Obviously, there are $2^s$ $\beta$s in $GF(2^s)$. Therefore, there are altogether $2^s$ points on each line. Any two lines can only intersect at one point or be parallel. This can be proven as follows: let $\mathbf{v}_1$ and $\mathbf{v}_2$ be two linearly independent points in $EG(m, 2^s)$ and also independent on $\mathbf{v}_0$; then two lines can be described as $\{\mathbf{v}_0 + \beta \mathbf{v}_1\}$ and $\{\mathbf{v}_0 + \beta \mathbf{v}_2\}$ correspondingly; and
apparently they intersect at the point $\bar{v}_1$ (set $\beta=0$). Assume they also intersect at another point besides $\bar{v}_1$, then there exist $\beta_1$ and $\beta_2$ in $GF(2^s)$ satisfying

$$\bar{v}_0 + \beta \bar{v}_1 = \bar{v}_0 + \beta \bar{v}_2$$

Therefore, we can get $\beta_1 \bar{v}_1 - \beta_2 \bar{v}_2 = 0$, i.e., $\bar{v}_1$ and $\bar{v}_2$ are linearly dependent on each other. This contradicts with our assumption that $v_1$ and $v_2$ are two linearly independent points. Note: if $\bar{v}_1$ and $\bar{v}_2$ are linearly dependent points in $EG(m, 2^s)$, and $\{\bar{v}_0 + \beta \bar{v}_1\}$ and $\{\bar{v}_0 + \beta \bar{v}_2\}$ represent the same line. Because each line has $2^s$ points and there are $2^{ms}$ points altogether in $EG(m, 2^s)$, for any given $\bar{v}_0$, there exist [25]

$$\frac{2^{ms-1}}{2^s-1}$$  (3.11)

lines passing through it.

If any two different lines do not intersect, they must be parallel. For instance, there is no any common point between these two lines, $\{\bar{v}_0 + \beta \bar{v} : \beta \in GF(2^s)\}$ and $\{\beta \bar{v} : \beta \in GF(2^s), \bar{v} \neq 0\}$; therefore they are parallel with each other. For any given line $l$ in $EG(m, 2^s)$, we can find $(2^{(m-1)s} - 1)$ lines that are parallel to $l$. All these $2^{(m-1)s}$ lines that are parallel to each other are collectively called a parallel bundle [25]. That is, all these $2^{(m-1)s}$ lines do not have any point in common. Based on the number of lines in a parallel bundle and the number of lines intersecting at each point, we can easily calculate the number of lines [25] over $EG(m, 2^s)$:

$$\frac{(2^{(m-1)s-1})(2^{ms-1})}{2^s-1}$$  (3.12)
A line introduced above are also known as a 1-flat in $EG(m, 2^s)$. The more general case is the $\mu$-flat (or the $\mu$-dimensional hyper-plane) ($1 \leq \mu < m$). Like a line, a $\mu$-flat $p$ can be defined as

$$\{\vec{v}_0 + \beta_1 \vec{v}_1 + \beta_2 \vec{v}_2 + \cdots + \beta_\mu \vec{v}_\mu : \beta_j \in GF(2^s), 1 \leq j \leq \mu\}$$

and $\vec{v}_0, \vec{v}_1, \cdots, \vec{v}_\mu$ are points in $EG(m, 2^s)$ and are linearly independent on each other. Similar to a line, this $\mu$-flat passes through $\vec{v}_0$. If we can find a point $\vec{v}_{\mu+1}$ that is not in the $\mu$-flat $p$, we can further extend $p$ to a $(\mu+1)$-flat shown as follows:

$$\{\vec{v}_0 + \beta_1 \vec{v}_1 + \beta_2 \vec{v}_2 + \cdots + \beta_\mu \vec{v}_\mu + \beta_{\mu+1} \vec{v}_{\mu+1} : \beta_j \in GF(2^s), 1 \leq j \leq (\mu + 1)\}$$

Obviously, this $(\mu+1)$-flat have all the points of $\mu$-flat $p$. Because a $(\mu+1)$-flat is a general case of a line, we can get some similar properties of $(\mu+1)$-flat to lines. For instance, there are

$$\frac{2^{(m-n)s-1}}{2^s-1}$$

$(\mu+1)$-flats in $EG(m, 2^s)$ that intersect on $\mu$-flat $p$ [25] [55]. Based on the concept of $\mu$-flat, we can generate LDPC codes with some special characteristics. In this dissertation, we are only interested in the girths of LDPC codes [13]. Therefore, our raw LDPC codes can be generated using only line concept.

From what has been discussed about the $m$-dimension Euclidian geometry $EG(m, 2^s)$, we can easily get the idea that all the elements in the Galois field $GF(2^{ms})$ can have a one-to-one correspondence to the elements in an $EG(m, 2^s)$. And this idea can be proven: suppose $\alpha$ is a primitive element of Galois field $GF(2^{ms})$, then all $2^{ms}$ elements in $GF(2^{ms})$ can be written as follows

$$F = \{\alpha^0, 0, \alpha^1, \alpha^2, \cdots, \alpha^{2^{ms}-2}\}$$
From expression (3.7), we know that each element in $GF(2^{ms})$ can be presented as
\[ a^i = b_{i,0} + b_{i,1} \cdot \alpha + b_{i,2} \cdot \alpha^2 + \cdots + b_{i,m-1} \cdot \alpha^{m-1} \]
or as a $m$-tuple like (3.8)
\[ (b_{i,0}, b_{i,1}, b_{i,2}, \cdots, b_{i,m-1}) \]
where $b_{i,j}$ comes from the subfield $GF(2^s)$ of $GF(2^{ms})$ for $0 \leq j < m$. So all $2^{ms}$ elements in $GF(2^{ms})$ and all $2^{ms}$ points in $EG(m, 2^s)$ can form a one-to-one correspondence. Therefore, we can regard the Galois field $GF(2^{ms})$ as the Euclidian geometry $EG(m, 2^s)$ and vice versa.

3.4 Constructing an Structured Irregular LDPC Code Based on Euclidean Geometry

In this section, an approach of generating a raw set diagram based on Euclidean geometries is proposed. By investigating the generating process, we can see that this raw set diagram has girth no less than 6. When simulating this raw LDPC code with MATLAB, this code shows very a very low error floor compared the previously discussed regular codes due to its good combinatorial structure.

3.4.1 Generating a Regular Raw Set Diagram Based on Euclidean Geometries

A set diagram (or LDPC code) can be algebraically generated based on the points and lines of Euclidian geometry. In an $m$-dimension Euclidian geometry $EG(m, 2^s)$, there are $n = 2^{ms} - 1$ points (Though there are $n = 2^{ms}$ points in $EG(m, 2^s)$, we usually do not use the origin point when constructing the set diagram). According to expression (2.12), there are $J = \frac{2^{(m-1)s}(2^{ms} - 1)}{2^s - 1}$ lines [25] in this $EG(m, 2^s)$. Based on (3.13), there are
lines intersecting at one common point and
\[ \rho = 2^s \quad (3.15) \]
points on each line [25] [55]. For any two points, there is one and only one line
contecting them. We also know that any two lines either have only one common point or
they are parallel. Based on these properties of the \( EG(m, 2^s) \), we can construct a set
diagram without cycles of length four. Suppose
\[ \vec{v} = (v_0, v_1, \ldots, v_{n-1}) \quad (3.16) \]
is a \( n \)-tuple over the binary Galois field \( GF(2) \) and \( n = 2^{ms} - 1 \). At the same time,
assume \( \alpha \) is a primitive element of the \( GF(2^{ms}) \). In this \( n \)-tuple \( V \), each component \( v_i \) can
be mapped from a point of \( EG(m, 2^s) \), \( \alpha^i \), in the one-to-one correspondence for every \( i \),
where \( 1 \leq i < n \) [25] [55]. When using a \( V \) to represent a line that does not pass through
the origin, the value of \( v_i \) can be set using this rule: if \( v_i \) is contained in a line, the value
of \( v_i \) is set to 1 in the corresponding \( \vec{v} \); otherwise, it is set to 0. The vector \( \vec{v} \) used to
represent a line is known as the incidence vector of this line [25].

Using the entire incidence vectors in \( EG(m, 2^s) \), we can form a raw set diagram.
This raw set diagram can be further carved into an irregular code with good
combinatorial characteristics: use each point to represent a variable node set and use the
lines intersected at each point to represent its elements; at the same time, each check node
set is represented by a line and its elements are the points on the corresponding line; a
raw regular set diagram can be generated. Apparently, each variable node set has \( \gamma \)
elements and each check node has \( \rho \) elements, where \( \gamma \) and \( \rho \) are shown as expression
(3.14) and (3.15) respectively. If the entire incidence vectors in $EG(m, 2^s)$ are used, there are $n = 2^{ms} - 1$ variable node sets in variable node set diagram and $J = \frac{2^{(m-1)s}(2^{ms}-1)}{2^s-1}$ check node sets in check node set diagram. The construction process and the performance of a raw set diagram (raw LDPC code) were explained in Example 3.1.

**Example 3.2** Construct a rate 1/2 raw LDPC code based on $EG(2, 2^4)$ and show its performance.

**Solution:** In $EG(2, 2^4)$, $m = 2$ and $s = 4$. The length of this raw LDPC code is $n = 2^{ms} - 1 = 255$ (not including the origin). From (3.12), we know that the number of lines in this geometry is $J - \gamma = \frac{2^{(m-1)s}(2^{ms}-1)}{2^s-1} - \frac{2^{ms}-1}{2^s-1} = 272 - 17 = 255$. Because we need to construct a code with rate $\frac{1}{2}$, we only need to collect 128 lines from $J$. To generate the $GF(2^8)$, we assume $\alpha$ is the primitive element in $GF(2^8)$ and $p(x) = 1 + x^2 + x^3 + x^4 + x^8$ is the primitive polynomial [34]. Therefore, from $p(\alpha) = 0$, we can get

$$\alpha^8 = 1 + \alpha^2 + \alpha^3 + \alpha^4$$

(3.17)

Using the method of Example 3.1, we can easily generate all the elements of $GF(2^8)$:

$$F = \{0, \alpha^0, \alpha^1, \alpha^2, \ldots, \alpha^{254}\}.$$  

This format is similar to the expression (3.16) in section 3.3.1. Therefore, all the elements in $F$ can be mapped to a point in $EG(2, 2^4)$ by one-to-one correspondence. Therefore, we have obtained all the points of $EG(2, 2^4)$. 
Next, the lines in $EG(2, 2^4)$ will be generated using the format of (3.10). The Galois field $GF(2^s)$, where $s=4$, can be written as $\{0, 1, \beta, \beta^2, \cdots, \beta^{14}\}$. Because the maximum power of $\beta$ is 14, the value of $\beta$ can be obtained either using the minimal polynomial of $\beta$ [55] or simply using $\beta = \frac{\alpha^{25}}{\alpha^{15}} = \alpha^{17}$ because $\beta$ needs to satisfy $\beta^{255} = 1$. The following shows how to calculate a line that passes through point $\alpha^{25}$, where $\alpha^{25} = 1 + \alpha$ by applying (3.17) three times.

$$
\begin{align*}
\alpha^{25} + 0 \cdot \alpha &= \alpha^{25} \\
\alpha^{25} + 1 \cdot \alpha &= 1 + \alpha + \alpha = 1 = \alpha^{255} \\
\alpha^{25} + \beta \cdot \alpha &= \alpha^{25} + \alpha^{17} \cdot \alpha = \alpha^{25} + \alpha^{18} = \alpha^{130} \\
\alpha^{25} + \beta^2 \cdot \alpha &= \alpha^{25} + \alpha^{34} \cdot \alpha = \alpha^{25} + \alpha^{35} = \alpha^{46} \\
&\vdots \\
\alpha^{25} + \beta^{14} \cdot \alpha &= \alpha^{25} + \alpha^{239} \cdot \alpha = \alpha^{25} + \alpha^{239} = \alpha^{141}
\end{align*}
$$

Finally, all the points that passing through point $\alpha^{25}$ can be obtained:

$$
\{\alpha^9, \alpha^{25}, \alpha^{32}, \alpha^{46}, \alpha^{74}, \alpha^{129}, \alpha^{130}, \alpha^{149}, \alpha^{201}, \alpha^{207}, \alpha^{211}, \alpha^{237}, \alpha^{240}, \alpha^{242}, \alpha^{255}\}
$$

Similarly, the points on other lines can be achieved. Based on the concept of incidence vectors shown in (3.16), each line can be written as a vector whose elements are 1s and 0s. By randomly selecting the 128 incidence vectors from the entire 256 lines, we can form a $128 \times 256$ matrix. This matrix can be used as the raw set diagram or raw parity-check matrix. Figure 3.8 shows the performance of such a raw LDPC code by simulating on MATLAB. From Figure 3.8, we can see this raw code has an extremely low error floor. This good error floor performance is attributed to its good combinatorial structures. Therefore, if we generate an irregular LDPC code based on the combinatorial
structure of this raw LDPC code, the newly-generated irregular code should have better combinatorial characteristics, such as large girth, compared with other randomly generated codes.

![Figure 3.8](image)

*Figure 3.8* The performance of a regular Geometry LDPC code with length 255. The parity check matrix is $128 \times 255$, and the number of iterations is 20.

### 3.4.2 The Wisely Chosen Degree Distribution Pair for an Irregular LDPC Code

An LDPC code with a carefully chosen degree distribution pair can closely approach the capacity of a channel. In [10], Richardson optimized the degree distribution pairs for different code lengths over various channels using the hill-climbing method at
both the local and the global area. In this dissertation, we will directly adopt this kind of
degree distribution pairs proposed in [10] according the code lengths and channels.

As discussed in Chapter 2, some crucial parameters of irregular LDPC codes can
be calculated based on degree distribution polynomials. Given a code length $n$ and a pair
of degree distribution (variable and check node degree distribution) polynomials, $\gamma(x)$ and
$\rho(x)$ shown in expression (2.2) and (2.3) respectively, the number of nonzero elements in
the irregular parity-check matrix can be calculated using the Equation (3.18) given in
[10] [45].

$$E = n \cdot \sum_{i=2}^{2^s \cdot 2^{m_s-1} / 2^{s-1}} \int_0^1 \gamma(x) dx = n \cdot \frac{1}{\int_0^1 \gamma(x) dx}$$  \hspace{1cm} (3.18)

When generating a previously discussed raw set diagram, the parameters $E$ and $n$ need to
be calculated in order to choose a proper $m$-dimension Euclidean Geometry over $GF(2^s)$. A loose condition [45] for $m$ and $s$ is given in expression (3.19), where $E$ can be
calculated using equation (3.18) and $n$ is the block length of the LDPC code.

$$2^s \cdot \frac{2^{m_s-1}}{2^{s-1}} \geq 2E \text{ and } 2^{m_s} < n$$ \hspace{1cm} (3.19)

3.4.3 Refine the Raw Set Diagram

After obtaining the raw set diagram, our next target is to apply the splitting-and-
filling technique to refine this set diagram in order to create an irregular LDPC code with
a given degree distribution pair. There are three steps to complete the refining process.
First, we need to calculate the number of variable (check) nodes for each variable (check)
degree in this code. Given the degree distribution pair

$$\gamma(x) = \sum_{i=2}^{d_v} y_i \cdot x^{i-1} \text{ for variable node}$$
and

\[ \rho(x) = \sum_{i=2}^{d_e} \rho_i \cdot x^{i-1} \quad \text{for check nodes} \]

which are also shown in (2.2) and (2.3), the number of variable nodes with degree \( i \) can be obtained by using the expression (3.20). Similarly, the number of check nodes with degree \( k \) can be calculated by using the expression shown in (3.21) [10] [45].

\[ N_{vi} = n \cdot \frac{\gamma_{i}/i}{\int_{0}^{1} \gamma(x)dx} \quad (3.20) \]

\[ N_{ck} = n \cdot \frac{\rho_{k}/k}{\int_{0}^{1} \rho(x)dx} \quad (3.21) \]

where \( n \) is the code length, and \( N_{vi} \) and \( N_{ck} \) are the number of variable nodes with degree \( i \) and the number of check nodes with degree \( k \) respectively. In the second step, we need to carve away the redundant nonzero elements in its corresponding parity-check matrix by applying the partial splitting-and-filling operation to all sets in either the check node set diagram or in the variable node set diagram. The rule of choosing variable node set diagram or check node set diagram in this step is shown in the Refine Strategy below. Suppose we choose to apply partial splitting-and-filling operations to all check node sets in the check node set diagram here. After this second step, a new check set diagram \( S_{G1} \) is generated and all check node sets in \( S_{G1} \) should have the desired degrees. In the third step, the full splitting-and-filling operation to \( S_{G1} \) is involved. After applying the full splitting-and-filling operation, another set diagram, say \( S_{G2} \), corresponding to the above newly-generated diagram \( S_{G1} \) is obtained so that all sets in \( S_{G2} \) have given degrees while all degrees of the sets in \( S_{G1} \) are kept unchanged [45].
Refine Strategy: “The set diagram to be refined in step 2 should have the smaller spread of the degree distribution distances.” [45]

As we mentioned earlier, the Refine Strategy should be to choose the target diagram used in step 2 in order to make the third step successfully proceed. The Refine Strategy is made based on the common sense, i.e., the more diversity of the degrees, the more flexibility of applying the full splitting-and-filling in $S_{G2}$. For an irregular code, its near capacity performance can be improved if variable nodes have high degrees but check nodes have low degrees [8]. However, there is a constraint that the variable nodes and the check nodes must have the same total number of edges. In order to balance the high degrees of some variable nodes, some variable nodes have to have very low degrees, such as degree 2 and degree 3 in most finite length irregular codes [10] [45]. Therefore, the variable node sets usually have a large spread of degree distances. According to the above refine strategy, we usually apply partial splitting-and-filling operations to the raw check node set diagram first [45].

In the third step, even though there is a large spread of the degree distribution distances in variable node set diagram, we cannot guarantee to make the refining process directly proceed successfully by applying the full splitting-and-filling on $S_{G2}$ at one time. In order to make step 3 work smoothly, we usually apply two rounds of full splitting-and-filling operations to $S_{G2}$. At the first round, we need to check whether the degree of each node set in $S_{G2}$ can be written as the sum of some/all desired degrees or not. If not, we have to go back to step one and re-create a new set to replace this set in $S_{G1}$. If yes, an intermediate degree $d_{inter}$, which is the least common multiple of the two smallest
degrees, is adopted. “After that, the full splitting-and-filing operation is applied to each node set in $S_{G2}$'. During the generating process, the sets with higher degrees should have higher priority. The second round mainly targets at splitting the sets with degree $d_{inter}$ and make all the sets have the right degrees.” [45]

In step 3, a set diagram $S_G$ with the given degree distribution pair can be successfully achieved after applying the splitting-and-filling operation to each set of $S_{G2}'$. As mentioned in section 3.3.2, the raw set diagram has girth of at least six. At the same time, there is no possibility for the splitting-and-filling process to form any new cycles. In contrast, it may break some cycles in the raw set diagram. Therefore, set diagram $S_G$ has not only girth at least six but given degree distribution pair. In Chapter 4, an approach for enumerating some specific combinatorial characteristics in the newly-generated set diagram will be proposed. Based on the enumeration result, the elements of each set in step 3 can be purposely assigned to the newly-generated sets in the filling procedure. Therefore, the combinatorial characteristics of the final set diagram can be optimized, and the error floor of the newly-generated code is able to be further lowered [45].

3.4.4 Simulation Results

A rate $\frac{1}{2}$ (1024, 512) irregular code was built by using the approach introduced in this section. This irregular code has a degree distribution pair with the maximum variable node degree of 9 as given in [10]. According to the expression shown in (3.18), the number of edges in the desired code can be obtained, which is 3637. By applying the rules shown in (3.19), the 2-dimensional Euclidean Geometry over $GF(2^4)$ was chosen to
generate a raw set diagram. Based on our previous discussion, we know that a raw set
diagram can be generated by using incidence vectors for the lines that are not passing
through origin in the 2-dimensional Euclidean Geometry over $GF(2^4)$. In this raw set
diagram, there are 256 check node sets in check set diagram and 256 variable node sets in
variable set diagram. According to the refine strategy, the raw check node set diagram
was chosen to be refined first. By applying the partial splitting-and-filling technique to
the raw check node set diagram, a new set diagram which has 256 variable node sets and
512 check node sets was obtained. According to the above step 3, this 512×256 raw set
diagram was refined to a 512×1024 matrix $H_1$ by using the full splitting-and filling
refining process. And the final matrix $H_1$ is an irregular low-density matrix and has a
given degree distribution pair and girth at least six.

The binary encoded bits by using the parity-check matrix $H_1$ were sent through an
AWGN channel. At the receiver side, the belief propagation algorithm was used and a
maximum of 100 iterations were performed for each ruined codeword, i.e., belief
propagation decoding process stops when a codeword is either successfully decoded or
the iteration number reaches 100. The BER and FER results are plotted in Figure 3.9. By
Comparing with the curves in this figure with those in Figure 3.2, we can see that this
code has better near capacity performance even though its length is smaller than the code
in Figure 3.2. This improvement of the capacity performances further verifies that the
irregular codes have better capacity performance than the regular codes. Compared
Figure 3.9 with Figure 3.6, this code has much better error floor performance even
though its length is smaller than the code in Figure 3.6. The comparison between Figure
3.9 and Figure 3.6 shows that this code has good combinatorial characteristics, because an LDPC code with longer length usually has better near capacity performance and better error floor performance than an LDPC code with shorter length if these two codes are generated using a same method. Therefore, the code constructed using our method possesses both good capacity performance and error floor performance.

![Figure 3.9](image.png)

*Figure 3.9* The performance of a rate 1/2 irregular LDPC code with length 1024. The parity-check matrix is $512 \times 1024$, and the number of iterations is 100.

In order to have an idea of this code’s performance, two 1024-bit codes, one of which is a random irregular code and the other is a Richardson’s irregular code (RU
code) [10], were included as benchmarks. The performances of these 3 irregular codes are drawn in Figure 3.10. This figure shows that the code $H_1$ generated using the proposed approach performs better than the randomly generated code. This RU code is an ideal code and was analyzed using the density evolution tools by Richardson in [10].

![Figure 3.10](image)

*Figure 3.10* The performance comparison between a raw irregular LDPC code and its optimized counterpart. The codes lengths are 1024 and their rates are both $\frac{1}{2}$. The matrix is $512 \times 1024$ and the number of iterations is 100.
Figure 3.11: The performance of a rate 1/2 irregular LDPC code with length 5999. The parity-check matrix is $2999 \times 5999$ and the iteration number is 20.

Compared with the RU code and the random code, a mineral penalty is shown up when $E_b/N_0 > 1.6$ because the given degree distribution pair is for random code but the code $H_1$ has a fixed structure. When $E_b/N_0 > 1.6$ the performance of code $H_1$ is improved a lot compared with the random code, especially in the error floor area. At $E_b/N_0 = 2.4$, the curve of code $H_1$ is lower approximately an order than the one of random code. This improvement mainly benefits from the good combinatorial characteristics of code $H_1$. If the combinatorial characteristic of the proposed code can be further optimized, the error...
floor will be further lowered [45]. In Chapter 4, an approach for further optimizing the combinatorial characteristics of this code will be proposed.

In order to show the flexibility of this construction algorithm, another half rate code of length 6000 whose stopping size is larger than 6 was generated using this algorithm. And its simulation result is demonstrated in Figure 3.11. Compared this code with the random regular and irregular codes with lengths shown in Figure 3.2 and Figure 3.6 respectively, this code’s near capacity performance loses mineral. However, the error floor shown in Figure 3.11 is lowered more than 1.5 orders compared with the one in Figure 3.6.

3.4 Summary

In Sum, this new direct and simple construction approach can be used to design the irregular LDPC codes with desired degree distribution pairs. The proposed codes exhibit both good near capacity performance and good error floor performance when decoded using the iterative belief propagation decoding method.
CHAPTER 4: A NEW APPROACH FOR ENUMERATING THE ERROR PATTERNS IN LDPC CODES

The stopping sets and stopping sets in LDPC codes are all referred to as error patterns in this dissertation. With the help of the proposed definitions of stopping sets and trapping sets based on the set diagram in Chapter 2, we will present an effective algorithm for enumerating the smallest non-empty error patterns for any individual LDPC code in this chapter. The enumeration approach proceeds by building a searching diagram similar to a searching tree. Based on the process of creating a searching diagram, the computational complexity of this algorithm will be analyzed with the help of the average degrees of variable and check nodes. Finally, we will show how to optimize the error floor performance of this code based on the search results obtained from this searching algorithm.

4.1 The Background of Enumerating the Error Patterns for LDPC Codes

The nonempty stopping sets or trapping sets, especially the smallest ones, play a crucial role in the error floor region of an LDPC code when this code is transmitted over BEC or AWGN channels respectively. Therefore, enumerating this kind of error patterns for any individual LDPC code is necessary for analyzing and optimizing it. However, this kind of searching is an NP-hardness problem [57], and there are few studies giving a specific approach for searching these error patterns, either stopping sets or trapping sets, for an individual LDPC code. Recently, Wang et al. [58] investigated the stopping sets and proposed an efficient algorithm of searching the error patterns for any arbitrary
LDPC code by analyzing its upper bound. However, Wang’s algorithm can be effective only when applied to the LDPC codes with very short lengths \( n \approx 500 \), but most of LDPC codes in real applications have length larger than 500. Besides Wang’s approach, some other researchers [23] [59] [60] [61] [62] proposed methods of searching or analyzing the stopping sets or Hamming distance for an individual code using the combinatorial or algebraic method, however these approaches are either based on some assumptions, such as full rank of parity-check matrices, or have strict restrictions on the structures of LDPC codes.

In this chapter, we will propose a novel approach for enumerating all the smallest non-empty stopping sets for any individual LDPC code, either random code or structured code. More importantly, there is no any restriction on the finite block lengths of LDPC codes. When applying this approach to an LDPC code, a tree-like searching diagram needs to be built based on the set diagram, a new representation of LDPC codes presented in chapter 2. This approach can be easily extended to enumerate all the possible trapping sets of any individual LDPC code by modifying the definitions of successful paths in the search diagram. After enumerating all these error patterns, a method of eliminating these error patterns for the code generated in chapter 3 will also be given. In this dissertation, the error patterns are mainly referred to the non-empty error patterns if there are no special descriptions.
4.2 Exhaustively Searching the Error Patterns in LDPC Codes

This section presents an effective algorithm which can search error patterns for any individual LDPC code. The searching process is a process of creating a searching diagram with the help of the set diagram. With different setting on the successful paths, this algorithm can exhaustively enumerate either stopping sets or trapping sets for a given LDPC code.

4.2.1 Enumerating Stopping Sets for LDPC Codes

As described in Chapter 2 and Section 4.1, the nonempty stopping sets, especially the smallest nonempty stopping set(s), may seriously degrade the error floor performance of an LDPC code in its error floor area when this code is transmitted over a BEC channel. The negative influence of a stopping set can be explained based on its definition in the set diagram [39]. Assume that $S$ is a nonempty stopping set defined as in Definition 2.5 in Chapter 2. Also suppose that all the variable nodes contained in $S$ happen to be a subset of the erased variable nodes at receiver side. For an iterative message passing decoder, it might stop its iterative decoding procedure when all variable nodes in a received codeword are restored or the decoder fails to propagate its decoding procedure. Because every element in $S$ (it can also be said as a check node connected with $S$ or a neighbor of $S$) is covered by at least two variable node sets in $S$, the true values of the element in $S$ cannot be decided after they receive the messages transmitted from the variable nodes in $S$. Therefore, all messages transmitted back to $S$ are uncertain (or erasure) messages, which means that the decoder fails to recover all the variable nodes in $S$ and the error
floor shows up. The complete proof of the failure of iterative decoder due to stopping sets can be found in [30].

Therefore, exploring an approach of searching stopping distance and exhaustively enumerating the smallest stopping set(s) is meaningful for analyzing or optimizing a specific LDPC code in order to evaluate and improve the performance of this code [39]. Next, we will present such an algorithm which can be used to search stopping sets for any individual LDPC code.

To enumerate all the nonempty smallest stopping set(s) for any individual LDPC code, this searching algorithm needs to construct a searching diagram similar to a searching tree with the help of the definition of stopping sets defined based on the set diagram [39]. For any given variable node, this algorithm can find all possible stopping sets in which this variable node is involved by traverse a searching diagram. The searching diagram is composed of edges and states. The edges are used to connect two states at different levels. At the same time, each edge is labeled with a set of check nodes. Each state represents a variable node in an LDPC code. A path in this searching diagram is defined as a sequence of states such that from each lower level of it states there is an edge to the next higher level state in the sequence. Paths in the searching diagram can be classified into failed paths and successful paths. A successful path in the searching diagram is defined in Rule 2, which can be referred to as a stopping set because the variable nodes in this path can satisfy the definition of a stopping set. On the contrary, a failed path has nothing to do with the stopping sets but is helpful for improving the efficiency of the searching diagram. The valid states at the searching procedure $i$ are
represented by a set of variable nodes at the $i$th level, and these variable nodes are possible to be included in one of stopping sets [39]. The initial state is the variable node whose stopping sets are to be searched. For convenience, we will use the terms state, variable (node), and vertex interchangeably in this dissertation.

Similar to the trellis diagram, there are two functions used to drive the searching diagram [39]: the output function and the searching state transition function. The output function can be described by

$$ O_i = f_i(s_i, I_i) \quad (4.1) $$

and the searching state transition function can be written as

$$ s_{i+1} = g_i(s_i, I_{i+1}) \quad (4.2) $$

where $I_i = O_{i-1}$ for $i > 1$ and $I_i = \emptyset$ for $i = 1$. The output function is used to generate the check nodes on each edge, and the state transition function targets at generating the next state in the searching diagram.

In the output function shown in expression (4.1), its inputs are vertex $s_i$ and a set of check nodes, $I_i$. The $I_i$ are the previously outputs of out function shown in (4.1). The $I_i$ are composed of several general check nodes and one special check node. If a check node in $I_i$ is used to derive the next states in level $i+1$, this check node is referred to as the special check node and marked by a hat in the searching diagram. Otherwise, a check node in $I_i$ is called the general check node. The outputs of expression (4.1) are a sequence of check nodes which are the union of the inputs $I_i$ and the neighbors of the node $i$. The neighbor of $i$ are usually obtained based on the information in the variable node set diagram. When generating the outputs $O_i$, if a check nodes is covered by both the input $I_i$,
and the neighbors of state $i$, this check node in the output $O_i$ needs to be boxed in the searching diagram in order to show that it has been visited twice or more. At the same time, if a check node in the input $I_i$ has a hat, and its hat needs to be kept in $O_i$. In order to reduce the computational complexity, the repeated paths need to be ceased and marked in the searching diagram. The rules of stopping a repeated path are list as rule 1 and rule 2 [39].

**Rule 1**: If a vertex in the level $i$ is the same as one vertex in the same path in the level $k$ (where $k \leq i$), we say the path traveling from the initial vertex to this vertex is repeated or invalid, and use the failed marker $F$ as the final state in this path [39].

**Rule 2**: For a vertex at the level $i$, if all the check nodes in the output of expression (4.2) are boxed, we say this path from the initial vertex to this vertex is a *successful path*, and mark this path with $T$ as the final state. At the same time, all other paths including any one vertex in the successful path are set as invalid paths or failed paths by putting the failed marker $F$ as the final states in these paths [39].

In the searching state transition function shown in equation (4.2), the states at next level $s_{i+1}$ are generated by the neighbors of the *special check node* in $I_{i+1}$ (or $O_i$) excluding the current state $s_i$ based on the check node set graph. Figure 4.1 is an example of building the searching diagram in order to enumerate all the possible stopping sets covering the variable node $v_0$ in Figure 2.2.
Figure 4.1 The searching diagram for code in Figure 2.2.
To drive the searching diagram from level $i$ to level $i+1$, the output function (4.1) needs to be first applied for generating the sets of check nodes, $O_i$, corresponding to each edges between state $i$ and state $i + 1$. Then based on the vertex at level $i$ and $O_i$, transition function shown in (4.2) is used to derive the states in the next level $i+1$. At level 0, $I_0$ is always empty and all output $O_1$ are generated based on the initial vertex.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig4.png}
\caption{The enumerating time cost: for codes of length 1024 and 2304 both having same degree distributions. The x-coordinate is stopping distance, and the y-coordinate is logarithm of time cost used to enumerate the corresponding stopping distance.}
\end{figure}

Therefore, all $O_i$ on the edges between level $i$ to level $i + 1$ are same except that the special check nodes on different edges are different. As mentioned in rule 2, if all the
check nodes on an edge are boxed, the searching state transition function will not be applied, and we say this path from the initial vertex to the vertex connected to this edge is successful. All the vertexes on a successful path can form a stopping set.

To enumerate all the smallest stopping sets, the searching process does not need to step into the next state if a successful path in the current state is found. After applying this algorithm to every variable node in the set diagram of a given LDPC code, we can ultimately have all the different smallest stopping sets and get the stopping distance for this code.

This searching algorithm was applied to the LDPC code shown in Figure 2.2 on a 2.2-GHz duo T7500 CPU IBM computer with 2GB of RAM [39]. The searching results demonstrates that the stopping number of this LDPC code is 3 and its smallest stopping sets were enumerated as \{v_2, v_3, v_9\}, \{v_0, v_2, v_5\}, \{v_0, v_2, v_3\}, \{v_0, v_2, v_5\} and \{v_0, v_1, v_4\} in 0.328s. In order to show the flexibility of this searching algorithm, we also applied it to two other randomly generated LDPC codes with the same degree distribution pairs, and the lengths of those two codes are 1024 and 2304 respectively [39]. The searching diagrams were applied to searching the stopping sets with size 3, 4, 5, and 6 for these two LDPC codes. The enumerating time costs for these two LDPC codes at each stopping size are shown in Figure 4.2. From the searching results shown in Figure 4.2, we can see that the code lengths are roughly linear with the time costs with given degree distribution pair (or with given sparsity of the LDPC codes). That is to say, the block length of an LDPC code can affect little on the searching complexity of this searching procedure. At the same, we did not rely on any structure of these two codes. Therefore,
there is no any restriction on code block length and code structure for our searching algorithm.

4.2.2 Enumerating Trapping Sets for LDPC Codes

In Chapter 2, a trapping set $T(v, c)$ was defined as a set of $v$ variable nodes that are failed to be decoded in an iterative message passing decoder. Among all the elements of these $v$ variable node sets in the set diagram, there are $c$ elements that are covered by odd-number of these $v$ variable node sets. Currently, the studies on trapping set are mainly focused on the trapping sets $T(v, c)$ that the $c$ elements are covered by only one of $v$ variable node sets [36] [58] [63], because this kind of trapping sets plays an more important role on the error floors than other trapping set. If extending our searching algorithm for searching stopping set, we can easily enumerate this kind of trapping sets. The searching diagram used to search trapping sets $T(v, c)$, where $c$ elements are covered by only one of $v$ variable node sets, can be obtained by modifying the definition of successful path in the previous searching diagram: if only one of check nodes on an edge is not boxed, the searching state transition function will not be applied, and we say this path from the initial vertex to the vertex connected to this edge is successful. And this modification will not generate any influence on the searching procedure and searching complexity. Obviously, we can extend to a more general case by modifying the rule of boxing a check node in $O_i$: if a check nodes is covered by both the input $I_i$ and the neighbors of state $i$, this check node in the output $O_i$ needs to be boxed if check node in $I_i$ is not boxed, otherwise, unboxed it. And this extension will affect neither searching
procedure nor the searching complexity. Therefore, the searching algorithm introduced in section 4.2.1 can be used for searching either stopping sets or trapping sets. In the remained sections, we will pay attention to searching the stopping sets for LDPC codes only. All the following analysis and methods can be applied to trapping sets.

4.3 Computational Complexity of the Searching Algorithm

When enumerating the smallest stopping set(s) for an LDPC code, the proposed searching algorithm needs to build a searching diagram for every variable node of this code. Therefore, n searching diagram need to be created for an LDPC code, where n is the length of this code. That is today, there are n steps in the entire searching process [39]. In each step, the time can space complexity are totally dependent on the complexity of the searching diagram for each node. Based on the procedure of building the searching diagram, we can see that the time and space computational complexities for each searching diagram are mainly dominated by the number of valid branches in the current searching diagram and the length of the shortest valid path (or possible stopping number) that have been achieved in the searching process. Therefore, the analysis on the time and space cost should focus on the above factors.

4.3.1 Time Complexity of the Searching Algorithm

For a specific LDPC code, the number of valid branches in every searching diagram may be estimated using the average check node degree ($ACND$) and the average variable node degree ($AVND$). Therefore, the number of valid branches in each searching
The stopping distance of a code. In [38], Orlitsky et al. proved that $s_d$ approaches to $\log n$ for either a regular or an irregular LDPC code with finite length $n$. For a regular LDPC code, the $ACND$ ($AVND$) are constant and equals the degree of check (variable) nodes. While for an irregular LDPC code with a degree distribution pair

$$\gamma(x) = \sum_{i=2}^{d_v} \gamma_i \cdot x^{i-1}$$

and

$$\rho(x) = \sum_{j=2}^{d_c} \rho_i \cdot x^{j-1}$$

the $AVND$ can be obtained by using any one of the following equations

$$AVND = \left(\sum_{i=2}^{d_v} \frac{\lambda_i}{\lambda_i}\right)^{-1}$$ \hspace{1cm} (4.1)

$$AVND = \left(\frac{1}{1-r} \sum_{j=2}^{d_c} \frac{\rho_j}{\rho_j}\right)^{-1}$$ \hspace{1cm} (4.2)

where $r$ is the code rate. Similarly, the $ACND$ can be calculated by using the expression shown either in (4.3) or in (4.4).

$$ACND = \left(\sum_{j=2}^{d_c} \frac{\rho_j}{\rho_j}\right)^{-1}$$ \hspace{1cm} (4.3)

$$ACND = ((1 - r) \cdot \sum_{i=2}^{d_v} \frac{\lambda_i}{\lambda_i})^{-1}$$ \hspace{1cm} (4.4)

The Equation from (4.1) to (4.4) can also be used for regular LDPC codes even though their column weights and row weights are constants. Therefore, the time complexity of searching all the smallest stopping sets for a given either regular or irregular LDPC code can be estimated by using the Equation (4.5) [39].

$$O(\left[\left(\text{AVND} - 1\right)\left(\text{ACND} - 1\right)\right]^{s_d} \cdot n)$$ \hspace{1cm} (4.5)

And the worse case is bounded by expression (4.6) [25].
Either Equation (4.5) or Equation (4.6) demonstrates that the time cost of the proposed searching algorithm is dependent on average check node degrees and average variable node degrees of an LDPC code, both of which decide the density of its parity-check matrix. Therefore, the complexity of our searching algorithm severely relies on the sparsity of the parity-check matrix for a given LDPC code. The expressions of (4.5) and (4.6) also demonstrate that the time cost is approximately linear with the code length under certain stopping distance \( s_d \) and certain AVND (ACND), which can be verified from Figure 4.2 [39]. For most finite length LDPC codes, their ACND are around 6 and 3 respectively. So the time cost of searching stopping sets with size less than 8 are acceptable for most finite length codes.

4.3.2 Space Complexity of the Searching Algorithm

When enumerating all the smallest stopping sets for an LDPC code, we need to store all the obtained stopping sets and the branches in the current level in the searching diagram. Compared with the number of branches in the current state, the number of the smallest stopping sets can be ignored. At the same, the search diagram for each variable node is built one by one. Therefore, the space complexity of the proposed searching algorithm is mainly dependent on the valid branches of the searching diagram for each variable node. Similar to the time cost, we can use

\[
O([(AVND - 1)(ACND - 1)]^{s_d})
\]  

(4.7)
to estimate the growth of space complexity [39]. And the worse case can be bounded by expression (4.8) [39].

$$O\left( [(AVND - 1)(ACND - 1)]^{logn} \right)$$  \hspace{1cm} (4.8)

From the expressions of space complexity (either (4.7) or (4.8)), we can see that the space complexity are also dominantly dependent on both the stopping distance and the sparsity of the parity-check matrix of given LDPC code.

4.3.3 Simulation Results for the Complexity of Proposed Algorithm

When applying our searching algorithm to an LDPC code with length 2304, the time cost and the corresponding searching results are shown in Table 4.1. The LDPC code is the one generated in chapter 3 and it has an irregular degree distribution with maximum variable node degree of 6 (given in Richardson’s paper [10]). From Table 4.1, we can see that our algorithm can successfully enumerate the stopping sets with size 3, 4, 5 and 6. And it only takes $3.3 \times 10^5$ seconds to enumerate all the stopping sets with size six. If the rule 1 and rule 2 could be adopted into our searching procedure when building the searching diagram, the time complexity can be further reduced. Compared with the brutal force searching algorithm, the enumerating process is easier to be managed and the searching complexity is much more favorable for many LDPC codes, especially when the stopping distances of LDPC codes are less than eight.
Table 4.1 Stopping sets in code with length 2304. The code’s degree distribution pair is from Richardson’s paper [10] and its maximum variable node degree is 5.

<table>
<thead>
<tr>
<th>Stopping sets</th>
<th>Stopping size</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>3</td>
<td>250.905</td>
</tr>
<tr>
<td>{200, 436, 936, 1593};</td>
<td>4</td>
<td>3.25×10³</td>
</tr>
<tr>
<td>{654, 1246, 2174, 1743}</td>
<td>5</td>
<td>4.08×10⁴</td>
</tr>
<tr>
<td>None</td>
<td>6</td>
<td>3.31×10⁵</td>
</tr>
<tr>
<td>{2, 830, 967, 1352, 1983, 2146};</td>
<td>7</td>
<td>8.8×10¹⁶</td>
</tr>
<tr>
<td>{51, 1551, 1279, 1639, 2039, 2280};</td>
<td>8</td>
<td>8.8×10²⁴</td>
</tr>
<tr>
<td>{82, 538, 987, 1039, 2085, 2088};</td>
<td>9</td>
<td>8.8×10³⁶</td>
</tr>
<tr>
<td>{112, 637, 991, 2085, 2111, 2238};</td>
<td>10</td>
<td>8.8×10⁴⁸</td>
</tr>
<tr>
<td>{171, 440, 1112, 1245, 1706, 2129};</td>
<td>11</td>
<td>8.8×10⁵⁰</td>
</tr>
<tr>
<td>{181, 355, 504, 1049, 1203, 2156};</td>
<td>12</td>
<td>8.8×10⁵²</td>
</tr>
<tr>
<td>{181, 848, 1020, 1322, 1710, 1744};</td>
<td>13</td>
<td>8.8×10⁵⁴</td>
</tr>
<tr>
<td>{256, 481, 722, 1136, 2018, 2224};</td>
<td>14</td>
<td>8.8×10⁵⁶</td>
</tr>
<tr>
<td>{382, 1383, 1533, 1725, 1897, 2283};</td>
<td>15</td>
<td>8.8×10⁵⁸</td>
</tr>
<tr>
<td>{438, 576, 854, 972, 1118, 2182};</td>
<td>16</td>
<td>8.8×10⁶⁰</td>
</tr>
<tr>
<td>{485, 728, 1130, 1681, 1704, 2190};</td>
<td>17</td>
<td>8.8×10⁶²</td>
</tr>
<tr>
<td>{586, 724, 982, 1113, 1369, 1634};</td>
<td>18</td>
<td>8.8×10⁶⁴</td>
</tr>
<tr>
<td>{608, 1330, 1446, 1509, 1704, 2161}</td>
<td>19</td>
<td>8.8×10⁶⁶</td>
</tr>
</tbody>
</table>

For example, the brutal force approach needs trials of $C_{2000}^6 \approx 8.8\times10^{16}$ times to enumerate all stopping sets of size 6 of an LDPC code with length 2000, while the
proposed algorithm can take much less time to finish this task. Compared with Dr. Wang’s [58] algorithm, our searching algorithm can be used to enumerate the stopping set for LDPC codes with any finite lengths. The complexity of the proposed searching algorithm is linear with the code length which can be verified from Figure 4.2. Wang’s algorithm can be used to search stopping sets for codes with stopping distances up to thirteen. The shortcoming of ours is that it could be guaranteed to work efficient only when enumerating stopping sets for the codes with stopping distances less than eight.

4.4 Optimizing the LDPC Codes with the Help of Enumerating Results

Enumerating the error patterns for a given LDPC code not only helps to analyze the code’s performance and estimate its error floor [32] but also be useful for optimizing the error floor performance of this LDPC code [45]. In this section, we will demonstrate how to eliminate the stopping sets based on the enumerating results of the searching algorithm proposed in this chapter. This optimizing process can be applied to eliminating the trapping sets similarly. After eliminating all the stopping sets with given sizes, the simulation results of the optimized code will be given in order to show the performance improvement for the given code.

4.4.1 Optimizing the LDPC Codes

The code we first consider is the code $H$ with length 1024 we generated in Chapter 3. By building the searching diagram for each variable node of this code, the
stopping distance is achieved which is six. And all the stopping sets of size six are listed in Table 4.2.

Table 4.2 Stopping sets with size six in code H.

<table>
<thead>
<tr>
<th>Stopping sets</th>
<th>The variable nodes in corresponding stopping set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stopping Set 1</td>
<td>{v_{101}, v_{387}, v_{735}, v_{808}, v_{866}, v_{924}}</td>
</tr>
<tr>
<td>Stopping Set 2</td>
<td>{v_{103}, v_{321}, v_{341}, v_{564}, v_{818}, v_{925}}</td>
</tr>
<tr>
<td>Stopping Set 3</td>
<td>{v_{110}, v_{667}, v_{917}, v_{818}, v_{893}, v_{966}}</td>
</tr>
<tr>
<td>Stopping Set 4:</td>
<td>{v_{330}, v_{527}, v_{556}, v_{563}, v_{692}, v_{713}}</td>
</tr>
</tbody>
</table>

Table 4.3 Variable node sets in Stopping Set 1 in Table 4.1.

<table>
<thead>
<tr>
<th>Variable node</th>
<th>The elements in variable node</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v_{101})</td>
<td>{c_{122}, c_{379}}</td>
</tr>
<tr>
<td>(v_{387})</td>
<td>{c_{116}, c_{373}}</td>
</tr>
<tr>
<td>(v_{735})</td>
<td>{c_{47}, c_{373}}</td>
</tr>
<tr>
<td>(v_{808})</td>
<td>{c_{116}, c_{379}}</td>
</tr>
<tr>
<td>(v_{866})</td>
<td>{c_{122}, c_{309}}</td>
</tr>
<tr>
<td>(v_{924})</td>
<td>{c_{47}, c_{309}}</td>
</tr>
</tbody>
</table>
Let’s eliminate the Stopping Set 1 first. Based on the variable node set diagram, the elements of each variable node in Stopping Set 1 can be obtained, which are shown in Table 4.3.

Table 4.4 Variable node sets in Stopping Set 1 after optimizing.

<table>
<thead>
<tr>
<th>Variable node</th>
<th>The elements in variable node</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v_{101})</td>
<td>({c_{122}, c_{379}, c_{402}})</td>
</tr>
<tr>
<td>(v_{387})</td>
<td>({c_{116}, c_{373}})</td>
</tr>
<tr>
<td>(v_{735})</td>
<td>({c_{47}, c_{373}})</td>
</tr>
<tr>
<td>(v_{808})</td>
<td>({c_{116}, c_{379}})</td>
</tr>
<tr>
<td>(v_{866})</td>
<td>({c_{122}, c_{309}})</td>
</tr>
<tr>
<td>(v_{924})</td>
<td>({c_{47}, c_{309}})</td>
</tr>
</tbody>
</table>

From variable node sets in the Table 4.3, this stopping set can be verified because each element is covered exactly twice by two of variable nodes in the stopping set. The Stopping Set 1 can be broken if another element \(c_j\), which is originally not covered by any variable node in Stopping Set 1, is put into one of variable node sets in Stopping Set 1, because this element will be covered only once by the variable nodes in this stopping set after this moving operation. When moving an element \(c_j\), two points needs to be satisfied in order to keep the good combinatorial structure of this code: 1) all the elements in one variable node set of the newly-generated set diagram must comes from one variable node set of raw set diagram \(S_{02}^j\); 2) there is no variable node set with degree one in the newly-generated set diagram.
In order to stick to the above two points, the splitting-and-filling process used to generate this code $H$ should be tracked. Assume that an element is to be put into variable node $v_{101}$ in order to break Stopping Set 1. By recalling the generating process of code $H$ discussed in Chapter 3, we can find that the column 101 in the irregular parity-check matrix $H$ coming from the column $v'_7$ in $S'_{G_2}$. In $S'_{G_2}$, the column $v'_7$ has 6 elements. Its original refine process is shown in Figure 4.3. Therefore, any one of element in $v_{100}$ can be moved into $v_{101}$ without breaking the two points mentioned earlier. Supposed the element to be moved is $c_{402}$. In order to move the element $c_{402}$ to the variable node set $v_{101}$, the refine process shown in Figure 4.3 needs to be re-applied. The new refine
process is shown in Figure 4.4. After moving element $c_{402}$, the variable nodes and their corresponding elements in Stopping Set 1 is shown in Table 4.4.

![Diagram](image)

*Figure 4.4 Re-apply full splitting-and-filling on variable node of $v'_{78}$ in $S'_{G2}$."

Obviously, these variable nodes in Table 4.4 cannot form a stopping set any longer. That is to say, the Stopping Set 1 is successfully broken without destroy the structure of the code $H$ because the point and line concepts are still adopted in this code. Similarly, the other stopping sets shown in Table 4.2 can be broken using the above method. After re-applying the refine process related to the variable nodes $v_{101}$, $v_{103}$, $v_{110}$, and $v_{330}$, the variable node sets in the original stopping sets are shown in Table 4.4,
where the green elements are ones that have been moved. And the parity-check matrix $H$ is eventually optimized into parity-check matrix $H_2$.

In the process of breaking a stopping set, we need to make sure that this process does not generate any new stopping sets with same or less stopping size. Otherwise, we have to move back the element that generates a new stopping set and choose to move a different element.

**Table 4.5** The variable node sets in the original “stopping sets with size 6” in code $H$.

<table>
<thead>
<tr>
<th>Stopping sets</th>
<th>The variable node sets in corresponding stopping set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stopping Set 1</td>
<td>$v_{101} = {c_{122}, c_{379}, c_{402}}$; $v_{387} = {c_{116}, c_{373}}$; $v_{735} = {c_{47}, c_{373}}$; $v_{808} = {c_{116}, c_{379}}$; $v_{866} = {c_{122}, c_{309}}$; $v_{924} = {c_{47}, c_{309}}$</td>
</tr>
<tr>
<td>Stopping Set 2</td>
<td>$v_{103} = {c_{147}, c_{399}, c_{85}}$; $v_{321} = {c_{147}, c_{416}}$; $v_{341} = {c_{117}, c_{395}}$; $v_{564} = {c_{129}, c_{399}}$; $v_{818} = {c_{129}, c_{395}}$; $v_{925} = {c_{117}, c_{416}}$</td>
</tr>
<tr>
<td>Stopping Set 3</td>
<td>$v_{110} = {c_{174}, c_{395}, c_{56}}$; $v_{667} = {c_{158}, c_{367}}$; $v_{818} = {c_{129}, c_{395}}$; $v_{893} = {c_{129}, c_{367}}$; $v_{917} = {c_{174}, c_{411}}$; $v_{966} = {c_{158}, c_{411}}$</td>
</tr>
<tr>
<td>Stopping Set 4</td>
<td>$v_{330} = {c_{132}, c_{381}, c_{242}}$; $v_{527} = {c_{132}, c_{357}}$; $v_{556} = {c_{38}, c_{349}}$; $v_{563} = {c_{46}, c_{357}}$; $v_{692} = {c_{38}, c_{381}}$; $v_{713} = {c_{46}, c_{349}}$</td>
</tr>
</tbody>
</table>
Figure 4.5 The performance for an optimized code $H_2$ compared with other codes introduced in previous chapters. All the codes have lengths 1024 and code rates $\frac{1}{2}$. The iteration number is 100.

One thing needs be noted is that the optimizing process modified the variable node degree distributions unexpectedly. For example, compared with Figure 4.3, the newly-generated variable nodes in Figure 4.4 are two nodes with degree 3 instead of degree 2 and degree 4 shown in Figure 4.3. However, the number of stopping sets is not large compared with the code block length. Therefore, this modification on the degree distributions is small and has minor punishment on the code’s capacity of approaching
the Shannon limit. The simulation results in the next section also verify this minor punishment.

4.4.2 Simulation Results for the Optimized LDPC codes

The simulation results for the optimized LDPC code are shown in Figure 4.5. In order to show the improvement, we also draw the original LDPC code $H$ and the Richardson’s parity-check code [10] in this chart. From this figure, we can see that the error floor of the optimized LDPC code $H_2$ is lowered around half an order compared to the original LDPC code $H$, and it is even lower than the Richardson’s code. However, as mentioned earlier, there is a very small penalty on the capacity performance of $H_2$ at around $E_b/N_0 = 1.4dB$, because our optimizing process damaged a little bit of the wisely chosen degree distributions, which can be seen from Figure 4.4 and Table 4.5. If we enumerate all the stopping sets with size larger than 6 and eliminate all of them, the error floor can be further lowered.

4.5 Summary

In sum, the searching algorithm for enumerating the stopping sets proposed in this chapter is useful and has favorable time and space costs when applied to stopping sets with stopping distance less than eight. More importantly there is no special limitation on block lengths and code types of the LDPC codes for this algorithm. Based on the searching results, an optimization method was presented. This optimization approach can
effectively lower the error floors of LDPC codes and is attractive especially for some systems that need low error floors.
CHAPTER 5: LOW ERROR FLOOR DECODERS

When decoding a finite length LDPC code using a message passing algorithm, its error floor performance may decrease suddenly [32] as the $E_b/N_0$ increases, which is known as an error floor phenomenon. The error floor phenomenon of LDPC codes restrains their application in some systems, such as deep space communication systems and storage systems, which desire very low error rates. Lowering the error floors of LDPC codes is extremely attractive to these systems. The error floor phenomenon of LDPC codes, which is associated with their message passing decoding algorithms, is mainly caused by some unfavorable combinatorial characteristics (or error patterns) of LDPC codes. In this chapter, a mathematical analysis method which enables the identification of some significant features of those possible error patterns is presented with the help of the beliefs passed in their decoders. Based on the analysis of beliefs passed among error patterns, an improved decoder is proposed, which can be applied to any individual LDPC code over an AWGN channel. The proposed decoder can effectively deal with the traversable trapping sets and achieve significantly improved error floor performance compared with current decoders. More importantly, this proposed decoder does not need to know all the possible trapping sets of an individual LDPC code when correcting the error bits caused by its error patterns.

5.1 Overview of the Belief Propagation Decoders for LDPC Codes

Low-density parity-check codes have been extensively studied since their rediscovery [3] [4] [5] [9] [53] because of their excellent Shannon-limit performance
when decoded using low complexity iterative decoding algorithms. The *message passing algorithms* that have been introduced in Chapter 2 are a class of iterative decoding algorithms. One of the most significant advantages of message passing decoding algorithms is their low computing complexity \[9\] \[29\] \[64\]. When decoding an LDPC code, the message passing algorithm proceeds in the code’s corresponding bipartite graph. At each round of this algorithm, there are messages exchanged between variable nodes and check nodes along the edges in the code’s bipartite, i.e., at each round, each variable node \(v_i\) sends its associated message to each neighboring check node \(c_j\), then each \(c_j\) processes the received messages and sends its processed result back to its connected variable nodes.

An important restriction on the process of sending messages is that a message passed along an edge \(e\) from a node cannot depend on the message previously received along the same edge. If the messages passed in a message passing decoder are continuous reliable probabilities of variable node values, these probabilities can be called beliefs and this decoding algorithm is referred to as a belief propagation algorithm. The BP decoding algorithms are a class of optimal or suboptimal message passing decoding algorithms. The concept of belief propagation is often used in the Artificial Intelligence community, and was applied to the decoding process of LDPC codes in the 1990’s \[3\] \[5\] \[4\] \[53\] \[65\]. Gallager’s decoding algorithm \(A\) in \[2\] is actually a simple version of the belief propagation algorithm even though it is based on hard decisions. In \[9\], the belief propagation decoding algorithm was extensively analyzed over various channels with the help of density evolution tools. Another powerful analysis tool for the BP algorithm is
extrinsic information transfer (EXIT) charts \[66\] \[67\] \[68\]. By analyzing the information passed in BP decoding algorithm using EXIT, the convergence tendency of iterative decoding process can be accurately predicted.

With the assumption that the incoming beliefs are independent at every round in the decoding process (this assumption is called the independence assumption), the BP decoding algorithm is able to efficiently decode an LDPC code transmitted over a channel within its threshold \[9\]. However, this assumption cannot always be guaranteed especially when the number of iterations becomes large because of some unfavorable combinatorial characteristics in LDPC codes. These unfavorable combinatorial characteristics are originally referred to as low-weight near code words \[31\], and later called trapping sets \[32\] (or stopping sets if codes are over binary erasure channels \[9\] \[30\]). The phenomena of the high error floors of LDPC codes are mainly attributed to the existence of such combinatorial characteristics \[32\]. Because the messages passed in the belief propagation decoding algorithm are related to the channels over which codes are transmitted, those combinatorial characteristics of LDPC codes should be investigated according to the different channel type. In \[9\] and \[30\], those combinatorial characteristics that lead to high error floors of LDPC codes over erasure channels were thoroughly analyzed. But for LDPC codes over other binary symmetrical channels, there is no systematic method of enumerating all the possible error patterns (trapping sets) for LDPC codes. Currently, this kind of combinatorial characteristic is collected only by continually observing the decoding results in the simulation process \[32\] \[33\] \[34\] \[35\]. In chapter 3, a searching algorithm similar to the tree-searching was proposed to
exhaustively enumerate trapping sets and stopping sets, but this algorithm is only effective for some small size trapping sets and stopping sets. A mathematic strategy has been presented in [58] to searching trapping sets or stopping sets with size up to 13 for LDPC codes with lengths less than 500. However, most LDPC codes in real application are larger than 500. In [36] and [37], the authors adopted the Important Sampling method to calculate and estimate these combinatorial characteristics with the help of observation results. No matter what methods are used, exhaustively enumerating all such possible combinatorial characteristics is a profoundly unpleasant task and often infeasible either using the observations or any one of the above methods. Therefore, a systematic analysis of such combinatorial characteristics is needed in order to analyze the performance of any specific LDPC code and improve its error floor performance. This chapter will investigate the beliefs passed in the belief propagation decoder and present a mathematical analysis method of identifying some significant features of the possible unfavorable combinatorial characteristics of LDPC codes over binary AWGN channels.

As mentioned earlier, a final target of identifying bad combinatorial characteristics for LDPC codes is to analyze and improve their error floor performance. Lowering error floors of LDPC codes is extremely attractive for some communication systems and storage systems. Based on the identified bad combinatorial characteristics, many approaches have been explored to lower the error floors of LDPC codes. These approaches can generally be classified into three categories: designing LDPC codes with better combinatorial characteristics [23] [41] [45] [58] [69] [70] [71] [72] [73] [74] [75], modifying the scheduling of decoders to overcome the negative effect of error patterns
and adding outer codes to the LDPC codes [77]. In Chapter 3, a systematic approach of optimizing the combinatorial characteristics of LDPC codes has been proposed. In this chapter, a two-stage decoder which can effectively improve the error floor performance of LDPC codes will be presented. Like the post-processing techniques in [34], [35], and [36], the first stage of our two-stage decoder is a conventional decoding process. In the second stage, our decoder focuses on directly identifying which bits are wrong instead of using erasure decoding [35] or looking up tables [36]. Compared with those approaches in [35] and [36], the improved decoder to be presented can effectively identify the variable nodes in a traversable trapping set and mitigate their negative impact by analyzing the beliefs in the BP decoding algorithm. More importantly, we do not need to know any information about the combinatorial characteristics of an individual LDPC code to solve its error floor problem caused by trapping sets.

The rest of this chapter is organized as follows. Section 5.2 briefly introduces some terms and the BP algorithm. In section 5.3, the features of possible unfavorable combinatorial characteristics for binary erasure channels and binary AWGN channels are characterized by analyzing the belief propagation decoding algorithm for LDPC codes. Section 5.4 provides an effective approach to mitigating the impact of these characteristics. Section 5.5 gives the simulation results and the decoder evaluation. Finally, section 5.6 provides conclusions for this improved decoder.
5.2 Terminology and Background

In this section, the Belief propagation decoding algorithm is discussed in detail. Based on the decoding process and the different channels, the format of beliefs passed in the decoding procedure and some related rules for passing beliefs will be presented. These belief formats and related rules are very helpful for analyzing the error patterns and building our improved decoder for mitigating the negative influence of unfavorable combinatorial characteristics on the error floors of LDPC codes.

5.2.1 The Belief Propagation Algorithm

Currently, the belief propagation algorithm is the most widely used decoding algorithm among message passing algorithms for LDPC codes. Therefore, our study will focus on the error patterns of LDPC codes when decoded using BP decoding algorithm. Like message passing algorithms, the belief propagation decoding algorithm consists of several computing rounds and each round includes two parts. In the first half, every check node processes the messages coming from its neighbors and then sends the extrinsic processed results back along the edges in the Tanner graph. In the second half, every variable node processes the incoming messages sent in the first part and passes the extrinsic result back to its connected check nodes. At iteration zero, all check nodes send nothing to their neighbors and all variable nodes get their process results only according to the observation (inputs of the decoder).
Let \( \mathbf{y} \) be the observation and \( \mathbf{x} \) be an equally-probable transmitted codeword. Given the observed value \( y \), the bits in a codeword can be estimated using their likelihood ratios (APP ratios)

\[
l(x_i) = \frac{p(x_i=0|y_i)}{p(x_i=1|y_i)} = \frac{p(x_i=0|y_i)}{1-p(x_i=0|y_i)} \tag{5.1}
\]

or log-likelihood ratios

\[
L(x_i) = \ln[l(x_i)] = \ln[p(x_i = 0/y_i)/p(x_i = 1/y_i)] \tag{5.2}
\]

If a code’s factor graph is a tree, the belief propagation decoding algorithm can achieve the performance of maximum a posteriori (MAP) decoding when using likelihood or log-likelihood representation \[9\] \[64\]. In practice, the log-likelihood ratios are more convenient compared with likelihood ratios and our analysis will be based on the log-likelihood ratios. On the check node side, the information should be processed according to the sum-product algorithm in the factor graph [64], and each check node’s log-likelihood ratios may be obtained by the following relation

\[
L = \ln[l(x_i \oplus \cdots \oplus x_{d-1}/y_1, \cdots, y_{d-1})] \tag{5.3}
\]

In order to simplify equation (5.3) \[78\], we need to prove another equation first. From equation (5.1), we can get the probability shown in equation (5.4) based on hyperbolic function.
With the assumption that the channels are AWGN channels, the observed bit values \( y_1, y_2, \ldots, y_d \) are independent and \( x_1, x_2, \ldots, x_n \) (\( x_i \) is either 0 or 1) may be viewed as random variables. By applying Bayes’ rule to \( L(x/y) \), we have

\[
\ln(L(x/y_1, y_2, \ldots, y_d)) = \sum_{i=1}^{d} \ln(L(x/y_i))
\]

Therefore, we can induce the following conclusion using equation (5.5).

\[
2 \cdot p(x_1 \oplus \cdots \oplus x_d = 0/y_1, \ldots, y_d) = 1
= 2 \cdot [p(x_1 = 0, x_2 \oplus \cdots \oplus x_d = 0/y_1, \ldots, y_d) + p(x_1 = 1, x_2 \oplus \cdots \oplus x_d = 1/y_1, \ldots, y_d) - 1 - p(x_1 = 0/y_1)]
= 2 \cdot [p(x_1 = 0/y_1) \cdot p(x_2 \oplus \cdots \oplus x_d = 0/y_2, \ldots, y_d) + [1 - p(x_1 = 0/y_1)] \cdot [1 - p(x_2 \oplus \cdots \oplus x_d = 0/y_2, \ldots, y_d)] - 1]
= [2 \cdot p(x_1 = 0) - 1][2 \cdot p(x_2 \oplus \cdots \oplus x_d = 0) - 1]
= \prod_{i=1}^{d} [2 \cdot p(x_i = 0/y_i) - 1]
\]

With the help of equation (5.2), equation (5.4) and equation (5.6), we can get the following iteration equation for \( L \).
Based on the iteration equation in equation (5.7) and the rule of the sum-product algorithm [64], the messages transmitted between variable nodes and check nodes can be obtained as shown in Eq. (5.8-5.9) [40] [78].

\[
L = \ln \frac{p(x_1 \oplus \cdots \oplus x_{d_c-1} = 0 / y_1, \ldots, y_{d_c-1})}{1 - p(x_1 \oplus \cdots \oplus x_{d_c-1} = 0 / y_1, \ldots, y_{d_c-1})} = \ln \frac{2 \cdot p(x_1 \oplus \cdots \oplus x_{d_c-1} / y_1, \ldots, y_{d_c-1}) - 1 + 1}{1 - 2 p(x_1 \oplus \cdots \oplus x_{d_c-1}) + 1} = \ln \frac{1 + \prod_{i=1}^{d_c-1} (2p(x_i = 0 / y_i) - 1)}{1 - \prod_{i=1}^{d_c-1} 2p(x_i = 0 / y_i) - 1} = \ln \frac{1 + \prod_{i=1}^{d_c-1} \tanh[L(x_i)/2]}{1 - \prod_{i=1}^{d_c-1} \tanh[L(x_i)/2]} \tag{5.7}
\]

Eq. (5.9) can also be written in the form of its inverse function [79] which is shown in equation (5.10).

\[
m_{v}^{l} = \begin{cases} m_{v}^{l} & \text{if } l = 0; \\ m_{v}^{l} + \sum_{c' \in C_{v}(c)} m_{c'}^{l-1} & \text{if } l \geq 1; \end{cases} \tag{5.8}
\]

\[
m_{c}^{l} = \ln \frac{1 + \prod_{v \in V_{c}(v)} \tanh \left( \frac{m_{v}^{l} / 2}{2} \right)}{1 - \prod_{v \in V_{c}(v)} \tanh \left( \frac{m_{v}^{l} / 2}{2} \right)} \tag{5.9}
\]

where \(m_{v}^{l} \) is the belief passed from variable node \( v \) to check node \( c \) at the \( l \)th round of decoding process; \( m_{c}^{l} \) has a similar meaning, i.e., the belief passed from check node \( c \) to variable node \( v \) at the \( l \)th round of the algorithm; \( C_{v}(c) \) is the set of neighbors of variable node \( v \) exclude \( c \); and \( V_{c}(v) \) is the set of neighbors of check node \( c \) exclude \( v \). equation (5.9) can also be written in the form of its inverse function [79] which is shown in equation (5.10).

\[
m_{c}^{l} = 2 \tanh^{-1} \left( \prod_{v' \in V_{c}(v)} \tanh \left( \frac{m_{v'}^{l}}{2} \right) \right) \tag{5.10}
\]
In real applications, the standard pulse amplitude modulation (PAM) rule is usually used. In this situation, the AWGN channels may be defined by the transition p.d.f. \( p_{Y/X}(y/x) \) with input \{-1, 1\} and output \( \mathbb{R} \). The outputs of channels can be directly used as the inputs of decoders.

5.2.2 Stopping Sets and Trapping Sets in LDPC Codes

If an LDPC code is transmitted over a binary erasure channel, its error pattern can be defined as a set of variable nodes whose neighbors are connected to this set twice or more in the subgraph induced by this set of variable nodes. Such an error pattern is called a stopping set \([9][30]\). The definition of stopping set has been given in Definition 2.2 in Chapter 2.

Since there are three possible output values over binary erasure channels, i.e. certainty value 0, 1, and erasure value \( \varepsilon \), we generally set the messages of belief propagation algorithm at each round using this rule \([40]\): the processed result of a variable node will be a certainty value if there is at least one incoming certainty message; otherwise, the processed result of this variable node will be an erasure value; for a check node, its processed result would be a certainty value if and only if all incoming messages are certainty values; otherwise, its result would be an erasure value. Combining this rule with Definition 2.2, we can explain why the stopping sets might degrade the code’s performance in the error floor area under iterative belief propagation decoding when message bits are transmitted over a binary erasure channel. The definition of stopping
sets and their roles on error floors of LDPC codes were well explained in [9], [30] and Chapter 3.

If the LDPC codes are transmitted over other binary memoryless channels, such as binary symmetric channels and binary additive white Gaussian noise channels, the error patterns cannot be analyzed as easily as stopping sets. By simulating the LDPC codes over binary symmetric channels (BSCs) and binary additive white Gaussian noise channels, Richardson [32] observed another type of error patterns which dominate the error floor of LDPC codes and called them as trapping sets (MacKay named them as low weight near code words in [31]) However, Richardson did not give any general analysis method of identifying trapping sets. He only presented some possible combinatorial characteristics of this kind of variable node sets according to his observation. The definition of trapping sets has been given in Definition 2.3. General speaking, a trapping set $T(v, c)$ is a set of $v$ variable nodes that are not successfully decoded in the decoder, and the subgraph induced by $T(v, c)$ has $c$ check nodes with odd-degrees.

Based on the studies of [31] and [32], many papers [34] [35] [36] [63] were published on the distribution and influence of trapping sets. At the same time, some subtle strategies of mitigating the negative influence of trapping sets and lowering the error floors of LDPC codes [34] [35] [36] [76] were stated. However, these studies are mainly based on the observed trapping sets and only work for some specific LDPC codes. There is no analysis on identifying trapping sets for any individual LDPC code. In section 5.3, we will analyze the combinatorial properties of these error patterns for LDPC codes over binary symmetric channels and binary AWGN channels. Based on the beliefs passed
in the belief propagation decoding process, some features of error patterns in LDPC codes will be discussed.

5.3 Error Patterns in LDPC Codes

In this section, the combinatorial characteristics of error patterns are investigated with the help of beliefs passed between variable nodes and check nodes. For LDPC codes over BEC channels, the conclusion on the combinatorial properties of stopping sets made in [32] and [30] is verified. While for LDPC codes over AWGN channels, some features of error patterns (named trapping sets in this case) can be obtained by analyzing the beliefs passed in codes’ bipartite graphs.

5.3.1 Error Patterns for LDPC Codes over BEC

In order to delve into the error patterns of LDPC codes over BEC channels, the rule mentioned in section 5.2 should be considered, i.e., the output messages of a variable node are erasures if all its input messages are erasures, and the output messages of a check node are erasures if it has one or more erasure inputs. According to equation (5.9), any output of a check node with degree $l$ is dependent on the product of $(l-1)$ hyperbolic tangents of its inputs. Therefore, any output of a check node will be an erasure if its two or more inputs are erasures. Based on this observation, a necessary condition of stopping a decoding process can be obtained: for a set of variable nodes, each neighbor of this set (or subgraph) has two or more erasure inputs at each round of decoding. From equation (5.9), we can see that the decoding process will fail if each neighbor of a set of variable
nodes whose values are erasures is connected to this set twice or more. Therefore, for an LDPC codes transmitted over BECs, a set of variable nodes with the above combinatorial property can be identified as an error pattern of this code, or called a stopping set [9] [30]. This conclusion on stopping sets is the same as the one given in [30].

5.3.2 Error Patterns for LDPC Codes over BAWGN Channels

The analysis of belief propagation decoding over AWGN channels is more complicated and general compared with BEC channels. In [32], Richardson defined the decoding failure of an LDPC code over AWGN channels based on a set of incorrect variable nodes which is called a trapping set or failure set $T(y)$, i.e., after a certain large number of iterations, the decoding fails if $T(y)$ is not empty. In this sub-section, some features of $T(y)$ will be analyzed by observing the reliability of beliefs passed in decoding process. For convenience, we will use $T(y)$ and the subgraph induced by $T(y)$ interchangeably, and message bits and variable nodes interchangeably in this paper.

(a) OCN connected to $T(y)$ once.
(b) OCN is connected to $T(y) \times 2^j$ times.

(c) OCN is connected to $T(y) \times (2^j + 1)$ times.

*Figure 5.1* Combinatorial characteristics of out-check-nodes. $\varnothing =$ variable nodes in this trapping set; $\blacksquare =$ even-degree check nodes in the subgraph; $\blacklozenge =$ odd-degree check nodes in the subgraph; $\Diamond =$ variable nodes outside of the subgraph; $j$ is an integer.
For convenience, the neighbors of $T(y)$ are classified into two types. One type is the check nodes that are only connected to variable nodes in $T(y)$, and named as in-check-nodes (ICNs). The other is named out-check-nodes (OCNs), which are connected not only to variable nodes in $T(y)$ but also to variable nodes outside of $T(y)$. For a trapping set, most check nodes connected to it are OCNs. For either OCNs or ICNs, if the corresponding check equations are satisfied, they cannot provide any meaningful beliefs to their neighbors and are not able to help correct the message bits in $T(y)$. This type of check nodes are referred to as mis-satisfied check nodes in [35] [36]. In this chapter, only OCNs will be considered when analyzing the beliefs related to $T(y)$ because the ICNs have similar influence on $T(y)$ to OCNs. In Fig. 5.1, three possible connections are listed for the out-check nodes. According to the connections ways as shown in Figure 5.1, the ability of OCNs to correct the message bits in $T(y)$ will be analyzed. Before investigating the beliefs related to the OCNs, a useful theorem need to proved first because the analysis on the beliefs will be based on this theorem.

5.1 Theorem (Consistent Belief Theorem): For any message node in a successfully decoded codeword, each incoming belief from its neighbors always has the same sign as that of its correct value. We say this phenomenon as the belief is consistent with the bit value. Otherwise, the belief is called inconsistent (or conflicting) with the bit value.

Proof: from the incoming belief format of a variable node as shown in Eq. (9), the sign of $m_{iv}$ is totally dependent on the sign of $\prod_{v' \in V_c/(v)} tanh \left( \frac{m_{iv'}}{2} \right)$. Under PAM modulation and the parity check equations, the restraint of sum of 1 equal to 0 is
transferred to the restraint of the product of -1 equal to 1. Because the hyperbolic function is an odd function within the range (-1, 1), the product of all $m_{v'c}$, where $v' \in V_c/\{v\}$, must have the same sign as that of $\prod_{v' \in V_c/\{v\}} \tanh \left( \frac{m_{v'c}}{2} \right)$. Therefore, $m_{cv}$ and $m_{vc}$ have to have the same sign in order to satisfy the product restraint.

Remark: The target of the belief propagation decoding algorithm in LDPC codes is to correct the error bits and increase the reliability of the correct bits. This target is consistent with the above Theorem 5.1.

From the Theorem 5.1, we can see that in Figure 5.1 (a), the belief $f_{in}$ should be inconsistent with $v_{T1}$, because $f_{in}$ should have the same sign as that of the correct value of $v_{T1}$. However, the value of $v_{T1}$ is incorrect with the assumption that the message bits in $T(y)$ are trapped. Similarly, the beliefs $f_{outi}$ ($0 < i \leq k$) are conflict with the current values of $v_i$ because there is always a wrong value, $v_{T1}$, in $\prod_{v' \in V_c/\{v\}} \tanh \left( \frac{m_{v'c}}{2} \right)$. Under such connection in Fig. 1(a), a loose condition that $T(y)$ occurs is $abs(f_{in}) < abs(v_{T1})$ and $abs(f_{outi}) < abs(v_i)$, i.e., the belief $f_{in}$ cannot correct $v_{T1}$ and all outgoing beliefs $f_{outi}$ cannot change the sign of variable nodes $v_i$ where $0 < i \leq k$. Therefore, for a trapping set $T(y)$, it is possible that all or part of its OCNs have degree 1 in its corresponding subgraph. Actually, it is very common among trapping sets that all check node of a trapping set have either degree one or degree two. And this kind of trapping sets is referred to as elementary trapping sets [35] [63] or 1-out trapping sets [58].

In Figure 5.1(b), there are even number beliefs coming from the out-check-node $c_{01}$ to $T(y)$. Based on the analysis on Figure 5.1(a) and the outgoing belief format of
OCNs, $\prod_{v' \in \mathcal{V}_c \setminus \{v\}} \tanh \left( \frac{m_{v'v}}{2} \right)$, it can be easily determined that all beliefs $f_{ink} (k=2j, j$ is an integer) have the same signs as those of $v_{rk}$. Therefore, the OCN $c_{01}$ does not have any ability to correct its neighbors in $T(y)$. In contrast, it will make its neighbors to approach incorrect values further. At the same time, these wrong values are unable to be detected by this OCN $c_0$. Actually, for any neighbor (which includes OCNs and ICNs) of $T(y)$, if it has even-degree in the subgraph induced by $T(y)$, the check equation corresponding to this neighbor will be mis-satisfied even though the variable nodes in $T(y)$ are incorrect. If all check nodes in the subgraph induced by $T(y)$ have even-degrees, this “trapping set” (it should not be called trapping sets any more if under BP decoding because we cannot detected the any bits being trapped.) can form a nonzero codeword whose all 1s are the bits in this trapping set. Therefore, this phenomenon is not a near codeword (or a trapping set) problem any longer. It is a codeword’s Hamming distance problem where a codeword is wrongly decoded into another codeword, and the Hamming distance of these two codewords is the size of this “trapping set”. Therefore, we will not consider the neighbors of $T(y)$ whose degrees are even numbers when dealing with trapping sets.

Based on the analysis of Figure 5.1(a) and Figure 5.1(b), Figure 5.1(c) can be easily analyzed. As mentioned previously, the OCNs with even-degrees are not able to correct the variable nodes in $T(y)$ and all the beliefs coming from those OCNs are always consistent with the values of beliefs’ destinations. In Figure 5.1(b), if an extra edge is added between an OCN and one of variable nodes in $T(y)$, Figure 1(c) can be formed. Under this situation, the check equation corresponding to this OCN will no longer be
satisfied because there is an extra incorrect value in this check equation. Therefore, the signs of all the outgoing beliefs would be flipped and this OCN sends inconsistent beliefs to its neighbors. At the same time, the influence of the outgoing beliefs $m_{cv}^l$ would become less with larger odd-degree of this check node because the range of hyperbolic function is between -1 and 1, which can be concluded from equation (5.9). Therefore, the OCNs in Figure 5.1(c) would have similar influence on its neighbors as the OCN in Figure 5.1.

Figure 5.2. Trapping Set (12, 4) by SPA decoders.  = a path completely composed of the variable nodes in this trapping set;  = a path composed of the variable nodes outside of this trapping set; other symbols are the same as that in Figure 5.1.
The ICNs of a trapping set have the same influence as the OCNs, either with odd-degrees or with even-degrees. Therefore, if a set of variable nodes form a detectable trapping set, there must be some beliefs inconsistent with the values of their corresponding destinations (the connected variable nodes) and these beliefs are from at least one check node with odd-degree in the subgraph induced by this trapping set. This conclusion is the same as in [19] which was proposed based on observation results. In [32], a trapping set $T(v, c)$ which induces a subgraph with $c$ odd-degree check nodes is defined as a set of $v$ variable nodes. In this dissertation, we still use this definition, but we will pay more attention to the inconsistent beliefs rather than the odd-degree check in a subgraph because these beliefs can easily be monitored in the decoding process and can be used to mitigate the negative influence of trapping sets on the error floor performance of a BP decoder.

Another issue related to the trapping set under the BP decoding algorithm is the quantization level of a decoder. If a decoder has a lower level of quantization, the values of variable nodes are more easily to be forced to zeros, which can lead to forming a steady trapping set more easily. In [80], the author provided detailed information about the quantization and accuracy of BP decoders. However, our improved decoder mentioned in section 5.4 would not be affected by the approximation of a decoder because the final decisions on the value of a variable node will be randomly chosen if its value is zero.
5.4 Improved Decoders

There are few studies on decoder improvements for solving the error floor problem caused by trapping sets. In this section, we will briefly review the current improved belief propagation decoders. After that, our two-stage decoder will be presented.

5.4.1 Reviews on Current Belief Propagation Decoders

In [36], a post-processing approach was proposed by modifying the decoding procedure: the decoder first runs as a conventional BP decoder for certain number of iterations or until successfully terminated; if the decoder gets a non-zero syndrome, a look-up table of known trapping sets will be used to locate the error bits. This post-processing technique can effectively decrease the negative influence of trapping set under an assumption that a set of unsatisfied check nodes has a one to one correspondence to a trapping set. However, the application of this approach will be restricted if the cardinality of trapping sets of an LDPC code is large.

In [34] and [35], the authors proposed a bit-pinning technique which erases the bits of each trapping set of interest before transmission. This bit-pinning approach only works when there is little trapping set overlap after bit-pinning. In [35], the authors also put forward two other approaches for improving a decoder. One is called a generalized-LDPC decoder which adds check node processors to deal with the unsatisfied checks in the trapping sets. The other is named bi-mode syndrome-erasure technique which mitigates the negative influence of trapping sets on the error floor performance of an
LDPC code by combining the conventional SPA decoding and BP decoding over BECs. This method is easy to apply only when the erasure bits do not form any stopping set.

There are two major drawbacks for these improved belief propagation decoders. One is that all these decoders need to know all the possible trapping sets in order to reduce or eliminate the negative influence of the trapping sets on error floors. The other is that these decoders are restrained by the format and/or cardinality of the trapping sets in an LDPC code. Our two-stage decoder to be presented is free of these two restraints.

5.4.2 The Principle of the Two-Stage Decoder

The idea of building this two-stage decoder is based on this observation: for a cycle-free LDPC code with infinite length, the BP decoding algorithm may successfully decode when transmitted at a rate below its capacity, i.e., a wrong message bit can be corrected with any arbitrarily large possibility by the correct beliefs coming from its neighbors when running a BP decoding algorithm; However, if this wrong message bit is included in a short cycle, it may not be able to receive correct information any longer after certain number iterations, because its information travels back to itself; this is the reason for requiring the independence assumption for MPA decoding; in real applications, even though there is no cycle, a wrong message bit \( v_k \) may still be unable to be corrected because of the existence of some other wrong bits which could send wrong information to this bit after a certain number iterations, which resembles a state of having a cycle. In this dissertation, we say this wrong bit \( v_k \) is involved into a pseudo-cycle because it received wrong information from its neighbor(s) after several iterations. A
trapping set is composed of one or more cycles, either pseudo-cycles or real cycles. Based on our observation, which is also shown in Appendix 1, the shortest cycles are usually pseudo-cycles. In a trapping set, these shortest pseudo-cycles can be identified by the shortest paths between two unsatisfied check nodes because these check nodes are able to send wrong information to their neighbors. If a stopping set is traversable, most, if not all, message bits in this trapping set can be located by those shortest paths. The task of a two-stage decoder is to identify most (or all) message bits in a trapping set and further break this trapping set.

In the first stage of a two-stage decoder, a conventional BP decoding algorithm is run like the post-processing decoder. If an error occurs, the decoder goes to the second stage. At the same time, the occurrence of errors can be identified by the inconsistent belief(s) passed in the BP decoder or by the non-zero syndrome of a codeword. In the second stage, the decoder enumerates all the possible shortest paths between any two unsatisfied check nodes. After that, the decoder collects all the variable nodes in these paths and flips the values of these bits. According to our observations on our simulation results (as shown in Appendix 1), over 80% paths identified in the error floor region are purely composed of trapped bits. Therefore, the modified codeword after flipping should be a correct codeword or have very few error bits. After re-run the conventional decoding process using the modified codeword as the decoder’s inputs, over 90% the wrong codeword in the first stage of decoder can be correctly decoded in the seconded stage with high probability. The exceptional merit of this two stage decoder is that it does not need to know the format and cardinality of trapping sets when solving the error floor
problem caused by small trapping sets. Therefore, this approach can be applied to any specific LDPC code and eliminates the worries that some trapping sets are not exhaustively enumerated for some codes, which may lead to the failure of the post-processing techniques mentioned earlier. This approach can effectively correct a class of commonly occurring trapping sets, i.e., traversable trapping sets \( T(v, c) \) with parameter \( c \) no less than 2, by monitoring the beliefs passed from check nodes to variable nodes during the decoding process.

5.4.3 Building the Two-Stage Improved Decoder

Let’s assume \( H \) is an arbitrary LDPC code and its decoder converges to a traversable trapping set \( T(v, c) \) with \( c \geq 2 \) after certain number of iterations. Based on the analysis in section 5.3, all beliefs outgoing from any odd-degree check node should be inconsistent with the values of their destinations (variable nodes). If a trapping set is traversable, some paths that pass any two odd-degree check nodes should be totally within the subgraph induced by this trapping set. Those paths can be obtained by building a group of \( d_{cl} \) trees whose roots are the neighbors of an odd-degree check node \( c_i \) with degree \( d_{ci} \). In these \( d_{cl} \) trees, there should be at least one tree whose child is either an OCN or a variable node having at least one incoming belief inconsistent with its value. The shortest paths between this kind of children and the root are possibly purely composed of the variable nodes in a trapping set. If all the shortest paths are found for each odd-degree check node, some or all of the bits in a trapping set might be obtained.
Searching Path Algorithm 1:

Generate $d_{c_1}$ trees with depth 1 whose roots are the neighbors of $c_1$, and these trees are group as $a$;

Generate $d_{c_2}$ trees with depth 1 whose roots are the neighbors of $c_2$, and these trees are group as $b$;

for ($i = 1; i < m+1; i+=2)$

    extend trees in group $a$ to $i+2$ level;

    if $v_{ai} \cap v_{bi} \neq \text{null}$

        create a path between two trees having common variable node.

        Flip the values of variable node in this path

        If weight difference between new syndrome and old syndrome is equal to flipped variable nodes

            break;

        end

    elseif $v_{ai(i+2)} \cap v_{bi} \neq \text{null}$

        create a path between two trees having common variable node.

        Flip the values of variable node in this path

        If weight difference between new syndrome and old syndrome is equal to flipped variable nodes

            break;

        end

    else

        extend trees in group $b$ to $i+2$ level;

    end

end
However, this method needs to build very large trees if the size of a trapping set is large (the depth of trees is around the double size of this trapping set). We can simplify the searching process by parallel building \( c \) trees, where \( c \) is the number of odd-degree check nodes detected in the decoder. For any LDPC code \( H \), suppose the size of a trapping set of \( H \) is \( m \), and \( c_a \) and \( c_b \) are any two of odd-degree check nodes in the subgraph induced by this trapping set. For two trees, \( a \) and \( b \), whose roots are \( c_a \) and \( c_b \) correspondingly, the variable nodes in level \( i \) in these two trees \( a \) and \( b \) are marked as \( v_{ai} \) and \( v_{bi} \) correspondingly. By continuously comparing \( v_{ai} \) and \( v_{bi} \), the traversable paths between these two unsatisfied check nodes \( c_a \) and \( c_b \) can be found. For these two odd-degree check nodes, \( c_a \) and \( c_b \), whose degrees are \( d_{ca} \) and \( d_{cb} \) correspondingly in \( H \), a detailed and more efficient algorithm of finding such a path between any two check nodes is given in the above Searching Path Algorithm 1.

In the second stage of our two-stage decoder, the odd-degree check nodes need to be identified. One easy method of detecting the odd-degree check nodes is to set a marker for each check node in the conventional BP decoding process. The basic idea of setting the marker is as follows: first, set all markers of check nodes to zero; during the decoding process, if an incoming belief is inconsistent with the value of a variable node, the check node which sends out this belief must be an unsatisfied check node and its marker is set to 1. Based on the markers of check nodes, we can apply the previously mentioned algorithm of searching paths to any two odd-degree check nodes to identify the possible variable nodes in a trapping set. For paths between two unsatisfied check nodes in the subgraph, some of them may not be purely composed of the nodes in a trapping set, even
though this kind of paths is rare according to our observation. One such example is shown in Figure 5.2. Therefore, if all bits in those paths are flipped, the number of error bits may be greatly decreased (possibly to zero). If the updated codeword is used as inputs to the decoder once more, the codeword can be correctly decoded with high probability. Another approach of identifying the unsatisfied check nodes is to checking a codeword’s syndrome because all unsatisfied check nodes exactly correspond to the nonzero elements in a syndrome.

5.5 Simulation Results and Discussion

Based on the discussion in section 5.4, we applied this two-stage decoder to a rate-1/2 (1024, 512) irregular code generated in [45]. When $E_b/N_0 > 1$, the second stage runs if an error block with error bits is less than 45 (because we want to focus only on small trapping sets which dominate the error floor performance of LDPC codes). The statistical results of the error events are shown in Fig. 3 and the performance of the two-stage decoder is shown in Figure 5.4. From Figure 5.3, we can see the number of collected error events (blocks) increases as the $E_b/N_0$ increases, which means small trapping sets play a more important role in higher $E_b/N_0$ area. From Figure 5.3, we can also find that the percentage of codewords corrected in stage two increases along with the increase of the $E_b/N_0$. The detailed information about the corrected errors is given in the Appendix 1.
Figure 5.3 indicates that after flipping the bits in the searched paths, more than 32% of error events (blocks) are completely corrected when $E_b/N_0 \geq 2.2$. It is also shown that the error bits in another 40% of error events (blocks) are greatly decreased after flipping (error bits are decreased more than 80% in each block). Regarding the remaining 23% error events (blocks) that cannot be corrected, a possible explanation is that they are not steady trapping sets (oscillating trapping sets). At $E_b/N_0 = 2.6$ and $E_b/N_0 = 2.8$, around 80% error events are corrected after running the second stage of the improved decoder, because the number of error bits in these error blocks is dramatically reduced and the trapping sets are broken after flipping. That is to say, the
second stage of this decoder can effectively mitigate the negative influence of trapping sets on error floors.

Figure 5.4. The performance of the improved decoder. All codes parity-check matrix dimension are $512 \times 1024$, and decoding iteration numbers are 100.

The Figure 5.4 shows that performance of the improved decoder is much better than the conventional one. At the error floor region ($E_b/N_0 > 2.4$) where small trapping sets play a more important role, both BER and FER curves are lowered at least an order of magnitude. Because we did not use any structure characteristics of this LDPC code when dealing with its error floor, we can declare this code is chosen randomly. Therefore,
this improved decoder can effectively improve the error floor performance for any individual LDPC code.

5.6 Summary

In this chapter, some features of error patterns for LDPC codes under the BP decoding were analyzed based on the beliefs passed in the decoder. With the help of these features, a two-stage decoder is proposed which can effectively mitigate the negative influence of trapping sets on the error floors of LDPC codes. Another merit of this decoder is that it does not need to know all the possible trapping sets of an individual LDPC code when correcting the error bits in a trapping set.
6.1 Conclusions

Constructing an LDPC code with both near Shannon limit performance and low error floor performance is the main subject of this dissertation. Irregular LDPC codes with wisely chosen degree distribution pairs have better Shannon limit performance [10] but worse error floor performance [11] than their regular counterparts. Therefore, the objects of this research focus on lowering the error floors of LDPC codes, including generating good irregular LDPC codes with good combinatorial characteristics and mitigating the impact of unfavorable combinatorial characteristics.

Low error floor performance is desirable for many information transmission systems, especially the deep space systems and storage systems. By delving into the belief propagation decoding algorithm, this dissertation studies the possible causes of low error floors for LDPC codes and shows that some unfavorable combinatorial characteristics of LDPC code might severely degrade the error floor performance of LDPC codes. These unfavorable combinatorial characteristics can be named as cycles, stopping sets, trapping sets, and pseudo-cycles under different situations. At the same time, this dissertation explores the relationships among those characteristics and their impact on the codes’ error floors. Based on these analyses, several effective approaches of mitigating the influence of those unfavorable combinatorial characteristics are explored, from both the encoder side and the decoder side.
This dissertation first develops a systematic method of constructing irregular LDPC codes with good combinatorial characteristics based on Finite Geometry, with the help of the sub-optimized degree distribution pairs proposed by Richardson [10]. The constructed LDPC codes can achieve comparable Shannon limit performance with those proposed in [10]. Because the cycles with length 4 are free in our constructed codes, their error floors are lower than the randomly generated ones.

In order to conveniently apply some operations on LDPC codes, splitting the parity-check matrix for instance, a new representation method for LDPC codes called set diagram is proposed in this dissertation. Based on this representation, an efficient approach of enumerating small unfavorable combinatorial characteristics of LDPC codes, including stopping sets and trapping sets, is proposed. After combinatorial characteristics are enumerated, our constructed LDPC codes are further optimized by eliminating these small characteristics. The simulation results show that error floors of the optimized code are lowered more than half of magnitude than the non-optimized ones.

The error floors performance of LDPC codes are also improved by adopting a two-stage decoder, which was introduced in Chapter 5. This two stage decoder can effectively mitigate the influence of the unfavorable combinatorial characteristics when an error event occurs.

By combining all of the above approaches, the LDPC codes finally achieve both better Shannon limit performance and low error floor performance. An example shown in this dissertation indicates that the error floors of an LDPC code with length 1024 are
lower more than 1.5 order of magnitude, and it also has as good Shannon limit performance as the one provided by Richardson [10].

6.2 Future Studies

6.2.1 Enumeration of Trapping Sets (or Stopping Sets)

The approach of enumerating the combinatorial characteristics of LDPC codes proposed in this dissertation can be effective only for small characteristics. When the sizes of characteristics $g$ are large, the time and space complexity would become unacceptable. For some systems, very low error floors might be required. Therefore, exploring another efficient method of enumerating larger combinatorial characteristics will be helpful for these kinds of systems.

6.2.2 Identifying the Activity of Oscillating Trapping Sets

The two-stage decoder proposed in this dissertation cannot correct all events happen in the first stage of decoder. One of the probable reasons is the existence of oscillating trapping sets. By studying the activity of oscillating trapping sets in the decoding process, the form of this kind of trapping set might be identified, which would be helpful for mitigating the impact of this class of trapping set.
REFERENCES


APPENDIX 1: STATISTICAL RESULTS FOR THE IMPROVED DECODER

Note:

1. Each row is an error event collected in the first stage of the improved decoder;

2. the first column is the SNRs where the statistic results are obtained; the second column is the number of error bits in each error event collected in the first stage of the improved decoder; the third column is the number of unsatisfied check nodes of each error event collected in the first stage of the improved decoder; the third column is the number of error bits in each error event after flipping the possible variable nodes in the stopping sets in the corresponding event; the fourth column is the number of unsatisfied check nodes in the corresponding error event after running this improved decoder; the last column is the number of error bits in the corresponding event after running the improved decoder.

<table>
<thead>
<tr>
<th>SNR</th>
<th>error bits in the 1st stage</th>
<th>unsatisfied CN in the 1st stage</th>
<th>errors after Flipping</th>
<th>unsatisfied CN in the 2nd stage</th>
<th>error bits in the 2nd stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 9</td>
<td>num_unsatisfiedCNs=11</td>
<td>errorAfterFlp = 3</td>
<td>num_unsatisfiedCNs2=0</td>
<td>errorNum2 = 0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 13</td>
<td>num_unsatisfiedCNs=17</td>
<td>errorAfterFlp = 4</td>
<td>num_unsatisfiedCNs2=492</td>
<td>errorNum2 = 371</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 20</td>
<td>num_unsatisfiedCNs=22</td>
<td>errorAfterFlp = 2</td>
<td>num_unsatisfiedCNs2=4</td>
<td>errorNum2 = 2</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 12</td>
<td>num_unsatisfiedCNs=12</td>
<td>errorAfterFlp = 0</td>
<td>num_unsatisfiedCNs2=0</td>
<td>errorNum2 = 0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 8</td>
<td>num_unsatisfiedCNs=8</td>
<td>errorAfterFlp = 1</td>
<td>num_unsatisfiedCNs2=0</td>
<td>errorNum2 = 0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 10</td>
<td>num_unsatisfiedCNs=12</td>
<td>errorAfterFlp = 2</td>
<td>num_unsatisfiedCNs2=7</td>
<td>errorNum2 = 2</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 11</td>
<td>num_unsatisfiedCNs=12</td>
<td>errorAfterFlp = 1</td>
<td>num_unsatisfiedCNs2=4</td>
<td>errorNum2 = 1</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 7</td>
<td>num_unsatisfiedCNs=8</td>
<td>errorAfterFlp = 2</td>
<td>num_unsatisfiedCNs2=0</td>
<td>errorNum2 = 0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 20</td>
<td>num_unsatisfiedCNs=22</td>
<td>errorAfterFlp = 3</td>
<td>num_unsatisfiedCNs2=183</td>
<td>errorNum2 = 101</td>
</tr>
<tr>
<td>errorNum1</td>
<td>num_unsatisfiedCNs</td>
<td>errorAfterFlp</td>
<td>num_unsatisfiedCNs2</td>
<td>errorNum2</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>-------------------</td>
<td>-----------</td>
<td>-------------------</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>5</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>9</td>
<td>13</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>10</td>
<td>12</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>9</td>
<td>13</td>
<td>3</td>
<td>477</td>
<td></td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>24</td>
<td>32</td>
<td>10</td>
<td>481</td>
<td></td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>17</td>
<td>18</td>
<td>3</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.2E+000dB</td>
<td>14</td>
<td>15</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2.2E+000dB</td>
<td>11</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.2E+000dB</td>
<td>7</td>
<td>13</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.2E+000dB</td>
<td>33</td>
<td>55</td>
<td>26</td>
<td>485</td>
<td></td>
</tr>
<tr>
<td>2.2E+000dB</td>
<td>20</td>
<td>22</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

The table lists values for errorNum1, num_unsatisfiedCNs, errorAfterFlp, num_unsatisfiedCNs2, and errorNum2. Each row represents a different set of values for these parameters.
<table>
<thead>
<tr>
<th>errorNum1</th>
<th>num_unsatisfiedCNs</th>
<th>errorAfterFlp</th>
<th>num_unsatisfiedCNs2</th>
<th>errorNum2</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>2</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>18</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>22</td>
<td>3</td>
<td>109</td>
<td>60</td>
</tr>
<tr>
<td>11</td>
<td>20</td>
<td>7</td>
<td>482</td>
<td>367</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>24</td>
<td>3</td>
<td>483</td>
<td>366</td>
</tr>
<tr>
<td>35</td>
<td>66</td>
<td>39</td>
<td>378</td>
<td>207</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>50</td>
<td>28</td>
<td>407</td>
<td>270</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.2E+000dB</td>
<td>2.2E+000dB</td>
<td>2.2E+000dB</td>
<td>2.2E+000dB</td>
<td>2.2E+000dB</td>
</tr>
<tr>
<td>errorNum1</td>
<td>num_unsatisfiedCNs</td>
<td>errorAfterFlp</td>
<td>num_unsatisfiedCNs2</td>
<td>errorNum2</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------</td>
<td>---------------</td>
<td>---------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>16</td>
<td>3</td>
<td>487</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>15</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>27</td>
<td>8</td>
<td>479</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>14</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>15</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>33</td>
<td>10</td>
<td>486</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>51</td>
<td>21</td>
<td>321</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>43</td>
<td>19</td>
<td>419</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>3</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2.2E+000dB</td>
<td>errorNum1 = 9</td>
<td>num_unsatisfiedCNS = 9</td>
<td>errorAfterFlp = 0</td>
<td>num_unsatisfiedCNS2 = 0</td>
</tr>
<tr>
<td>2.2E+000dB</td>
<td>errorNum1 = 12</td>
<td>num_unsatisfiedCNS = 12</td>
<td>errorAfterFlp = 0</td>
<td>num_unsatisfiedCNS2 = 0</td>
</tr>
<tr>
<td>2.2E+000dB</td>
<td>errorNum1 = 8</td>
<td>num_unsatisfiedCNS = 8</td>
<td>errorAfterFlp = 1</td>
<td>num_unsatisfiedCNS2 = 0</td>
</tr>
<tr>
<td>2.2E+000dB</td>
<td>errorNum1 = 16</td>
<td>num_unsatisfiedCNS = 26</td>
<td>errorAfterFlp = 8</td>
<td>num_unsatisfiedCNS2 = 483</td>
</tr>
<tr>
<td>2.2E+000dB</td>
<td>errorNum1 = 18</td>
<td>num_unsatisfiedCNS = 29</td>
<td>errorAfterFlp = 9</td>
<td>num_unsatisfiedCNS2 = 478</td>
</tr>
<tr>
<td>2.2E+000dB</td>
<td>errorNum1 = 13</td>
<td>num_unsatisfiedCNS = 23</td>
<td>errorAfterFlp = 7</td>
<td>num_unsatisfiedCNS2 = 477</td>
</tr>
<tr>
<td>2.2E+000dB</td>
<td>errorNum1 = 8</td>
<td>num_unsatisfiedCNS = 8</td>
<td>errorAfterFlp = 1</td>
<td>num_unsatisfiedCNS2 = 4</td>
</tr>
<tr>
<td>2.2E+000dB</td>
<td>errorNum1 = 22</td>
<td>num_unsatisfiedCNS = 34</td>
<td>errorAfterFlp = 18</td>
<td>num_unsatisfiedCNS2 = 486</td>
</tr>
<tr>
<td>2.2E+000dB</td>
<td>errorNum1 = 16</td>
<td>num_unsatisfiedCNS = 21</td>
<td>errorAfterFlp = 4</td>
<td>num_unsatisfiedCNS2 = 0</td>
</tr>
<tr>
<td>2.2E+000dB</td>
<td>errorNum1 = 14</td>
<td>num_unsatisfiedCNS = 14</td>
<td>errorAfterFlp = 0</td>
<td>num_unsatisfiedCNS2 = 0</td>
</tr>
<tr>
<td>2.2E+000dB</td>
<td>errorNum1 = 9</td>
<td>num_unsatisfiedCNS = 9</td>
<td>errorAfterFlp = 2</td>
<td>num_unsatisfiedCNS2 = 0</td>
</tr>
<tr>
<td>2.2E+000dB</td>
<td>errorNum1 = 13</td>
<td>num_unsatisfiedCNS = 21</td>
<td>errorAfterFlp = 6</td>
<td>num_unsatisfiedCNS2 = 0</td>
</tr>
<tr>
<td>2.2E+000dB</td>
<td>errorNum1 = 17</td>
<td>num_unsatisfiedCNS = 22</td>
<td>errorAfterFlp = 3</td>
<td>num_unsatisfiedCNS2 = 0</td>
</tr>
<tr>
<td>2.2E+000dB</td>
<td>errorNum1 = 10</td>
<td>num_unsatisfiedCNS = 19</td>
<td>errorAfterFlp = 10</td>
<td>num_unsatisfiedCNS2 = 478</td>
</tr>
<tr>
<td>2.2E+000dB</td>
<td>errorNum1 = 8</td>
<td>num_unsatisfiedCNS = 8</td>
<td>errorAfterFlp = 0</td>
<td>num_unsatisfiedCNS2 = 0</td>
</tr>
<tr>
<td>2.2E+000dB</td>
<td>errorNum1 = 20</td>
<td>num_unsatisfiedCNS = 24</td>
<td>errorAfterFlp = 7</td>
<td>num_unsatisfiedCNS2 = 487</td>
</tr>
<tr>
<td>2.2E+000dB</td>
<td>errorNum1 = 15</td>
<td>num_unsatisfiedCNS = 22</td>
<td>errorAfterFlp = 5</td>
<td>num_unsatisfiedCNS2 = 487</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 13</td>
<td>num_unsatisfiedCNS = 14</td>
<td>errorAfterFlp = 0</td>
<td>num_unsatisfiedCNS2 = 0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 13</td>
<td>num_unsatisfiedCNS = 13</td>
<td>errorAfterFlp = 1</td>
<td>num_unsatisfiedCNS2 = 0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 19</td>
<td>num_unsatisfiedCNs = 34</td>
<td>errorAfterFlp = 17</td>
<td>num_unsatisfiedCNs2 = 481</td>
</tr>
<tr>
<td>------------</td>
<td>----------------</td>
<td>-------------------------</td>
<td>-------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 9</td>
<td>num_unsatisfiedCNs = 9</td>
<td>errorAfterFlp = 0</td>
<td>num_unsatisfiedCNs2 = 0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 8</td>
<td>num_unsatisfiedCNs = 8</td>
<td>errorAfterFlp = 1</td>
<td>num_unsatisfiedCNs2 = 4</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 10</td>
<td>num_unsatisfiedCNs = 17</td>
<td>errorAfterFlp = 6</td>
<td>num_unsatisfiedCNs2 = 470</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 12</td>
<td>num_unsatisfiedCNs = 26</td>
<td>errorAfterFlp = 11</td>
<td>num_unsatisfiedCNs2 = 489</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 24</td>
<td>num_unsatisfiedCNs = 42</td>
<td>errorAfterFlp = 26</td>
<td>num_unsatisfiedCNs2 = 490</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 38</td>
<td>num_unsatisfiedCNs = 66</td>
<td>errorAfterFlp = 37</td>
<td>num_unsatisfiedCNs2 = 362</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 23</td>
<td>num_unsatisfiedCNs = 25</td>
<td>errorAfterFlp = 6</td>
<td>num_unsatisfiedCNs2 = 21</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 8</td>
<td>num_unsatisfiedCNs = 8</td>
<td>errorAfterFlp = 0</td>
<td>num_unsatisfiedCNs2 = 0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 21</td>
<td>num_unsatisfiedCNs = 30</td>
<td>errorAfterFlp = 8</td>
<td>num_unsatisfiedCNs2 = 466</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 12</td>
<td>num_unsatisfiedCNs = 19</td>
<td>errorAfterFlp = 6</td>
<td>num_unsatisfiedCNs2 = 488</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 23</td>
<td>num_unsatisfiedCNs = 40</td>
<td>errorAfterFlp = 21</td>
<td>num_unsatisfiedCNs2 = 490</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 9</td>
<td>num_unsatisfiedCNs = 9</td>
<td>errorAfterFlp = 0</td>
<td>num_unsatisfiedCNs2 = 0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 11</td>
<td>num_unsatisfiedCNs = 12</td>
<td>errorAfterFlp = 0</td>
<td>num_unsatisfiedCNs2 = 0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 8</td>
<td>num_unsatisfiedCNs = 8</td>
<td>errorAfterFlp = 1</td>
<td>num_unsatisfiedCNs2 = 0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 8</td>
<td>num_unsatisfiedCNs = 8</td>
<td>errorAfterFlp = 1</td>
<td>num_unsatisfiedCNs2 = 4</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 9</td>
<td>num_unsatisfiedCNs = 9</td>
<td>errorAfterFlp = 0</td>
<td>num_unsatisfiedCNs2 = 0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 9</td>
<td>num_unsatisfiedCNs = 9</td>
<td>errorAfterFlp = 0</td>
<td>num_unsatisfiedCNs2 = 0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 10</td>
<td>num_unsatisfiedCNs = 10</td>
<td>errorAfterFlp = 0</td>
<td>num_unsatisfiedCNs2 = 0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>errorNum1 = 7</td>
<td>num_unsatisfiedCNs = 13</td>
<td>errorAfterFlp = 3</td>
<td>num_unsatisfiedCNs2 = 483</td>
</tr>
<tr>
<td>errorNum1</td>
<td>num_unsatisfiedCNs</td>
<td>errorAfterFlp</td>
<td>num_unsatisfiedCNs2</td>
<td>errorNum2</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------</td>
<td>---------------</td>
<td>---------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>18</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>52</td>
<td>21</td>
<td>467</td>
<td>344</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>15</td>
<td>1</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>21</td>
<td>6</td>
<td>491</td>
<td>376</td>
</tr>
<tr>
<td>9</td>
<td>14</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>13</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Details:**

- **2.4E+000dB**

- **errorNum1 = 8**
- **num_unsatisfiedCNs = 8**
- **errorAfterFlp = 0**
- **num_unsatisfiedCNs2 = 0**
- **errorNum2 = 0**

- **errorNum1 = 17**
- **num_unsatisfiedCNs = 18**
- **errorAfterFlp = 1**
- **num_unsatisfiedCNs2 = 5**
- **errorNum2 = 1**

- **errorNum1 = 28**
- **num_unsatisfiedCNs = 52**
- **errorAfterFlp = 21**
- **num_unsatisfiedCNs2 = 467**
- **errorNum2 = 344**

- **errorNum1 = 8**
- **num_unsatisfiedCNs = 9**
- **errorAfterFlp = 0**
- **num_unsatisfiedCNs2 = 0**
- **errorNum2 = 0**

- **errorNum1 = 13**
- **num_unsatisfiedCNs = 15**
- **errorAfterFlp = 1**
- **num_unsatisfiedCNs2 = 5**
- **errorNum2 = 0**

- **errorNum1 = 13**
- **num_unsatisfiedCNs = 21**
- **errorAfterFlp = 6**
- **num_unsatisfiedCNs2 = 491**
- **errorNum2 = 376**

- **errorNum1 = 9**
- **num_unsatisfiedCNs = 14**
- **errorAfterFlp = 3**
- **num_unsatisfiedCNs2 = 0**
- **errorNum2 = 0**

- **errorNum1 = 8**
- **num_unsatisfiedCNs = 8**
- **errorAfterFlp = 1**
- **num_unsatisfiedCNs2 = 0**
- **errorNum2 = 0**

- **errorNum1 = 6**
- **num_unsatisfiedCNs = 11**
- **errorAfterFlp = 1**
- **num_unsatisfiedCNs2 = 0**
- **errorNum2 = 0**

- **errorNum1 = 10**
- **num_unsatisfiedCNs = 10**
- **errorAfterFlp = 0**
- **num_unsatisfiedCNs2 = 0**
- **errorNum2 = 0**

- **errorNum1 = 10**
- **num_unsatisfiedCNs = 10**
- **errorAfterFlp = 0**
- **num_unsatisfiedCNs2 = 0**
- **errorNum2 = 0**

- **errorNum1 = 8**
- **num_unsatisfiedCNs = 8**
- **errorAfterFlp = 0**
- **num_unsatisfiedCNs2 = 0**
- **errorNum2 = 0**

- **errorNum1 = 9**
- **num_unsatisfiedCNs = 11**
- **errorAfterFlp = 0**
- **num_unsatisfiedCNs2 = 0**
- **errorNum2 = 0**

- **errorNum1 = 8**
- **num_unsatisfiedCNs = 8**
- **errorAfterFlp = 1**
- **num_unsatisfiedCNs2 = 0**
- **errorNum2 = 0**

- **errorNum1 = 8**
- **num_unsatisfiedCNs = 8**
- **errorAfterFlp = 0**
- **num_unsatisfiedCNs2 = 0**
- **errorNum2 = 0**

- **errorNum1 = 10**
- **num_unsatisfiedCNs = 10**
- **errorAfterFlp = 1**
- **num_unsatisfiedCNs2 = 0**
- **errorNum2 = 0**

- **errorNum1 = 9**
- **num_unsatisfiedCNs = 16**
- **errorAfterFlp = 4**
- **num_unsatisfiedCNs2 = 0**
- **errorNum2 = 0**

- **errorNum1 = 9**
- **num_unsatisfiedCNs = 9**
- **errorAfterFlp = 0**
- **num_unsatisfiedCNs2 = 0**
- **errorNum2 = 0**

- **errorNum1 = 9**
- **num_unsatisfiedCNs = 9**
- **errorAfterFlp = 0**
- **num_unsatisfiedCNs2 = 0**
- **errorNum2 = 0**

- **errorNum1 = 7**
- **num_unsatisfiedCNs = 13**
- **errorAfterFlp = 4**
- **num_unsatisfiedCNs2 = 0**
- **errorNum2 = 0**
<table>
<thead>
<tr>
<th></th>
<th>errorNum1</th>
<th>num_unsatisfiedCNs</th>
<th>errorAfterFlp</th>
<th>num_unsatisfiedCNs2</th>
<th>errorNum2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4E+000dB</td>
<td>5</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>14</td>
<td>15</td>
<td>2</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>79</td>
<td>54</td>
<td>54</td>
<td>400</td>
<td>245</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>6</td>
<td>6</td>
<td>3</td>
<td>420</td>
<td>283</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>14</td>
<td>14</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>21</td>
<td>21</td>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>16</td>
<td>16</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>13</td>
<td>16</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>12</td>
<td>14</td>
<td>2</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>9</td>
<td>17</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>12</td>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>473</td>
<td>346</td>
</tr>
<tr>
<td>Error Level</td>
<td>Error Num 1</td>
<td>Num Unsatisfied CNs</td>
<td>Error After Flip</td>
<td>Num Unsatisfied CNs 2</td>
<td>Error Num 2</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>---------------------</td>
<td>------------------</td>
<td>------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>13</td>
<td>13</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.4E+000dB</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>4</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>15</td>
<td>20</td>
<td>5</td>
<td>482</td>
<td>2</td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>20</td>
<td>22</td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>11</td>
<td>12</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>20</td>
<td>22</td>
<td>3</td>
<td>212</td>
<td>0</td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>9</td>
<td>14</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>7</td>
<td>10</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>25</td>
<td>35</td>
<td>12</td>
<td>481</td>
<td>352</td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>4</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>13</td>
<td>15</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>errorNum1</td>
<td>num_unsatisfiedCNs</td>
<td>errorAfterFlp</td>
<td>num_unsatisfiedCNs2</td>
<td>errorNum2</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-------------------</td>
<td>---------------</td>
<td>---------------------</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>8</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>35</td>
<td>22</td>
<td></td>
<td>491</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>13</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>9</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>8</td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>8</td>
<td>2</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>8</td>
<td>6</td>
<td></td>
<td>475</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>12</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>10</td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>11</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>12</td>
<td>2</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>8</td>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>8</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>9</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>9</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>10</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>8</td>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>8</td>
<td>17</td>
<td></td>
<td>484</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>32</td>
<td></td>
<td></td>
<td>374</td>
<td></td>
</tr>
<tr>
<td>errorNum1</td>
<td>num_unsatisfiedCNS</td>
<td>errorAfterFlp</td>
<td>num_unsatisfiedCNS2</td>
<td>errorNum2</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>5</td>
<td>13</td>
<td>6</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>10</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>7</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>14</td>
<td>15</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>14</td>
<td>14</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>20</td>
<td>21</td>
<td>3</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>14</td>
<td>16</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>7</td>
<td>12</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>8</td>
<td>17</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>errorNum1</td>
<td>num_unsatisfiedCNs</td>
<td>errorAfterFlp</td>
<td>num_unsatisfiedCNs2</td>
<td>errorNum2</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>--------------------</td>
<td>---------------</td>
<td>---------------------</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>14</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>13</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>2</td>
<td>386</td>
<td>235</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>12</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>errorNum1 = 9</td>
<td>num_unsatisfiedCNs = 9</td>
<td>errorAfterFlp = 0</td>
<td>num_unsatisfiedCNs2 = 0</td>
<td>errorNum2 = 0</td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>errorNum1 = 8</td>
<td>num_unsatisfiedCNs = 8</td>
<td>errorAfterFlp = 0</td>
<td>num_unsatisfiedCNs2 = 0</td>
<td>errorNum2 = 0</td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>errorNum1 = 10</td>
<td>num_unsatisfiedCNs = 10</td>
<td>errorAfterFlp = 0</td>
<td>num_unsatisfiedCNs2 = 0</td>
<td>errorNum2 = 0</td>
</tr>
<tr>
<td>2.6E+000dB</td>
<td>errorNum1 = 8</td>
<td>num_unsatisfiedCNs = 8</td>
<td>errorAfterFlp = 0</td>
<td>num_unsatisfiedCNs2 = 0</td>
<td>errorNum2 = 0</td>
</tr>
</tbody>
</table>