Technical University of Crete

Department of Electronic and Computer Engineering

Analysis and Design of LDPC Codes for the Relay Channel

By:

Alexios Balatsoukas-Stimming

Submitted on February 26th, 2010 in partial fulfilment of the requirements for the Electronic and Computer Engineering diploma degree.

Advisor:	Professor Athanasios Liavas
Co-advisor:	Assistant Professor Georgios Karystinos
Co-advisor:	Professor Nikolaos Sidiropoulos

Abstract

Since the invention of Information Theory by Shannon in 1948, coding theorists have been working on finding ways to achieve capacity. Recently, two capacity approaching code classes have emerged, namely LDPC and Turbo codes. In this thesis, we focus on LDPC codes. More precisely, we analyse encoding, decoding, design and construction of LDPC codes. The main design tool, Density Evolution, is presented in depth, and its application on the design problem is explained. A very good tool for constructing LDPC codes of moderate length, namely the Progressive Edge Growth algorithm, is also presented and support for irregular check node distributions and construction of codes in upper-triangular form for quick encoding is added. Furthermore, the state of the art of the coding problem for the relay channel is approached through the existing bibliography. More precisely, an analysis of Bilayer LDPC codes and Bilayer Density Evolution is made. This thesis is concluded by remarks and suggestions for improving the existing coding schemes for the relay channel and for constructing codes which can be used in practical relaying scenarios.

Περίληψη

Από την εφεύρεση της Θεωρίας Πληροφορίας από τον Shannon το 1948, η επιστήμη της Θεωρίας Κωδίκων ασχολείται με την εύρεση μεθόδων επίτευξης της χωρητικότητας του κάθε καναλιού. Πρόσφατα, ήρθαν στο προσκήνιο δυο κλάσεις κωδίκων που πλησιάζουν αυτή τη χωρητικότητα, οι κώδικες Turbo και οι κώδικες LDPC. Σε αυτή την εργασία, εστιάζουμε στους κώδικες LDPC. Πιο συγκεκριμένα, αναλύουμε την κωδικοποίηση και αποκωδικοποίηση, το σχεδιασμό και την κατασκευή LDPC κωδίκων. Το κύριο εργαλείο σχεδιασμού, ο αλγόριθμος Εξέλιξης Πυκνότητας (Density Evolution), αναλύεται σε βάθος και εξηγείται η εφαρμογή του στο πρόβλημα σχεδιασμού κωδίκων. Παρουσιάζεται επίσης ένα πολύ καλό εργαλείο κατασκευής LDPC κωδίκων μικρού και μεσαίου μήκους, ο αλγόριθμος Προοδευτικής Αύξησης Ακμών (Progressive Edge Growth), και επεκτείνονται οι δυνατότητες του με την υποστήριξη μη-κανονικών κατανομών βαθμών κόμβων ελέγχου (irregular check node degree distributions) και την υποστήριξη κατασκευής κωδίκων σε άνω τριγωνική μορφή για ταχεία κωδικοποίηση. Επιπροσθέτως, γίνεται μια προσέγγιση της παρούσας κατάστασης της χρήσης κωδικοποίησης για το κανάλι με αναμεταδότη (relay) μέσω της υπάρχουσας βιβλιογραφίας. Πιο συγκεκριμένα, γίνεται ανάλυση των LDPC κωδίκων δύο επιπέδων (Bilayer LDPC Codes) και του αλγόριθμου Εξέλιξης Πυκνότητας δυο επιπέδων (Bilayer Density Evolution). Η εργασία αυτή καταλήγει με παρατηρήσεις και προτάσεις για τη βελτίωση των υπάρχοντων τεχνικών κωδικοποίησης για το κανάλι με αναμεταδότη καθώς και για την κατασκευή κωδίκων που μπορούν να χρησιμοποιηθούν σε πρακτικά σενάρια αναμετάδοσης.

Acknowledgements

I would like to take some time here and thank the people who made the completion of this work possible.

First of all, my parents, without whom my entire studies would not have been possible. I would also like to thank my thesis supervisor, Professor Athanasios Liavas, for all his guidance, support and patience. Many thanks also go to my thesis committee, particularly to Assistant Professor George Karystinos who provided his experience and knowledge whenever needed.

In a broader circle, I would also like to thank, although it may never reach him, Professor David MacKay of Cambridge University for freely distributing his books, papers, tools and general information through his webpage. They were a valuable ally in my quest for knowledge. I am also very grateful to the authors of several tutorials, tools and demos which helped me understand many things which were a bit clouded in my mind.

"Quotation is a serviceable substitute for wit." - Oscar Wilde

Oops.

Contents

List of Figures

List of Tables

Preface

Many sections of this thesis are loosely based on Richardson and Urbanke's *Modern Coding Theory* [1]. Most of the code used to generate the figures in this thesis, along with some documentation, is available at `http://www.telecom.tuc.gr/~alex/`.

A Note on Notation

Apart from a nifty title, this section contains some useful comments on the notation used hereafter.

Newly introduced terminology will be *emphasised* the first time it appears.

Bold lowercase letters denote row vectors, e.g.:

$$\mathbf{u} = \begin{bmatrix} u_1 & u_2 & \dots & u_n \end{bmatrix}.$$

Bold uppercase letters denote matrices, e.g.:

$$\mathbf{G} = \begin{bmatrix} g_{11} & g_{11} & \cdots & g_{1n} \\ g_{21} & g_{22} & \cdots & g_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ g_{m1} & g_{m2} & \cdots & g_{mn} \end{bmatrix}.$$

As above, single matrix and vector elements will be denoted using regular, lowercase lettering, i.e.:

$$g_{ij} = 1.$$

Calligraphic uppercase letters denote sets, e.g.:

$$\mathcal{C} = \{\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_n\}.$$

Definitions, theorems, lemmas, corollaries and examples share a common counter within each chapter, e.g.:

Definition 0.1. This is a definition defining something.

Theorem 0.2. *Boys will be boys.*

Theorem proofs are omitted from the text to allow smooth reading.

Lemma 0.3. *My milkshakes bring all the boys to the yard.*

Corollary 0.4. *From theorem 0.2 it is obvious that $P \subset NP$.*

Example 0.5. Here be an example. And dragons. ■

The ■ symbol is used as an end-of-example symbol.

Matrix and vector multiplications involving binary matrices and vectors, are performed by substituting standard addition with the binary XOR operator, and standard multiplication with the binary AND operator.

We use the notation $f_x(x)$ and $p_x(x)$ (or simplified $p(x)$) interchangeably when it is obvious from the context if we are referring to a probability density function or a probability mass function.

Chapter 1

Motivation

1.1 Shannon's Theorem

The noisy channel coding theorem, first presented by Shannon in 1948 [2], states that for every communications channel, however noisy it may be, there exists a maximum, in most cases positive, rate, denoted R, at which information can be transmitted with vanishingly small probability of error. This is in contrast to what was believed until then, namely that the information rate must tend to zero in order for the probability of error to also tend to zero. The maximum rate at which reliable communication is achievable, is called the *channel capacity.*

A more formal statement of the theorem is:

Theorem 1.1 (Shannon's Theorem)**.** *For a discrete memoryless channel all rates below capacity C are achievable.*

Specifically, for every rate $R < C$, there exists a sequence of $(2^{nR}, n)$ codes with maximum probability of error $\lambda^{(n)} \to 0$.

Conversely, any sequence of $(2^{nR}, n)$ codes with $\lambda^{(n)} \to 0$ must have $R \leq C$.

Unfortunately, Shannon's proof is not constructive, as he did not provide ways to construct such codes but merely proved their existence. Since 1948, a whole scientific region (coding theory) has been dedicated to the design and implementation of codes for efficient communication. In the past few years, some practical code classes which can achieve transmission at rates very close to the channel capacity have been introduced. Two major representatives of this class of codes are LDPC and Turbo codes.

1.2 Why use LDPC Codes?

LDPC[1] codes were first invented by Robert Gallager in 1960 [3] but forgotten for over 30 years, mainly due to practical issues regarding their implementation. Since their revival in 1995 by David MacKay and Radford Neal [4], a very lively research community has been engaged with LDPC codes. Some important advances include, but are definitely not limited to, the Density Evolution algorithm for asymptotic behaviour analysis [5, 6], the introduction of irregular LDPC codes [7], a linear-time encoding algorithm for codes possessing a special structure [8], as well as ways of constructing finite-length LDPC codes with

[1]Sometimes referred to as *Gallager* codes, in their inspirer's honour.

very good performance [9, 10].

This section's title question arises naturally from the fact that many capacity-approaching code classes exist. The advantages of LDPC codes over other code classes, particularly Turbo codes which are their main rival, are numerous. First of all, LDPC codes are not patented like Turbo codes, making them a more attractive candidate for use in commercial products. Secondly, as we will see, very good algorithms for (suboptimal) decoding with linear time complexity exist. With certain structure, linear time encoding is also possible. Moreover, Turbo codes often exhibit error floors at relatively high error rates, whereas LDPC codes, when properly designed, can have significantly lower error floors. A more sophisticated and very important property of LDPC codes is that almost no undetected errors occur when decoding with the Belief Propagation algorithm, making them particularly suitable for Hybrid-ARQ applications.

Some very recent applications of LDPC codes include:

- 10GBase-T Ethernet (IEEE 802.3an)
- G.hn/G.9960 (ITU-T Standard for networking over power lines, phone lines and coaxial cable)
- DVB-S2 (Digital video broadcasting)
- WiMAX (IEEE 802.16e standard for microwave communications)
- IEEE 802.11n standard for Wireless LANs

Chapter 2

Introduction

2.1 Linear Block Codes

When using block coding, we perform a set of predefined actions on an information symbol sequence (block), generating extra symbols which are used for error detection and correction. We can safely restrict ourselves to codes consisting of binary elements, since most codes used in practice are indeed binary. However, the theory mentioned below can easily be generalised to codes with nonbinary symbols.

If we let $\mathbf{u}$ denote an information symbol block of length k, then a block code is a mapping from each vector $\mathbf{u}_i \in \{0,1\}^k$ to a vector $\mathbf{c}_i \in \{0,1\}^n$, $i \in \{1, 2, \ldots, 2^k\}$. For the code to be uniquely decodable, and thus useful, the $\mathbf{c}_i$'s have to be distinct. The set of all codewords is denoted $\mathcal{C}$. If no structure exists in the code, the encoder has to store 2^k codewords of length n, which means that the memory complexity is prohibitively large[1] even for relatively small n and k.

Linear block codes enjoy a low spatial encoding complexity due to their structure, justifying our great interest in them[2]. It is sensible to introduce some basic definitions for linear block codes before we proceed.

Definition 2.1. A block code of length n containing 2^k codewords is called a *linear* (n, k) code iff its 2^k codewords form a k-dimensional subspace of the vector space of all the binary n-tuples.

In simpler terms, any linear combination of codewords is also a codeword. This implies the following:

Theorem 2.2. *Each linear block code contains the all-zero codeword.*

Proof. Since any linear combination $\mathbf{c} = \mathbf{c}_i + \mathbf{c}_j$ of codewords is also a codeword, we can choose $\mathbf{c}_i = \mathbf{c}_j$, so that $\mathbf{c} = \mathbf{c}_i + \mathbf{c}_j = \mathbf{0}$. □

Definition 2.3. The *rate* of a code is defined as the fraction of information bits in each codeword. More precisely:

$$r \triangleq \frac{k}{n}$$

Definition 2.4. The *weight* of a binary codeword is defined as the sum of its elements, i.e. the total number of non-zero elements.

Definition 2.5. The *minimum distance* of a code is defined as the minimum weight of the codewords of which it consists.

[1] $\mathcal{O}(n2^k)$ to be precise.

[2] A second, more relevant to this thesis, reason is that LDPC codes are linear block codes.

Theorem 2.6 ([11], p. 79)**.** *A linear block code with minimum distance d, can correct at most $\lfloor \frac{d-1}{2} \rfloor$ errors using minimum-distance decoding. This bound is called the* error correcting capability *of the code.*

We will see in later chapters that other decoding schemes can, under certain circumstances, exceed the error correcting capability of a code.

Since the subspace spanned by the code is of dimension k, we can find k linearly independent codewords such that every codeword in $\mathcal{C}$ is a linear combination of those k codewords. Let those codewords be denoted $\mathbf{g}_0, \mathbf{g}_1, \ldots, \mathbf{g}_{k-1}$, and let us arrange them in a $k \times n$ matrix as follows:

$$\mathbf{G} = \begin{bmatrix} \mathbf{g}_0 \\ \mathbf{g}_1 \\ \vdots \\ \mathbf{g}_{k-1} \end{bmatrix}.$$

Then, we can generate all codewords as follows:

$$\mathbf{c} = \mathbf{u}\mathbf{G}, \qquad \mathbf{u} \in \{0,1\}^k$$

Definition 2.7. A $k \times n$ matrix $\mathbf{G}$ consisting of k linearly independent codewords of $\mathcal{C}$ is said to be a *generator matrix* for $\mathcal{C}$.

The above definition explains why linear codes have a low spatial encoding complexity: it is sufficient for the encoder to store k rows of length n, leading to a spatial complexity of $\mathcal{O}(kn)$.

Definition 2.8. Two generator matrices (and their corresponding codes) are said to be *equivalent*, if one can be derived from the other via elementary row operations and column permutations.

Definition 2.9. A linear block code is said to be *systematic* if its generator matrix has the following form:

$$\mathbf{G} = \begin{bmatrix} \mathbf{P} & \mathbf{I}_k \end{bmatrix}$$

where $\mathbf{P}$ is an arbitrary $k \times (n-k)$ binary matrix and $\mathbf{I}_k$ is a $k \times k$ identity matrix. We define $m \triangleq n-k$ as the number of redundant parity symbols.

The essence of definition 2.9 is that systematic codewords consist of a redundant checking part followed by the unaltered information bits. Codes possessing this structure have the advantage that the information bits can be easily extracted from the decoded codeword. The following theorem is particularly interesting:

Theorem 2.10. *Every linear block code has an equivalent systematic code representation.*

Another useful matrix associated with each linear block code is the *parity-check* matrix. For any matrix $\mathbf{G}$ with k linearly independent rows, there exists an $(n-k) \times n$ matrix with $(n-k)$ linearly independent rows such that any vector in the row space of $\mathbf{G}$ is orthogonal to the rows of $\mathbf{H}$. Using the parity-check matrix, we can define a code as follows

Definition 2.11. The code $\mathcal{C}$ consists of all vectors $\mathbf{c}$ satisfying $\mathbf{c}\mathbf{H}^T = \mathbf{0}$.

This means that a code imposes a set of even, linearly independent, parity checks on the codeword bits, which can be used for error detection and correction by solving the resulting system of equations.

Definition 2.12. A set of parity check equations $\mathcal{P}_i$ is said to be *orthogonal* on the codeword bit position i, if the codeword bit i is the only common bit of all check equations in $\mathcal{P}_i$.

Orthogonality is a property which is particularly useful for some decoding schemes, as we will see in later chapters.

Definition 2.13. Consider the transmission of a codeword $\mathbf{c} \in \mathcal{C}$ which is corrupted by noise. The receiver performs hard decisions on the received symbols, resulting in the vector $\mathbf{x}$ which does not necessarily belong to $\mathcal{C}$. Then, the vector $\mathbf{s} = \mathbf{c}\mathbf{H}^T$ is called the *syndrome* of the received vector. The received vector is a codeword if and only if $\mathbf{s} = \mathbf{0}$.

Definition 2.14. The 2^{n-k} linear combinations of the rows of matrix $\mathbf{H}$ form an $(n, n-k)$ linear code which is the nullspace of $\mathbf{G}$. This code, denoted $\mathcal{C}_d$, is called the *dual* code of $\mathcal{C}$.

A second useful property of systematic generator matrices is that their corresponding parity-check matrices can be easily computed as follows

$$\mathbf{H} = \begin{bmatrix}\mathbf{I}_{n-k} & \mathbf{P}^T\end{bmatrix}.$$

Example 2.15. This example illustrates most of the above definitions and properties. Consider the following generator matrix

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

If we swap columns 1 and 6, we get an equivalent generator matrix[3]

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

Note that $\mathbf{G}$ is now systematic, since $\mathbf{G} = \begin{bmatrix}\mathbf{P} & \mathbf{I}_4\end{bmatrix}$. To create a codeword from an arbitrary information bit vector, say $\mathbf{u} = \begin{bmatrix}1 & 1 & 1 & 0\end{bmatrix}$, we proceed as follows:

$$\mathbf{c} = \mathbf{u}\mathbf{G} = \begin{bmatrix}0 & 1 & 0 & 1 & 1 & 1 & 1 & 0\end{bmatrix} = \begin{bmatrix}\mathbf{p} & \mathbf{u}\end{bmatrix}$$

where $\mathbf{p}$ denotes the redundant checking part of the codeword. This codeword's weight is 5.

Using the method described above, we can calculate the parity-check matrix of the code

$$\mathbf{H} = \begin{bmatrix}\mathbf{I}_3 & \mathbf{P}^T\end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}.$$

The parity checks imposed by this particular code are

$$c_1 \oplus c_4 \oplus c_5 \oplus c_6 = 0$$
$$c_2 \oplus c_4 \oplus c_5 \oplus c_6 = 0$$
$$c_3 \oplus c_5 \oplus c_6 \oplus c_7 = 0.$$

Calculating $\mathbf{c}\mathbf{H}^T$ we get

$$\mathbf{c}\mathbf{H}^T = \begin{bmatrix}0 & 0 & 0\end{bmatrix},$$

as expected. ■

This concludes our small introduction to linear block codes. Some notions, such as *weight enumerators* and *covering radius* were not discussed here since they are not of much relevance to LDPC codes. More on linear block codes for the interested reader can be found at [11, Chapter 3].

[3]This code is known as a (7,4) Hamming code.

2.2 LDPC Codes

As the name implies, Low-Density Parity-Check codes are linear block codes which have a parity-check matrix with a low density of non-zero entries, i.e. a sparse matrix. The definition of “low density” is a bit loose, but generally matrices with less than 10% non-zero entries are considered to be sparse. The number of non-zero entries in a sparse matrix increases linearly with respect to matrix dimensions, and not quadratically as in regular (dense) matrices. This parity-check matrix can be stored very efficiently using sparse matrix techniques.

The first LDPC codes, introduced by Gallager, were *regular* codes. This means that a constant number of symbols participate in each check equation, and each symbol participates in a constant number of check equations. In terms of the parity-check matrix **H**, this means that the column and row weights are constant. As shown in [7], much better performance can be achieved by allowing irregular variable and check node degrees. The analysis of the following chapters is general and can be applied to both cases.

2.3 Graph Representation of LDPC Codes

We can construct a graphical representation of an LDPC code using *Tanner graphs* [12].

Definition 2.16. A bipartite graph (or bigraph) is a graph whose vertices can be divided into two disjoint sets U and V, such that every edge connects a vertex in U to one in V.

Tanner graphs are bipartite graphs, and in our case we have two sets containing *variable nodes* and *check nodes*. Variable nodes correspond to codeword bits and check nodes correspond to check equations.

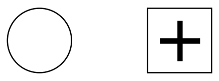


Figure 2.1: Symbols used for representing variable and check nodes.

Definition 2.17. The *degree* of a node is defined as the number of edges connected to that node.

In figure 2.22, for example, the topmost variable node has degree 1 and the topmost check node has degree 4.

We can construct a Tanner graph from an $m \times n$ parity-check matrix as follows:

1. Add n numbered variable nodes to the variable node set and m numbered check nodes to the check node set.
2. Add an edge between check node m_i and variable node n_j if and only if the corresponding entry h_{ij} in **H** is equal to 1.

Step 2 actually illustrates the fact that the codeword bit n_j participates in check equation m_i. More generally, a ’1’ at position i in row j of the parity-check matrix, indicates a connection between codeword bit i and parity-check equation j. Consequently, each row indicates which bits participate in the corresponding check equation, and each column indicates which check equations the corresponding bit participates in. Some useful properties of bipartite graphs are:

Definition 2.18. A *cycle* is a path on the graph such that the start vertex and the end vertex are the same.

Theorem 2.19. *Bipartite graphs can not contain odd-lengthed cycles.*

Proof. The proof stems directly from the definition of bipartite graphs. Choose a node from the cycle at random and define it as the beginning of the cycle. If this node is a variable node, we have to traverse an even number of edges in order to reach another variable node, so we also have to traverse an even number of edges in order to reach the beginning of the cycle. The same holds if this node is a check node. □

Definition 2.20. The minimum cycle length of a graph is called the *girth* of the graph.

Theorem 2.21. *The minimum girth of a Tanner graph is* 4.

Proof. Double edges between nodes are not allowed in Tanner graphs, excluding cycles of length 2. Using theorem 2.19, the next even positive integer after 2, is 4. □

Example 2.22. For this example we will use the parity-check matrix of example 2.15. This particular matrix is not sparse since it has a total of 28 elements and 13 non-zero entries, but it suffices for our current needs. Recall that:

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

We can construct the Tanner graph corresponding to **H** by simply following the two steps defined above:

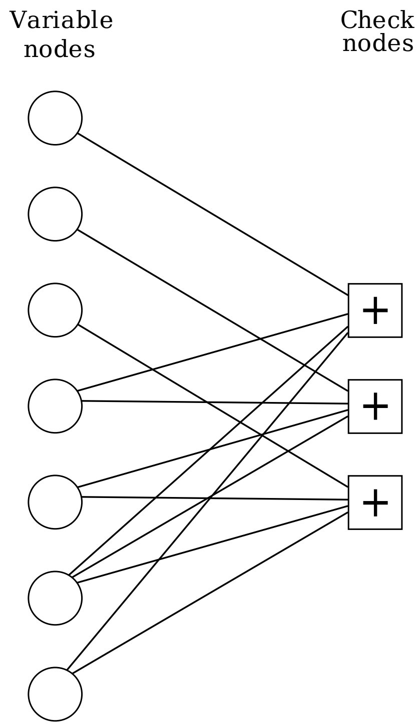


Figure 2.2: Tanner graph corresponding to the $(7, 4)$ Hamming code.

The edge connecting the topmost variable node with the topmost check node, corresponds to the entry $h_{11} = 1$. All other edges are connected accordingly. ■

As we will see, graph representation of LDPC codes plays a vital role in decoding, since the most widely used decoding algorithm is in fact an algorithm for marginalising functions, which can be implemented using a certain graph representation. We will also see that, for the decoding algorithm to work well, it is essential for the graph to have as large a girth as possible. Equivalent representations of the same code can differ significantly in terms of girth, so a particularly interesting and important problem is finding the best possible representation.

2.4 Degree Distributions

Assume that an LDPC code has length n and that the number of variable nodes of degree i is Λ_i, so $\sum_i \Lambda_i = n$. In the same fashion, the number of check nodes of degree i is P_i, so $\sum_i P_i = n(1-r) = m$, where r is the *design rate* of the code. The following compact notation is more convenient

$$\Lambda(x) = \sum_{i=1}^{l_{\max}} \Lambda_i x^i, \qquad P(x) = \sum_{i=1}^{r_{\max}} P_i x^i.$$

From the above definitions, we get the following useful relationships:

$$\Lambda(1) = n, \quad P(1) = n(1-r), \quad r(\Lambda, P) = 1 - \frac{P(1)}{\Lambda(1)}, \quad \Lambda'(1) = P'(1).$$

The first four relationships should be obvious. If we rewrite the left part of the last one as

$$\Lambda'(1) = \sum_{i=1}^{l_{\max}} i\Lambda_i,$$

we see that this sum represents the total number of connections on the variable node side. It is natural that this number should be equal to the number of connections on the check node side. $\Lambda(x)$ and $P(x)$ are called the *check and variable degree distributions* from a *node perspective.* Sometimes it is more convenient to use the corresponding normalised distributions

Definition 2.23. The *normalised variable node degree distribution* from a node perspective is defined as

$$L(x) = \frac{\Lambda(x)}{\Lambda(1)} = \sum_{i=1}^{l_{\max}} L_i x^i.$$

Definition 2.24. The *normalised check node degree distribution* from a node perspective can be represented as

$$R(x) = \frac{P(x)}{P(1)} = \sum_{i=1}^{r_{\max}} R_i x^i.$$

We can also define the distributions from an *edge perspective*, meaning that the polynomial coefficients represent the fraction of edges connected to degree i nodes. This definition is particularly useful for the asymptotic behaviour analysis, as we will see.

Definition 2.25. The *normalised variable node degree distribution* from an edge perspective can be represented as

$$\lambda(x) = \frac{L'(x)}{L'(1)} = \sum_{i=1}^{l_{\max}} \lambda_i x^{i-1}.$$

Definition 2.26. The *normalised check node degree distribution* from an edge perspective can be represented as

$$\rho(x) = \frac{R'(x)}{R'(1)} = \sum_{i=1}^{l_{\max}} \rho_i x^{i-1}.$$

The average variable and check node degrees can be calculated as follows

$$l_{\text{avg}} = \frac{1}{\int_0^1 \lambda(x)dx}, \qquad r_{\text{avg}} = \frac{1}{\int_0^1 \rho(x)dx}.$$

They are particularly useful for defining the design rate of a degree distribution

$$r(\lambda, \rho) = 1 - \frac{l_{\text{avg}}}{r_{\text{avg}}} = 1 - \frac{\int_0^1 \rho(x)dx}{\int_0^1 \lambda(x)dx}.$$

Example 2.27. The degree distributions for the $(7, 4)$ Hamming code are:

$$\Lambda(x) = 3x + 3x^2 + x^3, \qquad P(x) = 4x^4$$

$$L(x) = \frac{3}{7}x + \frac{3}{7}x^2 + \frac{1}{7}x^3, \qquad R(x) = x^4$$

$$\lambda(x) = \frac{3}{12} + \frac{6}{12}x + \frac{3}{12}x^2, \qquad \rho(x) = x^3.$$

2.5 Code Ensembles

Now, let us take a first glance at the usefulness of the previous section's endless list of definitions.

Definition 2.28 (The standard ensemble LDPC(Λ, P))**.** Given a degree distribution pair (Λ, P), define an *ensemble* of bipartite graphs LDPC(Λ, P) in the following way. Each graph in LDPC(Λ, P) has $\Lambda(1)$ variable nodes and $P(1)$ check nodes. Λ_i (P_i) variable (check) nodes have degree i. A node of degree i has i *sockets* from which the i edges emanate, so that in total there are $\Lambda'(1) = P'(1)$ sockets on each side. Label the sockets on each side using the elements of the set $\ell = \{1, \cdots, \Lambda'(1)\}$ in some arbitrary but fixed way. Let σ be a permutation on ℓ. Associate a bipartite graph by connecting the i-th socket on the variable side to the $\sigma(i)$-th socket on the check side. Letting σ run over the set of permutations on ℓ generates a set of bipartite graphs. Finally, define a probability distribution over the set of graphs by placing the uniform probability distribution on the set of permutations. This is the ensemble of bipartite graphs LDPC(Λ, P).

It remains to associate a code with each element of LDPC(Λ, P). We will do so by associating a parity-check matrix to each graph. Because of possible multiple edges and since the encoding is done over binary symbols, we define the parity-check matrix as the $\{0, 1\}$-matrix that has a non-zero entry at row i and column j if the i-th check node is connected to the j-th variable node an odd number of times. Even numbers of connections lead to terms in the check equations which cancel each other out, since we apply the XOR operation, which is equivalent to the given variable node not taking part in the check equation at all.

Since with every graph we can associate a code, we refer to both graphs and codes as being elements of LDPC(Λ, P). A subtle point is that the derived graphs are not labelled, meaning that their distribution is not necessarily the same as the one of the permutations. If in the sequel we say that we pick a code uniformly at random we really mean that we pick a graph at random from the ensemble of graphs and consider the induced code.

We know that (Λ, P), (n, L, R), and (n, λ, ρ) contain equivalent information, so we can freely switch between these perspectives, using the one that is more convenient in each case.

There is no guarantee that all check equations implied by the induced parity-check matrix are linearly independent, so the code's actual rate might be higher than the design rate of the ensemble. The following lemma asserts that, under some conditions, the actual rate of a code chosen at random from an ensemble is arbitrarily close to the design rate with the probability gap to 1 converging exponentially to 0 in the blocklength.

Lemma 2.29 ([1], Lemma 3.22, p. 80)**.** *Consider the ensemble LDPC(n, λ, ρ). Let $r(\lambda, \rho)$ denote the design rate of the ensemble and let $r(G)$ denote the actual rate of a randomly chosen code $G \in$ LDPC(n, λ, ρ). Consider the function $\Psi(y)$:*

$$\Psi(y) = -\Lambda'(1)\log_2\left[\frac{1+yz}{1+z}\right] + \sum_i L_i \log_2\left[\frac{1+y^i}{2}\right] + \frac{L'(1)}{R'(1)}\sum_j R_j \log_2\left[1 + \left(\frac{1-z}{1+z}\right)^i\right]$$

with

$$z = \left(\sum_i \frac{\lambda_i y^{i-1}}{1+y^i}\right) / \left(\sum_i \frac{\lambda_i}{1+y^i}\right).$$

Assume that for $y \geq 0$ it holds that $\Psi(y) \leq 0$ with equality only at $y = 1$. Then for $\xi > 0$ and $n \geq n(\xi)$, sufficiently large

$$\mathbb{P}\{r(G) - r(\lambda, \rho) > \xi\} \leq e^{-n\xi \ln(2)/2}.$$

Example 2.30. For the $(7, 4)$ Hamming code, from the previous example, we know that:

$$\lambda_H(x) = \frac{3}{12} + \frac{6}{12}x + \frac{3}{12}x^2, \qquad \rho_H(x) = x^3.$$

In addition, this code's length is 7, so it belongs to the $(7, \lambda_H, \rho_H)$ ensemble. ■

Using code ensembles, we can refer to a whole family of codes by their degree distributions. As we will see, the behaviour of codes belonging to the same ensemble is densely concentrated around the ensemble average performance, which makes their analysis more convenient.

We also present a simulation which compares a regular and an irregular code, in order to illustrate the claim that irregular codes perform better than regular codes. We use BPSK modulation with the mapping $\{0, 1\} \to \{-1, +1\}$. Noise variance is defined as

$$\sigma^2 = \frac{N_0}{2}$$

where

$$\mathrm{SNR}_{\mathrm{dB}} = 10\log_{10}\left(\frac{1}{N_0}\right).$$

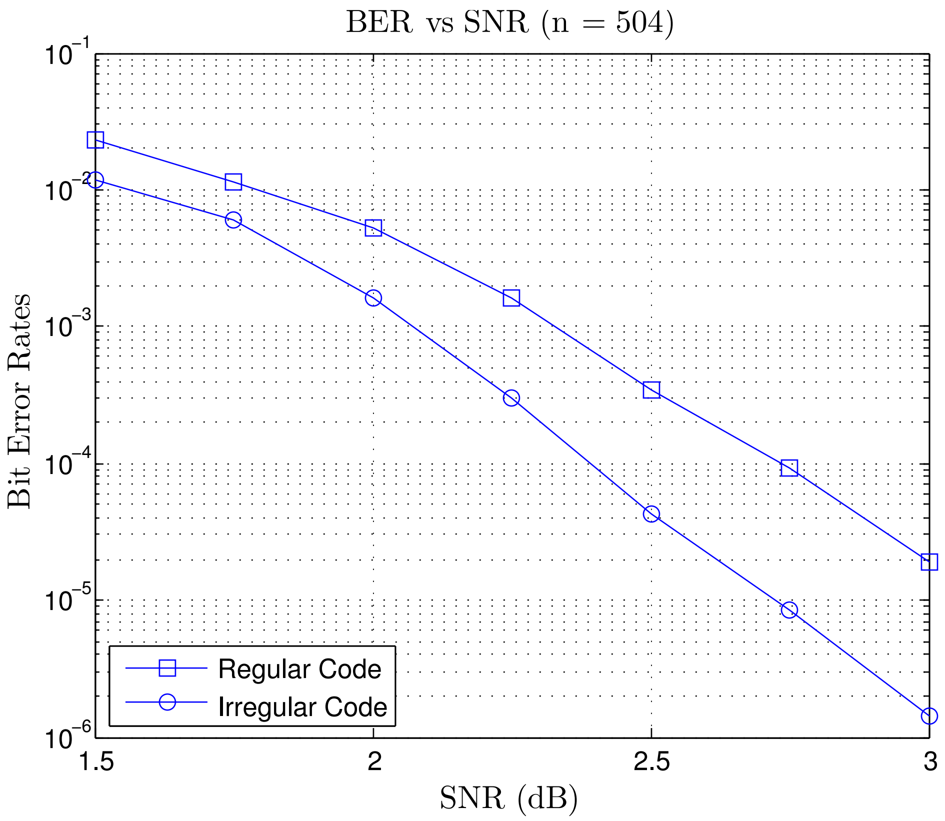


Figure 2.3: A regular and an irregular code, both with length $n = 504$ bits and rate 0.5.

The regular code is created from the $(3, 6)$ ensemble using the PEG algorithm. We choose the following variable node degree distribution (from a node perspective) for the irregular code from [13]

$$\Lambda(x) = 0.477081x^2+0.280572x^3+0.0349963x^4+0.0963301x^5+0.0090884x^7+0.00137443x^{14}+0.10055777x^{15}$$

and let PEG make the check node degree distribution as concentrated as possible. The maximum number of iterations of the belief propagation algorithm is set to 120. Clearly, the irregular code performs better. As the blocklength is increased, the irregular code's superiority in performance will be even more significant.

Chapter 3

Encoding of LDPC Codes

3.1 General Case

As with any linear code, an LDPC code can be encoded by simply multiplying the vector corresponding to the information bits we wish to encode with the generator matrix, i.e.

$$\mathbf{c} = \mathbf{u}\mathbf{G}, \qquad \mathbf{u} \in \{0,1\}^k.$$

Since LDPC codes are usually defined through their parity-check matrix, rather than their generator matrix, introducing a way to encode making sole use of the parity-check matrix would be useful. Recall that all codewords satisfy

$$\mathbf{c}\mathbf{H}^T = \mathbf{0}.$$

Now, if we find a decomposition of $\mathbf{H}$ using column permutations

$$\mathbf{H} = \begin{bmatrix} \mathbf{H}_p & \mathbf{H}_s \end{bmatrix}$$

such that $\mathbf{H}_p$ is $m \times m$ and invertible, we can split the vector $\mathbf{c}$ into a systematic and a parity part

$$\mathbf{c} = \begin{bmatrix} \mathbf{c}_p & \mathbf{c}_s \end{bmatrix}.$$

Then, combining the parity-check equation with the above decomposition, we get

$$\mathbf{c}_p\mathbf{H}_p^T = \mathbf{c}_s\mathbf{H}_s^T.$$

If we fill $\mathbf{c}_s$ with the information bits, we can then solve for the parity bits

$$\mathbf{c}_p = \mathbf{c}_s\mathbf{H}_s^T(\mathbf{H}_p^T)^{-1}.$$

A straightforward construction of such an encoder is to use Gaussian elimination and column permutations to bring $\mathbf{H}$ into an equivalent[1] upper triangular form. Then, we can solve for the m parity bits using back substitution. More precisely

$$c_{p,i} = \sum_{j=i+1}^{m} h_{i,j}c_{p,j} + \sum_{j=1}^{k} h_{i,j+m}c_{s,j}.$$

Unfortunately, matrix sparsity is generally not preserved by Gaussian elimination, so, in general, the resulting equivalent matrix will be dense, leading to an $\mathcal{O}(n^2)$ encoding complexity.

[1]Gaussian elimination is in fact right multiplication by an invertible matrix, so the resulting code is equivalent with the original one.

3.2 Linear Complexity Encoding

Since the parity-check matrix is sparse, it has $\mathcal{O}(n)$ non-zero entries. This leads to the question if there are cases where encoding in time linearly proportional to n is possible. As shown in [8], such cases do exist and we can achieve efficient encoding using the approach presented below.

If we manipulate the parity-check matrix using row and column permutations only, the resulting matrix will be sparse. However, it is not always possible to transform the matrix to an upper triangular form using this approach. We therefore introduce the *approximate upper triangular* form

$$\mathbf{H} = \begin{bmatrix} \mathbf{T} & \mathbf{A} & \mathbf{B} \\ \mathbf{E} & \mathbf{C} & \mathbf{D} \end{bmatrix}$$

where $\mathbf{T}$ is square and upper triangular. If we can make $\mathbf{T}$ of dimension $m \times m$, then encoding with linear time complexity is possible. In the general case where $\mathbf{T}$ is of dimension $(m-g)$, encoding is possible with complexity $\mathcal{O}(n+g^2)$, where g is called the *gap*. The dimensions of the matrices are

$$\begin{array}{lll} \mathbf{T} : (m-g) \times (m-g) & \mathbf{A} : (m-g) \times g & \mathbf{B} : (m-g) \times k \\ \mathbf{E} : g \times (m-g) & \mathbf{C} : g \times g & \mathbf{D} : g \times k. \end{array}$$

If we multiply $\mathbf{H}$ from the left by

$$\begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{E}\mathbf{T}^{-1} & \mathbf{I} \end{bmatrix}$$

we get

$$\begin{bmatrix} \mathbf{T} & \mathbf{A} & \mathbf{B} \\ \mathbf{0} & \mathbf{C} - \mathbf{E}\mathbf{T}^{-1}\mathbf{A} & \mathbf{D} - \mathbf{E}\mathbf{T}^{-1}\mathbf{B} \end{bmatrix}.$$

We have eliminated $\mathbf{E}$ and will now solve the system of equations directly, rather than completing Gaussian elimination. We further decompose our codeword as

$$\mathbf{c} = \begin{bmatrix} \mathbf{c}_{p1} & \mathbf{c}_{p2} & \mathbf{c}_s \end{bmatrix}$$

with $\mathbf{c}_{p1}$ having length $(m-g)$ and $\mathbf{c}_{p2}$ having length g. The parity-check equation $\mathbf{c}\mathbf{H}^t = \mathbf{0}$ then splits into two equations, namely

$$\mathbf{c}_{p1}\mathbf{T}^T + \mathbf{c}_{p2}\mathbf{A}^T + \mathbf{c}_s\mathbf{B}^T = \mathbf{0}$$

$$\mathbf{c}_{p2}(\mathbf{C} - \mathbf{E}\mathbf{T}^{-1}\mathbf{A})^T + \mathbf{c}_s(\mathbf{D} - \mathbf{E}\mathbf{T}^{-1}\mathbf{B})^T = \mathbf{0}.$$

We define $\mathbf{\Phi} = (\mathbf{C} - \mathbf{E}\mathbf{T}^{-1}\mathbf{A})^T$ and suppose that it is invertible (or equivalently that $\mathbf{H}_p$ is invertible). Then:

$$\mathbf{c}_{p2} = \mathbf{c}_s(\mathbf{D} - \mathbf{E}\mathbf{T}^{-1}\mathbf{B})^T\mathbf{\Phi}^{-1}.$$

As shown in [8], $\mathbf{c}_{p2}$ can be calculated with complexity $\mathcal{O}(n+g^2)$.

After calculating $\mathbf{c}_{p2}$, we can substitute and evaluate for $\mathbf{c}_{p1}$

$$\mathbf{c}_{p1} = (\mathbf{c}_{p2}\mathbf{A}^T + \mathbf{c}_s\mathbf{B}^T)(\mathbf{T}^T)^{-1}.$$

As shown in [8], $\mathbf{c}_{p1}$ can be calculated with complexity $\mathcal{O}(n)$. So, we have effectively reached the conclusion that the overall encoding complexity is $\mathcal{O}(n+g^2)$. If the gap is of order $\sqrt{n}$, we can achieve encoding with linear time complexity even if the gap is non-zero. In the more general case, the constant factor in front of the quadratic term is usually small, meaning that, for sufficiently small g, encoding remains manageable even for large blocklengths, despite not being linear.

If we can bring the parity-check matrix to a *perfect*[2] upper triangular form, the whole encoding scheme collapses to the calculation of

$$\mathbf{c}_{p1} = \mathbf{c}_s \mathbf{B}^T (\mathbf{T}^T)^{-1}.$$

The sparse matrix $\mathbf{V} = \mathbf{B}^T(\mathbf{T}^T)^{-1}$ can be precalculated, so it is easy to see that multiplying the source vector with $\mathbf{V}$ can be accomplished with complexity $\mathcal{O}(n)$. In the following chapters, we will see that a very efficient algorithm for constructing parity-check matrices, which are guaranteed to be in perfect upper triangular form *and* have large girth, exists. So, it is safe to say that we can construct practical codes which do exhibit the linear time complexity encoding property while still having good performance.

We will now give a sufficient condition on a degree distribution pair (λ, ρ) to give rise to linear encoding complexity.

Theorem 3.1 ([1], Theorem A.18, p. 449)**.** *Let (λ, ρ) be a degree distribution pair satisfying $1 - z - \rho(1 - \lambda(z)) > 0$ for $z \in (0, 1)$, $r_{\min} > 2$ and the strict inequality $\lambda'(0)\rho'(1) > 1$. Let G be chosen uniformly at random from the ensemble LDPC(n, λ, ρ). Then, there exists a strictly positive number c such that for each $k \in \mathbb{N}$ the probability that $g \leq k$ is asymptotically (in the blocklength) lower bounded by $1 - (1 - c)^k$.*

3.3 Concluding Remarks

In the above analysis, we focused on time complexity. Another equally important aspect of encoding complexity, is hardware[3] complexity. Special code structures, such as Quasi-Cyclic codes, exist which aim to minimise the hardware complexity of the encoder. However, we are mainly interested in software implementations of coding schemes using LDPC codes, so this aspect will not be analysed further here.

[2]We use the term *perfect* to emphasise the difference between the usual upper triangular form and the approximate upper triangular form.

[3]Memory elements, shift registers, logical units etc.

Chapter 4

Decoding of LDPC Codes

4.1 Introduction

In communications systems, the presence of various types of noise corrupts the received signal. Channel codes aim to utilise the dependencies between codeword bits imposed by the code's parity checks, in order to detect and correct errors.

Suppose that we transmit a codeword $\mathbf{x} \in \mathcal{C}$ with probability $p(\mathbf{x})$ over a channel with transition probability $p(\mathbf{y}|\mathbf{x})$, where $\mathbf{y}$ generally belongs to $\mathbb{R}^n$. If the decision rule $\mathbf{c} = \hat{\mathbf{x}}(\mathbf{y}) \in \mathcal{C}$ is used, the resulting block error probability[1] is $1 - p(\hat{\mathbf{x}}(\mathbf{y})|\mathbf{y})$. So, in order to minimise the block error probability, we wish to maximise $p(\hat{\mathbf{x}}(\mathbf{y})|\mathbf{y})$. Thus, the *maximum a posteriori probability* (MAP) decoding rule reads:

$$\begin{aligned}
\hat{\mathbf{x}}(\mathbf{y})^{\text{MAP}} &= \arg\max_{\mathbf{c}\in\mathcal{C}} p(\mathbf{c}|\mathbf{y}) && \text{(Bayes' rule)} \\
&= \arg\max_{\mathbf{c}\in\mathcal{C}} p(\mathbf{y}|\mathbf{c})\frac{p(\mathbf{c})}{p(\mathbf{y})} && \text{(Maximization does not depend on } \mathbf{y}\text{)} \\
&= \arg\max_{\mathbf{c}\in\mathcal{C}} p(\mathbf{y}|\mathbf{c})p(\mathbf{c}).
\end{aligned}$$

If the codewords are equiprobable, then

$$\begin{aligned}
\hat{\mathbf{x}}(\mathbf{y})^{\text{MAP}} &= \arg\max_{\mathbf{c}\in\mathcal{C}} p(\mathbf{y}|\mathbf{c})\frac{1}{|\mathcal{C}|} \\
&= \arg\max_{\mathbf{c}\in\mathcal{C}} p(\mathbf{y}|\mathbf{c}) \\
&= \hat{\mathbf{x}}(\mathbf{y})^{\text{ML}}
\end{aligned}$$

where $|\mathcal{C}|$ is the cardinality of $\mathcal{C}$. So, in this case, the MAP and Maximum-Likelihood (ML) rules are equivalent.

For hard-decision decoding schemes, the MAP decoder is equivalent to the *mininum-distance* or *nearest-neighbour* decoder, which simply decodes the received vector as the nearest, in terms of Hamming distance (i.e. the number of elements at which the two vectors differ), codeword.

Optimal (i.e. ML) decoding of linear block codes is generally an NP-complete problem. Fortunately, many feasible decoding algorithms exist. The decoding methods analysed in the following chapters range

[1]Also referred to as *frame* error probability.

from low to high decoding complexity and from reasonably good to very good error performance. They provide a wide range of trade-offs among decoding complexity, decoding speed and error performance. Majority Logic (MLG) and Bit Flipping (BF) decoding are hard-decision decoding schemes and can be easily implemented. MLG decoding has the least decoding delay and can achieve very high decoding speed. Weighted MLG and BF decoding are hard-decision decoding schemes which make use of some reliability information for each decision, thus landing somewhere between hard-decision and soft-decision decoding and offering a good trade-off between computational complexity and performance. The Belief Propagation (BP) algorithm is a soft-decision decoding scheme. It requires higher computational complexity but provides the best error performance among the presented suboptimal decoding schemes. An extensive example at the end of this chapter will demonstrate instances of all presented decoding algorithms.

4.2 Syndrome Decoding

The hard-decision ML problem can be reduced to a slightly simpler form for linear codes, due to their structure. Recall that the syndrome of a received vector $\mathbf{y}$ is defined as

$$\mathbf{s} = \mathbf{y}\mathbf{H}^T.$$

Now, define the error pattern $\mathbf{e}$ as

$$\mathbf{e} \triangleq \mathbf{y} \oplus \mathbf{c}$$

where $\mathbf{c}$ denotes the codeword which was actually sent. So, $\mathbf{e}$ is simply a vector which contains 1s in the positions where an error has occurred. The *standard array* of a linear block code $\mathcal{C}$ is created as follows and contains all 2^n binary vectors of length n:

$$\begin{array}{cccc}
\mathbf{c}_0 = \mathbf{0} & \mathbf{c}_1 & \dots & \mathbf{c}_{2^k-1} \\
\mathbf{e}_0 & \mathbf{c}_1 + \mathbf{e}_0 & \dots & \mathbf{c}_{2^k-1} + \mathbf{e}_0 \\
\mathbf{e}_1 & \mathbf{c}_1 + \mathbf{e}_1 & \dots & \mathbf{c}_{2^k-1} + \mathbf{e}_1 \\
\vdots & & & \vdots \\
\mathbf{e}_{2^{(n-k)}-1} & \mathbf{c}_1 + \mathbf{e}_{2^{(n-k)}-1} & \dots & \mathbf{c}_{2^k-1} + \mathbf{e}_{2^{(n-k)}-1}
\end{array}$$

where $\mathbf{c}_i \in \mathcal{C}$ and $\mathbf{e}_i$, $i \in \{0, 1, \dots, 2^{n-k} - 1\}$, are the error patterns. The elements of the first column are called *coset leaders*. The error patterns are not 2^n like one might expect for a codeword of length n, but $2^n - 2^k$. This happens because the remaining error patterns are in fact codewords and, since our code is linear, they simply transform the received codeword into another codeword. This type of error can not be corrected, in fact, it can not even be detected by the syndrome since

$$\mathbf{s} = (\mathbf{c}_i + \mathbf{c}_j)\mathbf{H}^T = \mathbf{c}_k\mathbf{H}^T = \mathbf{0}, \qquad \forall \mathbf{c}_i, \mathbf{c}_j, \mathbf{c}_k \in \mathcal{C},$$

so the receiver does not detect the error.

The key to syndrome decoding is the fact that all rows of the standard array have the same syndrome. Take for example the second row

$$(\mathbf{e}_0 + \mathbf{c}_j)\mathbf{H}^T = \mathbf{e}_0\mathbf{H}^T + \mathbf{c}_j\mathbf{H}^T = \mathbf{e}_0\mathbf{H}^T, \quad \forall \mathbf{c}_j \in \mathcal{C}.$$

By computing the syndrome of the received vector, we can estimate which error pattern occurred, namely the error pattern which has the same syndrome as the received vector. Call the estimate of the error pattern $\hat{\mathbf{e}}$. The optimal choice of coset leaders consists of the $2^{(n-k)} - 1$ most likely error patterns. After estimating the error pattern, we can decode the received vector to a codeword as follows

$$\hat{\mathbf{c}} = \mathbf{y} \oplus \hat{\mathbf{e}}.$$

A review of the steps of the algorithm is presented below.

1. Calculate the syndrome of the received vector $\mathbf{y}$.
2. Find the coset leader $\hat{\mathbf{e}}$ which has a syndrome equal to the syndrome of the received vector.
3. Decode the received vector as follows: $\hat{\mathbf{c}} = \mathbf{y} \oplus \hat{\mathbf{e}}$

Figure 4.1: The Syndrome Decoding Algorithm.

The syndrome decoder has the best performance among all hard-decision decoding schemes presented here. This performance, of course, comes at a cost: the decoder has to store $2^{(n-k)}$ binary vectors of length n, which is not very practical especially considering the fact that block codes require large blocklengths[2] to perform well. Clearly, we need to find some feasible way of decoding the received vector without sacrificing too much in terms of performance.

4.3 Bit-Flipping Decoding

Bit-Flipping decoding was introduced by Gallager along with the invention of LDPC codes in [3]. We already know that if the received vector $\mathbf{y}$, after hard decisions have been made, is not a codeword, some parity checks will fail, i.e. $\mathbf{y} = \mathbf{c}\mathbf{H}^T \neq \mathbf{0}$. The main idea behind the BF algorithm is choosing all variables which are contained in more than some fixed number δ of unsatisfied parity check equations and flipping their values. Using these new values, the parity checks are recomputed and the procedure is repeated. Usually we simply choose to flip the variables which are contained in the largest number of unsatisfied parity check equations. When all parity checks are satisfied, or the maximum allowed number of iterations is reached, decoding halts declaring successful decoding or a failure respectively.

Using the above information, the algorithm can be briefly described as follows.

```
while $\mathbf{s} \neq \mathbf{0}$ and maximum iteration number has not been reached do
    1. Find the number of unsatisfied parity check equations for each codeword bit, denoted $f_i$,
       $i \in \{0, 1, \ldots, n-1\}$.
    2. Identify the set $\Omega$ of bits for which $f_i$ is the largest.
    3. Flip the bits in $\Omega$.
    4. Recompute the parities based on the new values of the flipped bits.
    5. Increase the number of iterations by 1.
end while
```

Figure 4.2: The Bit-Flipping Algorithm.

The number of operations required for each iteration of the BF algorithm is linearly proportional to the blocklength of the code. Due to the nature of low density parity check codes, the above decoding algorithm corrects many error patterns which contain a number of errors exceeding the error correcting capability of the code.

4.3.1 Weighted BF Decoding

A significant loss in performance is introduced by the hard decisions required for the algorithm to work, since information is lost in the process. To combat this loss of information, a measure of reliability of the hard decisions can be introduced.

[2]In the order of at least some thousands of bits.

For the AWGN channel, a simple measure of realiability is the magnitude of each received symbol $\mathbf{y}_i$, denoted $|\mathbf{y}_i|$, $i \in \{0, 1, \ldots, n-1\}$. Define $|\mathbf{y}_{\min}|$ as

$$|\mathbf{y}_{\min}|_j \triangleq \{\min_i |\mathbf{y}_i| : h_{ji} = 1, 0 \leq i \leq n-1\}.$$

In other words, $|\mathbf{y}_{\min}|_j$ is the minimum reliability of all codeword bits participating in check equation j. We now define the weighted check sum E_i for the i-th codeword bit as follows

$$E_i \triangleq \sum_{s_j \in S_i} (1 - 2s_j)|\mathbf{y}_{\min}|_j$$

where S_i is the set of check equations orthogonal on the codeword bit i. Using the above information, the BF algorithm can be modified to include reliability information as follows:

```
while $\mathbf{s} \neq \mathbf{0}$ and maximum iteration number has not been reached do
    1. Compute $E_i$ for $i \in \{0, 1, \ldots, n-1\}$.
    2. Identify the set $\Omega$ of bits for which $E_i$ is the smallest (i.e. the “most wrong” bits).
    3. Flip the bits in $\Omega$.
    4. Recompute the parities based on the new values of the flipped bits.
    5. Increase the number of iterations by 1.
end while
```

Figure 4.3: The Weighted Bit-Flipping Algorithm.

This instance of the BF algorithm has an increased complexity since real additions are needed to calculate the weighted check sums, whereas the simple BF algorithm requires only binary operations. However, the increase in performance can be significant [14, Figures 1, 3, 5] if an appropriate reliability measure is chosen. Complexity is still linearly proportional to the blocklength of the code, although with a larger constant.

4.4 One-Step Majority-Logic Decoding

Let $\mathbf{y}$ denote the received vector after hard decisions have been made. Then, we know that if errors have occurred, $\mathbf{s} = \mathbf{y}\mathbf{H}^T \neq \mathbf{0}$, meaning that some parity check equations are not satisfied. The idea behind the majority logic decoder is that each codeword bit is assigned such a value as to satisfy as many of its parity check equations as possible. So, if most of its parity checks are unsatisfied, we have to flip its value, if not, we keep its original value. If we denote the set of parity checks corresponding to bit i as $\mathcal{P}_i$, then the update rule for each codeword bit is

$$\hat{\mathbf{y}}_i = \mathbf{y}_i \oplus \text{majority}(\mathcal{P}_i)$$

where $\oplus$ denotes modulo-2 addition and $\hat{\mathbf{y}}_i$ denotes the updated value for codeword bit i. For this procedure to work correctly, $\mathcal{P}_i$ has to be orthogonal on codeword bit i.

Another way of describing MLG decoding is through the code's Tanner graph. All variable nodes send their received values to their corresponding check nodes. When decoding variable node i, denoted v_i, check node $\mathcal{P}_i^{(j)}$, $j \in \{0, 1, \ldots, \deg(v_i)\}$, sends the modulo-2 sum of all its incoming values, except for the one originating from v_i, back to v_i. v_i is then decoded as the majority of its incoming messages. Orthogonality can be ensured by removing 4-cycles from the Tanner graph. Some code construction methods which ensure this property are presented in Chapter 6. An overview of the algorithm is presented below:

```
for $i = 0$ to $n-1$ do
    1. Calculate the value of the majority of $\mathcal{P}_i$.
    2. Assign the value $\mathbf{y}_i \oplus \text{majority}(\mathcal{P}_i)$ to codeword bit $i$.
    3. Recalculate the parity check equations based on the new value of $\mathbf{y}_i$.
end for
```

Figure 4.4: The One-Step Majority-Logic Decoding Algorithm.

The main difference between the MLG decoder and the BF decoder is that MLG decoding is performed in one step and on all codeword bits at once, whereas BF decoding is iterative and only operates on one codeword bit (or a fraction of codeword bits) at a time and not on the whole codeword.

4.4.1 Weighted MLG Decoding

Weighted MLG decoding was introduced in [15] to combat the loss of information introduced by the hard decisions required for the regular MLG algorithm to work. Instead of simply calculating the value of each parity check equation, a weighted checksum is first calculated, exactly like in the case of the weighted BF algorithm. Recall that for the AWGN channel we have

$$E_i \triangleq \sum_{s_j \in S_i} (1 - 2s_j)|\mathbf{y}_{\min}|_j.$$

Then, we can use the following simple rule to decide the value of parity check equation j

$$p_i = \begin{cases} 1, & \text{for } E_i > 0, \\ 0, & \text{for } E_i \leq 0. \end{cases}$$

The decoding procedure then continues as with the regular MLG algorithm, so this is the only point where soft information is used.

Since we have a simple loop which is linearly proportional to the blocklength of the code, this algorithm's complexity is also linearly proportional to the blocklength. It is, however, more complex than the MLG algorithm since real additions and multiplications have to be calculated in order to evaluate the weighted check sums. On the upside, performance can be significantly increased, as we can see in [14, Figures 1, 3, 5].

An overview of the algorithm is presented below:

```
for $i = 0$ to $n-1$ do
    1. Calculate the value of the majority of $\mathcal{P}_i$.
    2. Assign the value $\mathbf{y}_i \oplus \text{majority}(\mathcal{P}_i)$ to codeword bit $i$.
    3. Recalculate the parity check equations based on the new value of $\mathbf{y}_i$, using weighted check sums.
end for
```

Figure 4.5: The Weighted One-Step Majority-Logic Decoding Algorithm.

4.5 Belief Propagation Decoding

Assume that we transmit over a binary-input ($x_i \in \{-1, +1\}$) memoryless ($p(\mathbf{y}|\mathbf{x}) = \prod_{i=1}^{n} p(y_i|x_i)$) channel using a linear code $\mathcal{C}$ defined by its parity-check matrix $\mathbf{H}$ and assume that codewords are equiprobable.

The rule for the *bit-wise* MAP decoder reads

$$\begin{aligned}
\hat{x}_i^{\mathrm{MAP}}(\mathbf{y}) &= \arg \max_{x_i \in \{\pm 1\}} p(x_i|\mathbf{y}) && \text{(Law of total probability)} \\
&= \arg \max_{x_i \in \{\pm 1\}} \sum_{\sim x_i} p(\mathbf{x}|\mathbf{y}) && \text{(Bayes' rule)} \\
&= \arg \max_{x_i \in \{\pm 1\}} \sum_{\sim x_i} p(\mathbf{y}|\mathbf{x}) p(\mathbf{x}) \\
&= \arg \max_{x_i \in \{\pm 1\}} \sum_{\sim x_i} \prod_{j=1}^{n} p(y_i|x_i) \mathbb{1}_{[\mathbf{x} \in \mathcal{C}]}
\end{aligned}$$

where $\sum_{\sim x_i}$ denotes summation over all variables except x_i. Summation is done over $\mathbf{x}$ rather than $\mathbf{y}$, since the latter is the channel observation (i.e. a constant). For the last step, we used the fact that the channel is memoryless and that all codewords are equiprobable. The indicator function $\mathbb{1}_{[\mathbf{x} \in \mathcal{C}]}$ is defined as

$$\mathbb{1}_{[\mathbf{x} \in \mathcal{C}]} = \begin{cases} 1, & \mathbf{x} \in \mathcal{C}, \\ 0, & \text{otherwise.} \end{cases}$$

Assume that this code membership function has a factorised form, i.e. it can be written as a product of functions. Then, from the last equation, is it clear that the bit-wise decoding problem is equivalent to calculating the marginal of a factorised function and choosing the value that maximises this marginal.

Example 4.1. Consider the $(7, 4)$ Hamming code with parity-check matrix

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

Then, the code membership function can be written as:

$$\mathbb{1}_{[\mathbf{x} \in \mathcal{C}]} = \mathbb{1}_{[x_1 \oplus x_4 \oplus x_6 \oplus x_7 = 0]} \mathbb{1}_{[x_2 \oplus x_4 \oplus x_5 \oplus x_6 = 0]} \mathbb{1}_{[x_3 \oplus x_5 \oplus x_6 \oplus x_7 = 0]}$$

which simply means that for a vector $\mathbf{x}$ to be a codeword, all parity-check equations have to be satisfied. ■

We can calculate $\sum_{\sim x_i}$ by the brute force approach. This approach, however, requires $\mathcal{O}(2^n)$ calculations, where n is the codeword length. It is obvious that this naive solution is not practical.

4.5.1 The Distributive Law

The key to simplifying the calculation of a factorised function's marginal is the *distributive law*:

$$ab + ac = a(b + c), \qquad a, b, c \in \mathbb{F}$$

where $\mathbb{F}$ denotes any field (e.g. $\mathbb{R}$, the field of real numbers). Take, for example, the sum $\sum_{i,j} a_i b_j$ which can be rewritten as $(\sum_i a_i)\left(\sum_j b_j\right)$, greatly reducing the number of calculations needed for its evaluation. More precisely, if we assume that a, b take on values from the same alphabet $\mathcal{X}$ with $|\mathcal{X}| = n$, then the number of summations needed is reduced from n^2 to $2n$.

If we wish to marginalise a generic function $g(z, \ldots)$ over z, taking advantage of its generic factorisation

$$g(z, \ldots) = \prod_{k=1}^{K} [g_k(z, \ldots)]$$

we can apply the distributive law once

$$g(z) = \underbrace{\sum_{\sim z} \prod_{k=1}^{K} [g_k(z, \ldots)]}_{\text{sum of products}} = \underbrace{\prod_{k=1}^{K} \sum_{\sim z} [g_k(z, \ldots)]}_{\text{product of sums}} .$$

This already decreases complexity significantly, since the sums within the product are over less variables than the original sum. Note that while z appears in all factors, all other variables can appear in only *one* factor[3]. In order to further reduce complexity, we can apply the distributive law again for $g_k(z, \ldots)$ if it has a factorisation of the form

$$g_k(z, \ldots) = \underbrace{h(z, z_1, \ldots, z_J)}_{\text{kernel}} \prod_{j=1}^{J} \underbrace{[h_j(z_j, \ldots)]}_{\text{factors}} .$$

Then, $g_k(z)$ can be calculated as follows

$$\begin{aligned} g_k(z) &= \sum_{\sim z} g_k(z, \ldots) \\ &= \sum_{\sim z} h(z, z_1, \ldots, z_J) \prod_{j=1}^{J} [h_j(z_j, \ldots)] \\ &= \sum_{\sim z} h(z, z_1, \ldots, z_J) \prod_{j=1}^{J} \left[\sum_{\sim z_j} [h_j(z_j, \ldots)] \right] . \end{aligned} \tag{4.1}$$

In words, the desired marginal can be computed by multiplying the kernel with the individual marginals and summing out all variables except z. This process can proceed recursively until no further factorisations are possible.

Example 4.2. Let us see an example illustrating this recursive factorisation. Consider a function f with factorisation

$$f(x_1, x_2, x_3, x_4, x_5, x_6) = f_1(x_1, x_2, x_3) f_2(x_1, x_4, x_6) f_3(x_4) f_4(x_4, x_5)$$

and suppose that we wish to compute that marginal of f with respect to x_1. We denote this marginal $f(x_1)$

$$f(x_1) = \sum_{\sim x_1} f(x_1, x_2, x_3, x_4, x_5, x_6).$$

Assume that all variables take values in the same finite alphabet, $\mathcal{X}$. Determining $f(x_1)$ for all values of x_1 by brute force requires $\mathcal{O}(|\mathcal{X}|^6)$ operations, where $|\mathcal{X}|$ is the cardinality of the alphabet $\mathcal{X}$.

If we apply the distributive law once, taking advantage of the factorisation, we will have

$$f(x_1) = \left[\sum_{x_2, x_3} f_1(x_1, x_2, x_3) \right] \left[\sum_{x_4, x_5, x_6} f_2(x_1, x_4, x_6) f_3(x_4) f_4(x_4, x_5) \right] .$$

This reduced the cost of calculating the marginal to $\mathcal{O}(|\mathcal{X}|^4)$ since we need $\mathcal{O}(|\mathcal{X}|^3)$ operations to calculate the second sum for each of the $|\mathcal{X}|$ values of x_1. This reduction is, of course, very welcome, but we can

[3]This is equivalent to the condition that the resulting factor graph is a *tree*.

do even better. The first term can not be further expanded, but the second is in a factorised form, so we can apply the distributive law again to get

$$f(x_1) = \left[\sum_{x_2,x_3} f_1(x_1,x_2,x_3)\right]\left[\sum_{x_4} f_3(x_4)\left(\sum_{x_6} f_2(x_1,x_4,x_6)\right)\left(\sum_{x_5} f_4(x_4,x_5)\right)\right].$$

Then, for each x_1, we need $\mathcal{O}(|\mathcal{X}|^2)$ operations for the first factor. For the second factor, we need $\mathcal{O}(|\mathcal{X}|)$ operations for the sums over x_5 and x_6, which are then summed again over x_4, resulting in a total complexity of $\mathcal{O}(|\mathcal{X}|^2)$. Since there are $|\mathcal{X}|$ values for x_1, the calculation of the marginal for each one of these values has a total complexity of $\mathcal{O}(|\mathcal{X}|^3)$ operations, which is far better than $\mathcal{O}(|\mathcal{X}|^6)$ of the brute force approach. ■

4.5.2 Factor Graphs

Factor graphs provide an appropriate framework to systematically take advantage of the distributive law and, consequently, factorisations of functions like the one we saw in the previous example.

Consider a function and its corresponding factorisation. Associate with this factorisation a factor graph as follows. For each variable, draw a variable node (circle) and for each factor draw a factor node (square). Connect a variable node to a factor node by an edge if and only if the corresponding variable appears in this factor. The resulting graph is bipartite. For the function of example 4.2, the corresponding factor graph can be seen below.

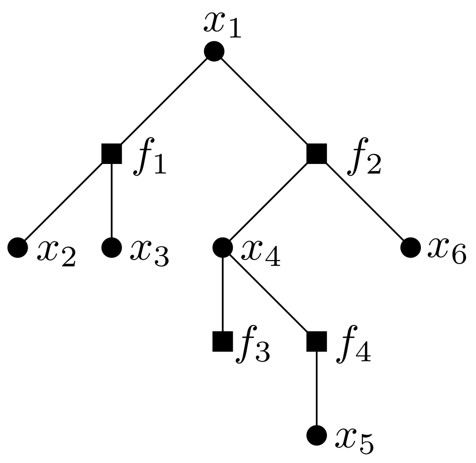


Figure 4.6: Factor graph corresponding to the factorisation of $f(x_1,x_2,x_3,x_4,x_5,x_6)$ of example 4.2.

This particular graph is a tree, i.e. it has no cycles. The factor graph representation can, naturally, be also applied to a factorised code membership function. In figure 4.7, we depict the factor graph corresponding to the membership function defined in example 4.1:

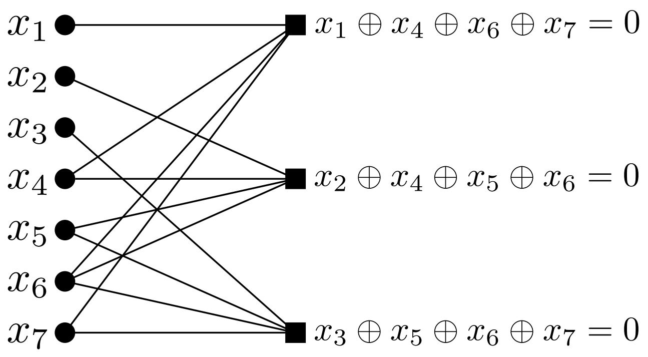

x_1
x_2
x_3
x_4
x_5
x_6
x_7
$x_1 \oplus x_4 \oplus x_6 \oplus x_7 = 0$
$x_2 \oplus x_4 \oplus x_5 \oplus x_6 = 0$
$x_3 \oplus x_5 \oplus x_6 \oplus x_7 = 0$

Figure 4.7: Factor graph corresponding to the factorisation of $\mathbb{1}_{[\mathbf{x}\in\mathcal{C}]}$ of example 4.1.

Note that this factor graph is in fact the code's Tanner graph. We will show in the following sections that, for factor graphs that are trees, marginals can be computed efficiently (and exactly) by *message-passing* algorithms. So, efficient computation of the marginal probability mass (or density) functions we need for bit-wise MAP decoding is possible, under some conditions, as we will see.

4.5.3 Marginalisation on Factor Graphs

We have already seen that, under the condition that the factor graph is a tree, the marginalisation problem can be broken down into smaller and smaller subproblems according to the structure of the tree.

This gives rise to the following efficient message-passing algorithm. The algorithm proceeds by sending messages along the edges of the tree. The messages signify marginals of parts of the function and these parts are combined to form the marginal of the whole function. Message passing starts at leaf nodes. Messages are passed up the tree and as soon as a node has received messages from all its children, the incoming messages are processed and the result is passed up to the parent node. At variable nodes, we simply perform multiplication of the incoming messages, and at function nodes processing is described by equation 4.1.

An aspect of the process which has to be considered is its initialisation. A function leaf node has the generic form $g_k(z)$, so that $\sum_{\sim z} g_k(z) = g_k(z)$, meaning that the initial message sent by a function leaf node is the function itself. To find out the correct initialisation at a variable leaf node consider the simple example of computing $f(x_1) = \sum_{\sim x_1} f(x_1, x_2)$. Here, x_2 is the variable leaf node. By the message passing rule in (4.1), the marginal $f(x_1)$ is equal to $\sum_{\sim x_1} f(x_1, x_2) \cdot \mu(x_2)$, where $\mu(x_2)$ is the message received from variable node x_2. We see that to get the correct result, this initial message should be the constant function 1.

Example 4.3. We will use the same function $f(x_1, x_2, x_3, x_4, x_5, x_6)$ which we used in previous examples to illustrate the calculation of the marginal $f(x_1)$ via message-passing. Edges whose corresponding messages have been sent, are depicted in red color. The labels of the edges denote the message that has been sent along each one. In figure (a) we see the factor graph corresponding to the factorisation of f. The initialisation step can be seen in figure (b). Variable nodes send the value 1 to their parent nodes, and function nodes send their function to their parent nodes. Now, function node f_1 has received all its incoming messages and can calculate its outgoing message $\sum_{\sim x_1} f_1(x_1, x_2, x_3)$. The same is true for function node f_4, which can send the message $\sum_{\sim x_4} f_4(x_4, x_5)$ to its parent node, as depicted in figure (c).

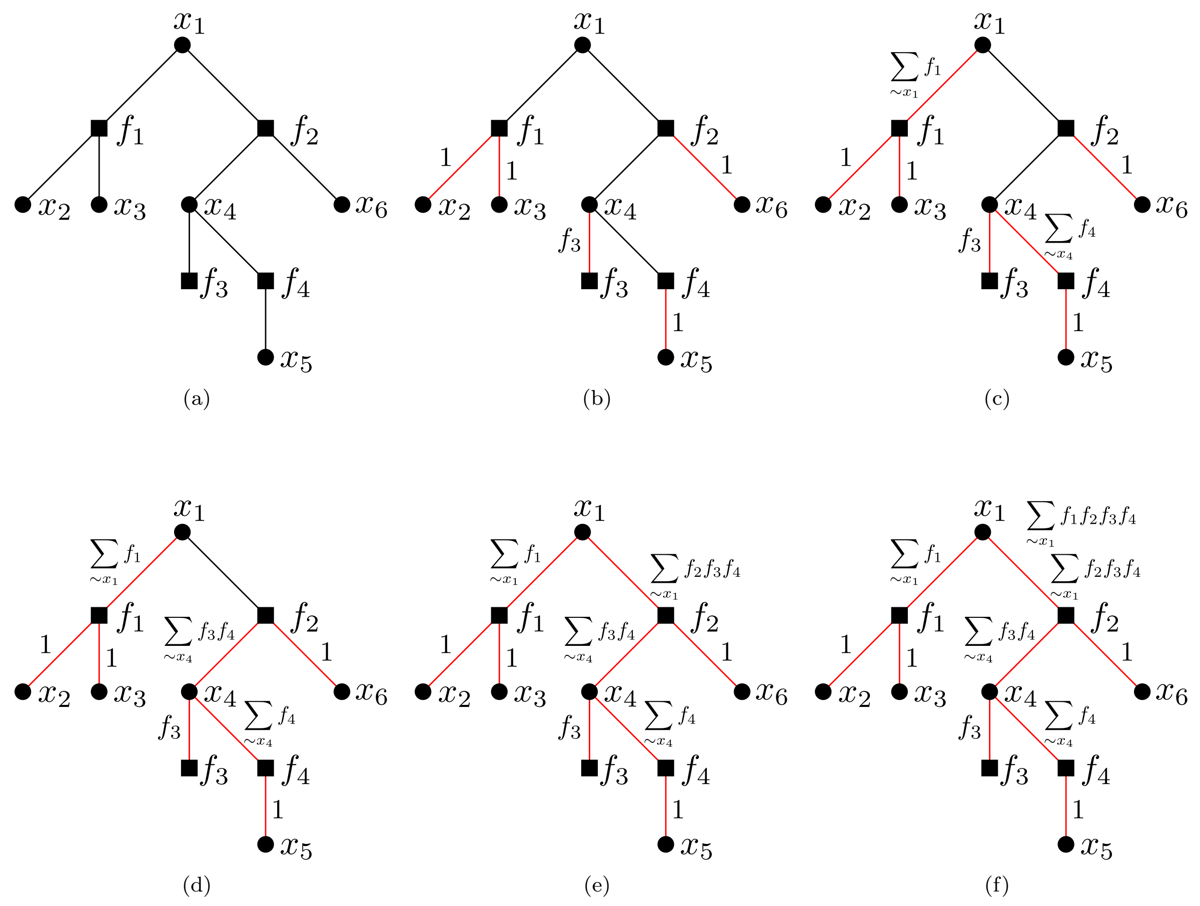
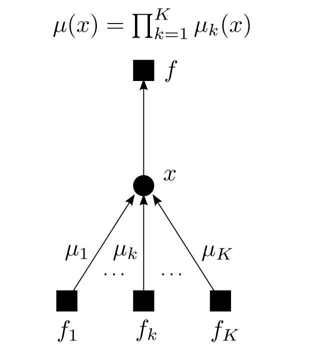


Figure 4.8: Calculation of the marginal $f(x_1)$ via message passing.

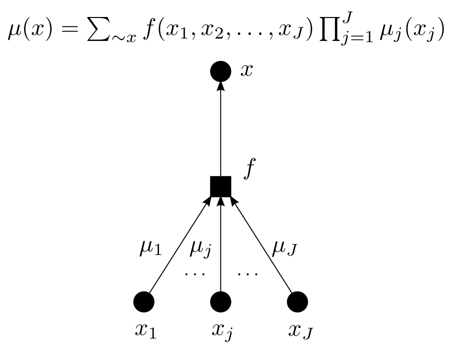
In figure (d), we see that variable node x_4 has received its incoming messages and can calculate their product $f_3(x_4)\sum_{\sim x_4} f_4(x_4,x_5) = \sum_{\sim x_4} f_3(x_4)f_4(x_4,x_5)$, which is then sent along the edge connecting it to function node f_2. Since function node f_2 now has all its incoming messages available, it can calculate its outgoing message $\sum_{\sim x_1} f_2 f_3 f_4$ which is then sent to the root variable node x_1 as seen in figure (e). The final step is for variable node x_1 to calculate the product of all its incoming messages, as seen in figure (f). So, we have successfully marginalised f with respect to x_1, demonstrating the message-passing algorithm. ■

It is worth noting at this point that so far we have considered the marginalisation of a function with respect to a single variable x_1. In practice, however, we are interested in marginalising a function over all variables at once, i.e. in the case of bit-wise MAP decoding. We could simply create a different tree for each variable and apply message-passing on them. It is easy to see, however, that the algorithm does not depend on which node is the root of the tree and that all the computations can in fact be performed simultaneously on a single tree. Simply start at the leaf nodes as usual, and for every edge compute the outgoing message along this edge as soon as you have received the incoming messages along all other edges that connect to the given node. Continue until messages have been sent across all edges in both directions. This calculates all marginals at once, with a slightly higher complexity than calculating a single marginal. This approach has a much lower complexity than applying message-passing to a different tree for each variable node since many messages, which would be exactly the same on all trees, are only calculated once.

The rules for message processing at variable and function nodes can be seen in figure 4.9.



(a) Variable Node Processing.



$\mu(x) = \sum_{\sim x} f(x_1, x_2, \ldots, x_J) \prod_{j=1}^{J} \mu_j(x_j)$
x
f
μ_1
μ_j
μ_J
x_1
x_j
x_J

(b) Function Node Processing.

Figure 4.9: Factor graph node processing rules.

The final marginalisation step for variable node x can be seen in figure 4.10.

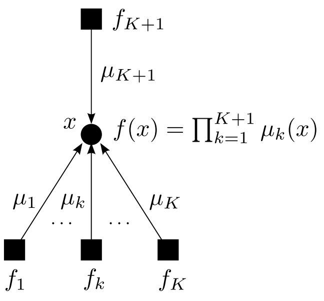


Figure 4.10: Final marginalisation step for variable x.

4.5.4 The Belief Propagation Algorithm

The algorithm that we will describe in the sequel, comes with many names, mainly depending on the context in which it is used; Sum-Product Algorithm (SPA) and Belief Propagation (BP) are the most common. We will henceforth refer to it as Belief Propagation. In the following section, we will derive the analytic expressions for the message update rules at variable and function nodes of an acyclic factor graph (i.e. a tree) for binary signalling. Recall that the bit-wise MAP decoding rule can be written as

$$\hat{x}_i^{\text{MAP}}(\mathbf{y}) = \arg \max_{x_i \in \{\pm 1\}} \sum_{\sim x_i} \prod_{j=1}^{n} p(y_i|x_i) \mathbb{1}_{[\mathbf{x} \in \mathcal{C}]}.$$

So, apart from the function nodes representing the code membership function, we will also have one function node connected to each variable node which represents $p(y_i|x_i)$, i.e. the effect of the channel.

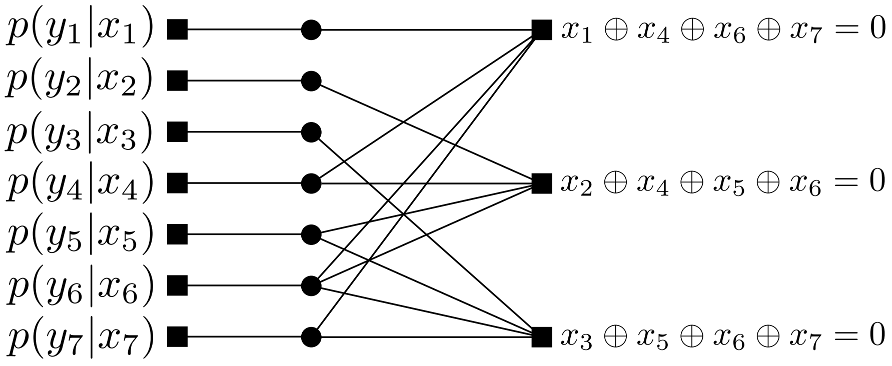

$p(y_1|x_1)$
$p(y_2|x_2)$
$p(y_3|x_3)$
$p(y_4|x_4)$
$p(y_5|x_5)$
$p(y_6|x_6)$
$p(y_7|x_7)$
$x_1 \oplus x_4 \oplus x_6 \oplus x_7 = 0$
$x_2 \oplus x_4 \oplus x_5 \oplus x_6 = 0$
$x_3 \oplus x_5 \oplus x_6 \oplus x_7 = 0$

Figure 4.11: Extension of the factor graph of figure 4.7 which includes the effect of the channel.

Variable Nodes

In the binary case (we use the bit values ± 1 instead of 0 and 1) a message $\mu(x)$ can be though of as a real-valued vector of length 2, $(\mu(1), \mu(-1))$. The initial such message sent from the factor leaf node representing the i-th channel realisation to the variable node i is $(p(y_i|1), p(y_i, -1))$. Recall that at a variable node of degree $K+1$ the message passing rule calls for a pointwise multiplication

$$\mu(1) = \prod_{k=1}^{K} \mu_k(1), \qquad \mu(-1) = \prod_{k=1}^{K} \mu_k(-1).$$

Introduce the ratio $r_k = \frac{\mu_k(1)}{\mu_k(-1)}$. These ratios are the likelihood ratios associated with the channel observations. We have

$$r = \frac{\mu_k(1)}{\mu_k(-1)} = \frac{\mu(1) = \prod_{k=1}^{K} \mu_k(1)}{\mu(-1) = \prod_{k=1}^{K} \mu_k(-1)} = \prod_{k=1}^{K} r_k$$

i.e., the ratio of the outgoing message at a variable node is the product of the incoming ratios. If we define the log-likelihood ratio $l_k = \ln(r_k)$, then the processing rule is simplified to $l = \sum_{k=1}^{K} l_k$. The log-likelihood ratio is widely used in practice, since the usage of the logarithm function simplifies the processing rule and increases numerical stability.

Check Nodes

At function nodes (henceforth referred to as check nodes, to comply with the terminology used for Tanner graphs) the processing rule is slightly more complex. Consider the ratio of an outgoing message at a check node which has degree $J+1$. For a check node the associated kernel is

$$f(x, x_1, \ldots, x_J) = \mathbb{1}_{[\prod_{j=1}^{J} x_j = x]}.$$

We have a product instead of a modulo-2 sum since we have assumed that x_i takes values in $\{\pm 1\}$ and not in $\{0,1\}$. For the outgoing message of a check node we have

$$\begin{aligned} r = \frac{\mu(1)}{\mu(-1)} &= \frac{\sum_{\sim x} f(1, x_1, \ldots, x_J) \prod_{j=1}^J \mu_j(x_j)}{\sum_{\sim x} f(-1, x_1, \ldots, x_J) \prod_{j=1}^J \mu_j(x_j)} \\ &= \frac{\sum_{x_1,\ldots,x_J : \prod_{j=1}^J \mu_j(x_j)=1} \prod_{j=1}^J \mu_j(x_j)}{\sum_{x_1,\ldots,x_J : \prod_{j=1}^J \mu_j(x_j)=-1} \prod_{j=1}^J \mu_j(x_j)} \\ &= \frac{\sum_{x_1,\ldots,x_J : \prod_{j=1}^J \mu_j(x_j)=1} \prod_{j=1}^J \frac{\mu_j(x_j)}{\mu(-1)}}{\sum_{x_1,\ldots,x_J : \prod_{j=1}^J \mu_j(x_j)=-1} \prod_{j=1}^J \frac{\mu_j(x_j)}{\mu(-1)}} \\ &= \frac{\sum_{x_1,\ldots,x_J : \prod_{j=1}^J \mu_j(x_j)=1} \prod_{j=1}^J r_j^{(1+x_j)/2}}{\sum_{x_1,\ldots,x_J : \prod_{j=1}^J \mu_j(x_j)=-1} \prod_{j=1}^J r_j^{(1+x_j)/2}} \\ &= \frac{\prod_{j=1}^J (r_j+1) + \prod_{j=1}^J (r_j-1)}{\prod_{j=1}^J (r_j+1) - \prod_{j=1}^J (r_j-1)}. \end{aligned}$$

In order to justify the last step, we use the fact that

$$\prod_{j=1}^J (r_j+1) + \prod_{j=1}^J (r_j-1) = 2 \cdot \prod_{x_1,\ldots,x_J : \prod_{j=1}^J x_j = 1} r_j^{(1+x_j)/2},$$

which can be applied to both the numerator and the denominator in order to get the last equation.

If we divide both numerator and denominator by $\prod_{j=1}^J (r_j+1)$, we see that

$$r = \frac{1 + \prod_{j=1}^J \frac{r_j-1}{r_j+1}}{1 - \prod_{j=1}^J \frac{r_j-1}{r_j+1}},$$

which in turn implies

$$\frac{r-1}{r+1} = \prod_{j=1}^J \frac{(r_j-1)}{(r_j+1)}.$$

Recall that l denotes the message from a variable node in the log-likelihood form. Then, $r = e^l$ and we see that

$$\frac{r-1}{r+1} = \tanh(l/2).$$

Combining these two statements, we have

$$\tanh(l/2) = \frac{r-1}{r+1} = \prod_{j=1}^J \frac{r_j-1}{r_j+1} = \prod_{j=1}^J \tanh(l_j/2),$$

so that the final processing rule for the check nodes is

$$l = 2\tanh^{-1}\left(\prod_{j=1}^J \tanh(l_j/2)\right).$$

At the end of each decoding iteration, the overall LLR, i.e. the sum of *all* incoming messages, is calculated for each variable node and its estimated value is changed accordingly. If at any point the estimated vector is found to be a codeword, i.e. its syndrome is the zero vector, decoding can halt declaring a success.

4.5.5 Limitations of Cycle-Free Codes

We have seen a way to efficiently perform bit-wise MAP decoding, assuming that the factor graph (equivalently, the Tanner graph) of the code we use is acyclic. Unfortunately, the following lemma states that this class of codes is not powerful enough.

Lemma 4.4 ([1], Lemma 2.24, p. 64)**.** *Let $\mathcal{C}$ be a binary linear code of rate r that admits a binary Tanner graph that is a forest (which is, quite literally, a set of acyclic graphs, i.e. trees). Then $\mathcal{C}$ contains at least $\left(\frac{2r-1}{2}\right) n$ codewords of weight 2.*

We know that much of a code's error correcting performance depends on its minimum distance, so codewords of weight 2 are definitely not welcome, especially if they can come in large numbers, as the lemma states.

4.5.6 Message Passing on Codes with Cycles

We have seen that the BP algorithm is a very efficient way of computing the marginal distributions of each codeword bit in a cycle-free Tanner graph, in order to perform decoding. We have also seen, however, that cycle-free Tanner graphs lead to poor error correcting performance. The problem that arises should be pretty obvious. In practice, we ignore the fact that cycles exist and perform BP as analysed in the previous sections. In this case, BP is strictly suboptimal since it no longer performs MAP decoding. However, excellent decoding performance can be achieved even with cycles present in the graph. Tools which allow us to determine the performance of this suboptimal decoding scheme, like EXIT charts and Density Evolution, will be analysed in the following chapter.

4.5.7 The Special Case of the Binary Erasure Channel (BEC)

The message-passing rules derived above can be applied to any binary symmetric memoryless channel. We will now derive the simplified message-update rules for a specific channel model, namely the *Binary Erasure Channel* (BEC) where the (binary) input is either received correctly, with probability $(1-\epsilon)$, or completely lost (erased), with probability ϵ. The parameter ϵ is called the *erasure probability* of the channel.

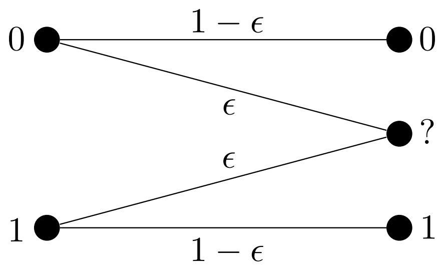


Figure 4.12: Binary Erasure Channel with parameter ϵ.

The decoding problem for this channel is to find the values of the erased bits given the non-erased part of the codeword. If the Tanner graph of the code is a tree, then a natural schedule for the decoding process presents itself: we begin at the leaf nodes and slowly move upwards as the needed messages become available at each level. In general, however, the Tanner graph of a code will not be a tree, so we need to explicitly define a schedule according to which messages will be calculated and passed. This is the most widely used convention: we proceed in *iterations*. We start by processing incoming messages at check nodes and then sending the resulting outgoing messages to variable nodes along all edges. The variable nodes then process these messages, and send the result back to the check nodes across all edges. These two steps constitute one iteration of the algorithm. At round 0, check nodes do not have any information to process, so the initial step consists of the variable nodes sending the messages received from the channel function nodes to their neighbouring check nodes.

So, the initial messages are:

$$\mu_j(0) = p(y_j|0), \qquad \mu_j(1) = p(y_j|1)$$

or, with more convenient tuple notation

$$(\mu_j(0), \mu_j(1)) = (p(y_j|0), p(y_j|1)).$$

In the case of the BEC the initial messages are one of $(1-\epsilon, 0)$, (ϵ, ϵ) and $(0, 1-\epsilon)$. This corresponds to the three possibilities, namely that the received value is 0, erasure or 1 respectively. To make this more clear, consider that the received value is 0. Then, obviously $p(y_j|0) = 1-\epsilon$ since correct transmission occurs with probability $1-\epsilon$. The second value, namely $p(y_j|1)$, will be equal to 0, since the channel does not introduce errors, only erasures. This means that if a value other than an erasure is received, it is guaranteed to be correct. The remaining two tuples can be derived accordingly. Since, as we have seen, we only need to know the ratio of the two outgoing messages and their normalisation plays no role in decoding, we can further simplify our notation by denoting the messages as $(1, 0)$, $(1, 1)$ and $(0, 1)$, which will also be called the "0", the "erasure" and the "1" message respectively.

We will first state the processing rules and then explain them. At a variable node, the outgoing message is an erasure if all incoming messages are erasures. Otherwise, since the channel never introduces errors, all incoming non-erasure messages must agree and be equal either to 0 or to 1. In this case the outgoing message is this common value. At a check node, the outgoing message is an erasure if any of the incoming messages is an erasure. Otherwise, it is the modulo-2 sum of the incoming messages.

Consider the variable node rule: if all messages entering a variable node are from the set $\{(1, 0), (1, 1)\}$, then the outgoing message (which is equal to the component-wise product of the incoming messages according to the general message passing rules) is also from this set. Further, it is equal to $(1, 1)$ (i.e. an erasure) only if all incoming messages are of the form $(1, 1)$ (i.e. erasures). The equivalent statement is true if all incoming messages are from the set $\{(1, 0), (1, 1)\}$ (since the channel never introduces errors, we only need to consider these two cases).

Now, consider the check node rule: it suffices to consider a check node of degree 3 with two incoming messages since check nodes of higher degree can be modelled as the cascade of several check nodes, each of which has two inputs and once output (e.g. $x_1 + x_2 + x_3 = (x_1 + x_2) + x_3$). Let $(\mu_1(0), \mu_1(1))$ and $(\mu_2(0), \mu_2(1))$ denote the incoming messages. By the standard message-passing rules, the outgoing message is

$$\begin{aligned}(\mu(0), \mu(1)) &= \left(\sum_{x_1, x_2} \mathbb{1}_{[x_1+x_2=0]} \mu_1(x_1)\mu_2(x_2), \sum_{x_1, x_2} \mathbb{1}_{[x_1+x_2=1]} \mu_1(x_1)\mu_2(x_2) \right) \\ &= (\mu_1(0)\mu_2(0) + \mu_1(1)\mu_2(1), \mu_1(0)\mu_2(1) + \mu_1(1)\mu_2(0)).\end{aligned}$$

If $(\mu_2(0), \mu_2(1)) = (1, 1)$ then, up to normalization, $(\mu(0), \mu(1)) = (1, 1)$. This shows that if any of the inputs is an erasure, the outgoing message will be an erasure as well. If, on the other hand, both messages are known, then it can be shown by simple calculations that the above rule corresponds to the modulo-2 sum of the incoming messages. Take, for example, the case where $(\mu_1(0), \mu_1(1)) = (0, 1)$ (the "0" message) and $(\mu_2(0), \mu_2(1)) = (1, 0)$ (the "1" message). Then, by the message-passing rule, the outgoing message will be

$$(\mu(0), \mu(1)) = (0 \cdot 1 + 1 \cdot 0, 0 \cdot 0 + 1 \cdot 1) = (0, 1)$$

which indeed corresponds to the modulo-2 sum of the incoming messages, since $1 \oplus 0 = 1$ and $(0, 1)$ corresponds to the "1" message. All other cases can be derived accordingly.

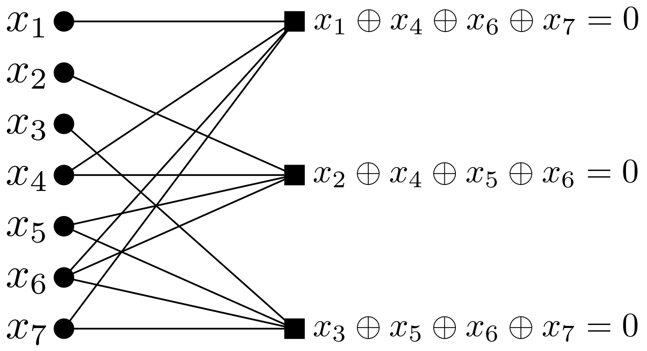
So in the case of the BEC, the message update rules take on a very simple form and the belief propagation algorithm on a whole is also very simple. This, apart from enabling very efficient implementations, also aids us in analysing the asymptotic behaviour of such a decoder, as we will see in the following chapter.

4.6 Concluding Example

For this example we will use the code of example 4.1

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

with the corresponding factor graph presented in example 4.7.


x_1
x_2
x_3
x_4
x_5
x_6
x_7
$x_1 \oplus x_4 \oplus x_6 \oplus x_7 = 0$
$x_2 \oplus x_4 \oplus x_5 \oplus x_6 = 0$
$x_3 \oplus x_5 \oplus x_6 \oplus x_7 = 0$

This code does not have particularly good properties, since many 4-cycles are present, but it suffices for our example.

Consider transmission of the codeword

$$\mathbf{c} = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

over an AWGN channel. Using BPSK modulation, the transmitted signal will be

$$\mathbf{x} = \begin{bmatrix} +1 & +1 & -1 & +1 & -1 & -1 & -1 \end{bmatrix}.$$

The signal is corrupted by noise; let the received vector be

$$\mathbf{y} = \begin{bmatrix} +0.2 & +0.6 & -0.5 & +0.3 & +0.1 & -1.3 & -0.8 \end{bmatrix}.$$

If we perform hard decisions on the received vector, we have

$$\mathbf{y}' = \begin{bmatrix} +1 & +1 & -1 & +1 & +1 & -1 & -1 \end{bmatrix}$$

which corresponds to the binary vector

$$\mathbf{c}' = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{bmatrix}.$$

The syndrome of this vector is

$$\mathbf{s} = \mathbf{c}'\mathbf{H}^T = \begin{bmatrix}0 & 1 & 1\end{bmatrix}$$

indicating that an error has occured. We will now demonstrate the various decoding methods presented in previous chapters.

Syndrome Decoding

We would first need to construct the standard array but, due to its size, it would not be very helpful in illustrating the decoding algorithm. Instead, directly observe that the syndrome of the error pattern

$$\mathbf{e} = \begin{bmatrix}0 & 0 & 0 & 0 & 1 & 0 & 0\end{bmatrix}$$

is

$$\mathbf{e}\mathbf{H}^T = \begin{bmatrix}0 & 1 & 1\end{bmatrix}$$

which is equal to the syndrome of the received vector. An error pattern containing only one 1 would definitely be chosen as a coset leader, since it is one of the n most probable error patterns. So, we decode the received codeword as

$$\hat{\mathbf{c}} = \mathbf{c}' + \mathbf{e} = \begin{bmatrix}0 & 0 & 1 & 0 & 1 & 1 & 1\end{bmatrix},$$

indicating that we have correctly decoded the received codeword, as expected since we know that the $(7, 4)$ Hamming code has a minimum distance of 3, and that a linear code of minimum distance 3 can decode all error patterns of $\lfloor\frac{3-1}{2}\rfloor = 1$ errors under minimum distance decoding.

Bit-Flipping Decoding

Codeword bits 5 and 6 are connected to most unsatisfied check equations, so their values will be flipped to 1 and 0 respectively. Check equations 2 and 3 remain unsatisfied and now check equation 1 is also unsatisfied. At this point, codeword 6 is connected to most unsatisfied check equations (all of them, in fact), so its value will be flipped back to 1, satisfying all check equations. The final codeword is:

$$\hat{\mathbf{c}} = \begin{bmatrix}0 & 0 & 1 & 0 & 1 & 1 & 1\end{bmatrix}$$

which is exactly the codeword we transmitted.

Weighted Bit-Flipping Decoding

We first need to calculate $|\mathbf{y}_{\min}|$ for each parity check

- Check 1: $|\mathbf{y}_{\min}|_1 = 0.2$
- Check 2: $|\mathbf{y}_{\min}|_2 = 0.1$
- Check 3: $|\mathbf{y}_{\min}|_3 = 0.3$.

We now need to calculate E_i for each codeword bit

- $E_1 = (1 - 2 \cdot 0)|\mathbf{y}_{\min}|_1 = +0.2$
- $E_2 = (1 - 2 \cdot 1)|\mathbf{y}_{\min}|_2 = -0.1$
- $E_3 = (1 - 2 \cdot 1)|\mathbf{y}_{\min}|_3 = -0.3$
- $E_4 = (1 - 2 \cdot 0)|\mathbf{y}_{\min}|_1 + (1 - 2 \cdot 1)|\mathbf{y}_{\min}|_2 = +0.1$

- $E_5 = (1 - 2 \cdot 1)|\mathbf{y}_{\min}|_2 + (1 - 2 \cdot 1)|\mathbf{y}_{\min}|_3 = -0.4$
- $E_6 = (1 - 2 \cdot 0)|\mathbf{y}_{\min}|_1 + (1 - 2 \cdot 1)|\mathbf{y}_{\min}|_2 + (1 - 2 \cdot 1)|\mathbf{y}_{\min}|_3 = -0.2$
- $E_7 = (1 - 2 \cdot 0)|\mathbf{y}_{\min}|_1 + (1 - 2 \cdot 1)|\mathbf{y}_{\min}|_3 = -0.1$.

We see that $\min_i E_i = E_5$, so we will flip codeword bit 5. Now all equations are satisfied and the final codeword is

$$\hat{\mathbf{c}} = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

indicating that we have correctly decoded the received codeword. In the case of weighted BF decoding, it was instantly clear which codeword bit was "most wrong", so the decoding was completed in only one step.

Majority-Logic Decoding

For codeword bit 1 we have that the only check it is connected to is equal to 0. So, we set $\hat{\mathbf{c}}_1 = \mathbf{c}'_1 \oplus 0 = 0$ and recalculate the check equations. Check equation 1 is now satisfied. Moving on to codeword bit 2 we see that the only check it is connected to is equal to 1. So, we set $\hat{\mathbf{c}}_2 = \mathbf{c}'_2 \oplus 1 = 1$ and recalculate the check equations. Check equation 2 is now also satisfied. Moving on to codeword bit 3 we see that the only check it is connected to is equal to 1. So, we set $\hat{\mathbf{c}} = \mathbf{c}'_3 \oplus 1 = 0$ and recalculate the check equations. Check equation 3 is now also satisfied. Since all check equations are now satisfied, none of the remaining codeword bits will be flipped. So, the decoded codeword using MLD is:

$$\hat{\mathbf{c}} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

which is obviously not the codeword we transmitted, so an erroneous decoding has occurred.

Weighted Majority-Logic Decoding

Recall that:

- Check 1: $|\mathbf{y}_{\min}|_1 = 0.2$
- Check 2: $|\mathbf{y}_{\min}|_2 = 0.1$
- Check 3: $|\mathbf{y}_{\min}|_3 = 0.3$.

We now proceed with MLG decoding as follows. For codeword bit 1 we have:

$$E_1 = (1 - 2 \cdot 0)|\mathbf{y}_{\min}|_1 = +0.2$$

meaning that its value will be chosen to be equal to 0. Parity check 1 is now satisfied. For codeword bit 2 we have:

$$E_2 = (1 - 2 \cdot 1)|\mathbf{y}_{\min}|_2 = -0.1$$

meaning that its value will be flipped to 1. Parity check 2 is now satisfied. For codeword bit 3 we have:

$$E_3 = (1 - 2 \cdot 1)|\mathbf{y}_{\min}|_3 = -0.3$$

meaning that its value will be flipped to 0. Parity check 3 is now satisfied. Since all parity checks are now satisfied, none of the remaining codeword bits will be flipped. So, the decoded codeword using MLD is:

$$\hat{\mathbf{c}} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

which is, again, obviously not the codeword we transmitted, so an erroneous decoding has occurred. Note that in both cases the decoding procedure starts with the 3 degree-1 variable nodes. If we use a different graph where the first variable nodes are of higher degree, the result will be different (probably better).

Belief Propagation

The LLR for an AWGN channel can be calculated as follows

$$\begin{aligned}\mathrm{LLR}(y) &= \ln \frac{f(y|x=-1)}{f(y|x=+1)} \\ &= \ln \frac{\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(y+1)^2}{2\sigma^2}\right\}}{\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(y-1)^2}{2\sigma^2}\right\}} \\ &= \ln \exp\left\{\frac{(y-1)^2}{2\sigma^2} - \frac{(y+1)^2}{2\sigma^2}\right\} \\ &= -\frac{2y}{\sigma^2}.\end{aligned}$$

Initialisation

Assuming a noise variance of, say, $\sigma^2 = 0.5$ (in practice, the noise variance is a known parameter), we get

$$\mathrm{LLR}(y) = -\frac{2y}{0.5} = -4y.$$

Now, we calculate the LLR for each variable node

- LLR(y_1) = -0.8
- LLR(y_2) = -2.4
- LLR(y_3) = +2.0
- LLR(y_4) = -1.2
- LLR(y_5) = -0.4
- LLR(y_6) = +5.2
- LLR(y_7) = +3.2.

These values are sent to the check nodes.

Iteration $\ell = 1$

At check nodes, the update rule reads

$$l = 2\tanh^{-1}\left(\prod_{j=1}^{J} \tanh(l_j/2)\right).$$

So, for the message headed from check node 2 to variable node 5,[4] we have

$$\begin{aligned} l_{2\to 5} &= 2\tanh^{-1}\left(\prod_{j=1}^{J}\tanh(l_j/2)\right) \\ &= 2\tanh^{-1}\left(\tanh(l_2/2)\tanh(l_4/2)\tanh(l_6/2)\right) \\ &= 2\tanh^{-1}\left(\tanh(-2.4/2)\tanh(-1.2/2)\tanh(+5.2/2)\right) \\ &= 2\tanh^{-1}(0.44) \\ &= 0.94. \end{aligned}$$

Accordingly, for the message headed from check node 3 to variable node 5, we have

$$\begin{aligned} l_{3\to 5} &= 2\tanh^{-1}\left(\prod_{j=1}^{J}\tanh(l_j/2)\right) \\ &= 2\tanh^{-1}\left(\tanh(l_3/2)\tanh(l_6/2)\tanh(l_7/2)\right) \\ &= 2\tanh^{-1}\left(\tanh(+2.0/2)\tanh(+5.2/2)\tanh(+3.2/2)\right) \\ &= 2\tanh^{-1}(0.69) \\ &= 1.71. \end{aligned}$$

Using these messages, variable node 5 can update its overall LLR by summing the incoming messages:

$$\text{LLR} = -0.40 + 0.94 + 1.71 = 2.25$$

making the decision for the value of variable node equal to 1. All other messages are calculated accordingly but are omitted here to avoid a flood of calculations. Since variable node 5 now has the correct value and all other nodes do not change their value[5], decoding halts declaring a success after only one iteration.

[4]We focus on variable node 5 for the sake of this example, since we know that it is the only one with a wrong value.
[5]This can be easily verified by calculating all messages $l_{i\to j}$ for $i = 1, 2, 3$ and $j = 1, \ldots, 7$.

Chapter 5

Design of LDPC Codes

In this chapter we introduce some tools which can help us analyse the behaviour of any code ensemble under belief propagation decoding. We will make a relatively analytical derivation of the rules for Density Evolution, explain the Gaussian Approximation and briefly discuss EXIT charts at the end of the chapter. First, however, we have to justify why we can focus on ensembles rather than individual codes. The following theorems assert that the individual elements of an ensemble behave, with high probability, close to the ensemble average for the BEC channel and the family of BMS channels respectively.

Theorem 5.1 ([1], Theorem 3.30, p. 85)**.** *Let G, chosen uniformly at random from LDPC(λ, ρ), be used for transmission over the BEC(ϵ) channel. Assume that the decoder performs ℓ rounds of message-passing decoding and let $P_b^{MP}(G, \epsilon, \ell)$ denote the resulting bit error probability. Then, for ℓ fixed and for any given $\delta > 0$, there exists an $\alpha > 0$, $\alpha = \alpha(\lambda, \rho, \epsilon, \delta, \ell)$, such that:*

$$\mathbb{P}\{|P_b^{MP}(G, \epsilon, \ell) - \mathbb{E}_{G' \in LDPC(n,\lambda,\rho)}\left[P_b^{MP}(G', \epsilon, \ell)\right]| > \delta\} \leq e^{-\alpha n}.$$

Theorem 5.2 ([1], Theorem 4.92, p. 216)**.** *Let G, chosen uniformly at random from LDPC(λ, ρ), be used for transmission over a BMS channel characterised by its L-density $\mathcal{L}_{BMSC}$. Assume that the decoder performs ℓ rounds of message-passing decoding and let $P_b^{MP}(G, \mathcal{L}_{BMSC,\ell})$ denote the resulting bit error probability. Then, for any given $\delta > 0$, there exists an $\alpha > 0$, $\alpha = \alpha(\lambda, \rho, \delta)$, such that:*

$$\mathbb{P}\{|P_b^{MP}(G, \mathcal{L}_{BMSC}, \ell) - \mathbb{E}_{LDPC(n,\lambda,\rho)}\left[P_b^{MP}(G, \mathcal{L}_{BMSC}, \ell)\right]| > \delta\} \leq e^{-\alpha n}.$$

In both cases, this probability tends to 0 as the blocklength tends to infinity, since α is strictly positive. One of the most common methods for asymptotic behaviour analysis is the Density Evolution algorithm, which can predict the decoding performance of a code ensemble in the limit of infinite blocklength and under an infinite number of iterations, using only the degree distribution of the code. We start with the case of the BEC channel, due to its simplicity, and then extend the results to the family of BMS channels. First, however, we have to introduce some new concepts and explain a basic simplification we use.

5.1 BEC Channel

5.1.1 Restriction to the All-Zero Codeword

We start with the most basic simplification which applies to both the BEC and BMS channel families. The basic idea is that the error probability of a decoder for a particular code and channel does not depend on the transmitted codeword. So, we can restrict ourselves to any codeword, and the most natural choice is the all-zero codeword which is contained in every linear code. A more formal expression of the above idea follows:

Theorem 5.3 ([1], Fact 3.29, p. 85)**.** *Let G be the Tanner graph representing a binary linear code $\mathcal{C}$. Assume that $\mathcal{C}$ is used to transmit over the BEC(ϵ) and assume that the decoder performs message-passing decoding on G. Let $P^{BP}(G, \epsilon, \ell, x)$ denote the conditional (bit or block) probability of erasure after the ℓ-th iteration, assuming x was sent, $x \in \mathcal{C}$. Then, $P^{BP}(G, \epsilon, \ell, x) = \frac{1}{|\mathcal{C}|}\sum_{x' \in \mathcal{C}} P^{BP}(G, \epsilon, \ell, x') = P^{BP}(G, \epsilon, \ell)$, i.e. $P^{BP}(G, \epsilon, \ell, x)$ is independent of the transmitted codeword.*

An equivalent statement is true for BMS channels ([1], Lemma 4.90, p. 215), under some symmetry conditions, but we will not state it here since some concepts needed for the formulation of the statement have not been introduced yet. What is important is to bear in mind that this simplification is indeed applicable.

5.1.2 Computation Graph

Let us focus on a single variable node of the Tanner graph of a code, say x_1. The decoding process for this variable node is initialised at round 0, where the node sends the channel log-likelihood ratio to its connected check nodes. These nodes, in turn, after processing all their received messages, send a message back to x_1. These messages depend on all other incoming messages at the check nodes, except for the one which came from x_1. If we unroll these dependencies for a fixed number of iterations, say ℓ, we will create the *computation graph* for x_1 of depth ℓ. Since this graph is rooted on a variable node, we call it the computation graph from a node perspective. Equivalently, we can create the computation graph from an edge perspective by rooting the graph at an edge, say e, and unrolling the dependencies of the messages passed along that edge.

Definition 5.4. Consider the ensemble LDPC(n, λ, ρ). The associated ensemble of computation graphs of height ℓ from a node perspective, denoted $\dot{\mathcal{C}}_\ell(n, \lambda, \rho)$, is defined as follows. To sample from this ensemble, pick a graph G from LDPC(n, λ, ρ) uniformly at random and draw the computation graph of height ℓ of a randomly chosen variable node of G. Each such computation graph, call it T, is an unlabelled rooted graph in which each distinct node is drawn exactly once. The ensemble $\dot{\mathcal{C}}_\ell(n, \lambda, \rho)$ consists of the set of such computation graphs together with the probabilities $\mathbb{P}\{T \in \dot{\mathcal{C}}_\ell(n, \lambda, \rho)\}$, which are induced by the preceding sampling procedure. In the same way, to sample from the ensemble of computation graphs from an edge perspective, denote it by $\vec{\mathcal{C}}_\ell(n, \lambda, \rho)$, pick randomly an edge e, and draw the computation graph of e of height ℓ in G. Since $\dot{\mathcal{C}}_\ell(n, \lambda, \rho)$ and $\vec{\mathcal{C}}_\ell(n, \lambda, \rho)\}$ share many properties it is convenient to be able to refer to both of them together. In this case, we write $\mathcal{C}_\ell(n, \lambda, \rho)$

Note that these graphs are unlabelled, meaning that in reality some variable (and check) nodes will be present multiple times in each graph, so the graph can not be generally considered to be a tree. The operational meaning of the ensembles $\dot{\mathcal{C}}_\ell(n, \lambda, \rho)$ and $\vec{\mathcal{C}}_\ell(n, \lambda, \rho)$ is clear: $\dot{\mathcal{C}}_\ell(n, \lambda, \rho)$ represents the ensemble of computation graphs that the BP decoder encounters when making a decision on a randomly chosen bit from a random sample of LDPC(n, λ, ρ), assuming the decoder performs ℓ iterations and $\vec{\mathcal{C}}_\ell(n, \lambda, \rho)$ represents the ensemble of computation graphs which the BP decoder encounters when determining the variable-to-check messages sent out along a randomly chosen edge in the ℓ-th iteration.

Since we are interested in analysing the decoding performance of ensembles of LDPC codes, the question arises whether we can use these computation graphs to make a prediction of the erasure probability in a graph, after ℓ decoding iterations have been performed. If we let $P_b^{\mathrm{BP}}(T, \epsilon)$ denote the conditional erasure probability of the BP decoder on a graph $T \in \dot{\mathcal{C}}_\ell(n, \lambda, \rho)$ for transmission over a BEC with erasure probability ϵ, then we can calculate the unconditional probability of erasure for the ensemble LDPC(n, λ, ρ) as follows

$$\mathbb{E}_{\mathrm{LDPC}(n,\lambda,\rho)}\left[P_b^{\mathrm{BP}}(G, \epsilon, \ell)\right] = \sum_T \mathbb{P}\{T \in \dot{\mathcal{C}}_\ell(n, \lambda, \rho)\} P_b^{\mathrm{BP}}(T, \epsilon).$$

The probabilities $\mathbb{P}\{T \in \dot{\mathcal{C}}_\ell(n, \lambda, \rho)\}$ are calculated at the construction of the ensemble by simple enumeration. For each computation graph $T \in \dot{\mathcal{C}}_\ell(n, \lambda, \rho)$, the conditional probability of erasure $P_b^{\text{BP}}(T, \epsilon)$ can also be calculated from is structure. This approach, however, very quickly becomes computationally infeasible. Consequently, we will limit ourselves to the analysis in the limit of infinite blocklength, where some simplifications can be made.

5.1.3 Tree Ensemble

Intuitively, as the blocklength increases the number of cycles in the Tanner graph decreases. In fact, in the limit of infinite blocklengths, the computation graph becomes a tree with probability 1 and each subtree of the computation graph tends to an independent sample whose distribution is determined only by the degree distribution pair (λ, ρ).

Definition 5.5. The tree ensembles $\dot{\mathcal{T}}_\ell(\lambda, \rho)$ and $\vec{\mathcal{T}}_\ell(\lambda, \rho)$ are the asymptotic versions of the computation graph ensembles $\dot{\mathcal{C}}_\ell(n, \lambda, \rho)\}$ and $\vec{\mathcal{C}}_\ell(n, \lambda, \rho)\}$. We start by describing $\vec{\mathcal{T}}_\ell(\lambda, \rho)$. Each element of $\vec{\mathcal{T}}_\ell(\lambda, \rho)$ is a bipartite tree rooted in a variable node. The ensemble $\vec{\mathcal{T}}_0(\lambda, \rho)$ contains a single element, the trivial tree consisting only of the root variable node. Let $L(i)$ denote a bipartite tree rooted in a variable node which has i (check-node) children and, in the same manner, let $R(i)$ denote a bipartite tree rooted in a check node which has i (variable-node) children. To sample from $\vec{\mathcal{T}}_\ell(\lambda, \rho), \ell \geq 1$, first sample an element from $\vec{\mathcal{T}}_{\ell-1}(\lambda, \rho)$. Next, substitute each of its leaf variable nodes with a random element from $\{L(i)\}_{i \geq 1}$, where $L(i)$ is chosen with probability λ_{i+1}. Finally, substitute each of its leaf check nodes with a random element from $\{R(i)\}_{i \geq 1}$, where $R(i)$ is chosen with probability ρ_{i+1}. The ensemble $\dot{\mathcal{T}}_\ell(\lambda, \rho)$ is defined in almost the same way, the only difference being that random elements from $\{L(i)\}_{i \geq 1}$ are chosen with probability L_{i+1} instead of λ_{i+1}. When we are referring to both ensembles, we will use the notation $\mathcal{T}_\ell(\lambda, \rho)$, as we did with computation graph ensembles.

The above recursive definition is the key to the analysis of BP decoding. We will also need to introduce the notion of the *channel code*, in order to discuss the *tree channel* in the next section.

Definition 5.6. Let $T \in \mathcal{T}_\ell(\lambda, \rho)$. Define the tree channel, $C(T)$, to be the set of valid codewords on T. More precisely, $C(T)$ is the set of 0/1 assignments of the variable nodes contained in T that fulfil the constraints on the tree. Further, let $C^0(T)$ and $C^1(T)$ denote the valid codewords on T such that the root variable node is 0 and 1 respectively. Clearly, $C^0(T)$ and $C^1(T)$ are disjoint and their union equals $C(T)$.

5.1.4 Tree Channel

We will now define the tree channel and then state a theorem that asserts that in the limit of infinite blocklengths the average performance of an ensemble LDPC(λ, ρ) converges to the performance of the corresponding tree channel. This theorem is the key to the asymptotic analysis of an ensemble using the density evolution algorithm.

Definition 5.7. Given the BEC characterised by its erasure probability ϵ and a tree ensemble $\mathcal{T}_\ell = \mathcal{T}_\ell(\rho, \lambda)$, we define the associated $(\mathcal{T}_\ell, \epsilon)$-tree channel. The channel takes binary input $X \in \{0, 1\}$ with uniform probability. The output of the channel is constructed as follows. Given X, first pick T from $\mathcal{T}_\ell$ uniformly at random. Next, pick a codeword from $C^0(T)$ uniformly at random if $X = 0$ and otherwise pick a codeword from $C^1(T)$ uniformly at random. As a shorthand, let us say that we pick a codeword $C^X(T)$ uniformly at random. Transmit this codeword over the BEC(ϵ). Call the output Y. The receiver sees (T, Y) and estimates X. Let $P_b^{\text{BP}}(\epsilon)$ denote the resulting bit error probability, assuming that (T, Y) is processed by a BP decoder.

Theorem 5.8 ([1], Theorem 3.49, p. 94)**.** *For a given degree distribution pair (λ, ρ) consider the sequence of associated ensembles LDPC(n, λ, ρ) for increasing blocklength n under ℓ rounds of BP decoding. Then*

$$\lim_{n\to\infty} \mathbb{E}_{LDPC(n,\lambda,\rho)}\left[P_b^{BP}(G,\epsilon,\ell)\right] = P^{BP}_{\mathcal{T}(\lambda,\rho)}(\epsilon),$$

meaning that the average performance converges to that of the corresponding tree channel.

5.1.5 Density Evolution

We will now derive an iterative expression for the calculation of the performance of a tree channel. By definition, the initial variable-to-check message is equal to the received message, which is an erasure message with probability ϵ. It follows that $P^{\mathrm{BP}}_{\vec{\mathcal{T}}_0} = \epsilon$. We will now use induction to find an expression for $x_\ell = P^{\mathrm{BP}}_{\vec{\mathcal{T}}_\ell}$ as a function of the erasure probability at the previous iteration, namely $x_{\ell-1} = P^{\mathrm{BP}}_{\vec{\mathcal{T}}_{\ell-1}}$. We start with the check-to-variable messages in the $(\ell+1)$-th iteration. Recall that by definition of the algorithm a check-to-variable message emitted by a check node of degree i along a particular edge is the erasure message if any of the $i-1$ incoming messages is an erasure. By assumption, each such message is an erasure with probability x_ℓ and all messages are independent, so that the probability that the outgoing message is an erasure is equal to $1-(1-x_\ell)^{i-1}$. Since the edge has probability ρ_i to be connected to a check node of degree i it follows that the expected erasure probability of a check-to-variable message in the $(\ell+1)$-th iteration is equal to:

$$\sum_i \rho_i(1-(1-x_\ell)^{i-1}) = 1-\rho(1-x_\ell).$$

Now consider the erasure probability of the variable-to-check message in the $(\ell+1)$-th iteration. Consider an edge e that is connected to a variable node of degree i. The outgoing variable-to-check message along this edge in the $(\ell+1)$-th iteration is an erasure if the received value of the associated variable node is an erasure and all $i-1$ incoming messages are erasures. This happens with probability $\epsilon(1-\rho(1-x_\ell))^{i-1}$. Averaging again over the variable node degree distribution λ, we get that

$$\begin{aligned} x_{\ell+1} = P^{\mathrm{BP}}_{\vec{\mathcal{T}}_{\ell+1}} &= \sum_i \lambda_i \epsilon(1-\rho(1-x_\ell))^{i-1} \\ &= \epsilon\sum_i \lambda_i(1-\rho(1-x_\ell))^{i-1} \\ &= \epsilon\lambda(1-\rho(1-x_\ell)). \end{aligned}$$

So, for a given variable and check node degree distribution pair (λ, ρ) and by initiliazing with $x_0 = \epsilon$, we can calculate the average bit erasure probability for the corresponding tree channel for any number of iterations ℓ with the iterative procedure we just derived

$$x_{\ell+1} = \epsilon\lambda(1-\rho(1-x_\ell)), \qquad x_0 = \epsilon.$$

It is important to note at this point that for this analysis we have made the assumption that the computation graph is a tree, i.e. a cycle-free graph. We have seen that this property does indeed hold in the limit of infinite blocklengths, but for finite blocklengths, and especially for small blocklengths, cycles can not be avoided. This is the main reason why the behaviour of shorter codes can not be accurately predicted by density evolution. In practice, however, density evolution is used to find an optimised degree distribution and then techniques, such as Progressive Edge Growth, are applied which aim to minimise the number and girth of cycles in the graph in order to achieve a performance as close as possible to the predicted. Another approach, which is not so practical but is also widely used, is to use codes of very long blocklength in order to emulate the asymptotic behaviour.

Below we illustrate density evolution for the $(3, 6)$ LDPC ensemble over the BEC channel. The threshold for this code ensemble has been found to be equal to $\epsilon^* = 0.42943$.

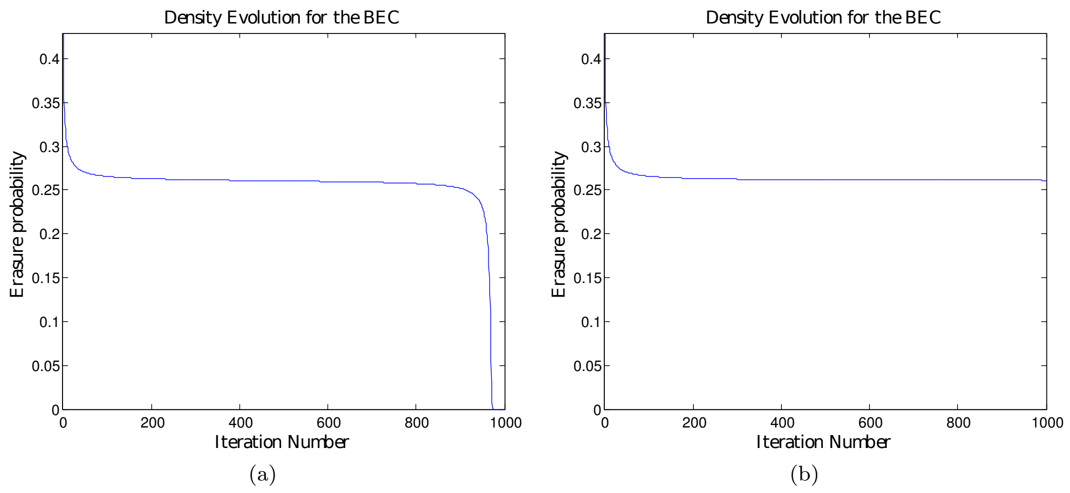


Figure 5.1: Density evolution for the BEC channel. In (a) we have $\epsilon = 0.42943$. In (b) we have $\epsilon = 0.42944$.

In figure 5.1(a), where the erasure probability is exactly equal to the threshold ($\epsilon = 0.42943$), we see that the average erasure probability of the exchanged messages converges to zero after a certain number of iterations. In figure 5.1(b) on the other hand, where the erasure probability is over the threshold ($\epsilon = 0.42944$), we see that the average erasure probability converges to a fixed, non-zero, value.

We now proceed to the introduction of some useful properties of density evolution, namely the monotocity (with respect to the channel and the iteration number), the concept of the *threshold* and, finally, the *stability condition*.

Monotonicity

Monotocity, either with respect to the channel parameter or with respect to the number of iterations ℓ, plays a fundamental role in the analysis of density evolution. Let $f(\epsilon, x) = \epsilon\lambda(1 - \rho(1 - x_\ell))$. The following lemma asserts the monotonicity of $f(\epsilon, x)$ with respect to the channel parameter.

Lemma 5.9 ([1], Lemma 3.54, p. 96)**.** *Let* (λ, ρ) *be a degree distribution pair and* $\epsilon \in [0, 1]$. *If* $P^{BP}_{\mathcal{T}_\ell}(\epsilon) \overset{\ell\to\infty}{\to} 0$, *then* $P^{BP}_{\mathcal{T}_\ell}(\epsilon') \overset{\ell\to\infty}{\to} 0$ *for all* $0 \leq \epsilon' \leq \epsilon$.

The following lemma is also very interesting since it actually states that, by increasing the number of iterations we allow our decoder to make, performance is not degraded.

Lemma 5.10 ([1], Lemma 3.55, p. 97)**.** *Let* $\epsilon, x_0 \in [0, 1]$. *For* $\ell = 1, 2, \ldots,$ *define* $x_\ell(x_0) = f(\epsilon, x_{\ell-1}(x_0))$. *Then* $x_\ell(x_0)$ *is a monotone sequence converging to the nearest (in the direction of monotonicity) solution of* $x = f(\epsilon, x)$.

Threshold

From the density evolution equations, we see that for every non-negative integer ℓ:

$$P^{\text{BP}}_{\vec{\mathcal{T}}_\ell}(\epsilon = 0) = 0, \qquad \text{but} \qquad P^{\text{BP}}_{\vec{\mathcal{T}}_\ell}(\epsilon = 1) = 1$$

and, in particular, these equalities are satisfied if $\ell \to \infty$. Combined with the preceding monotonicity property, this shows the existence of a well-defined supremum of ϵ for which $P^{\text{BP}}_{\vec{\mathcal{T}}_\ell} \overset{\ell\to\infty}{\to} 0$. This supremum is called the threshold. It can be shown that $P^{\text{BP}}_{\vec{\mathcal{T}}_\ell} \overset{\ell\to\infty}{\to} 0$ implies $P^{\text{BP}}_{\dot{\mathcal{T}}_\ell} \overset{\ell\to\infty}{\to} 0$, so we can generally write $P^{\text{BP}}_{\mathcal{T}_\ell} \overset{\ell\to\infty}{\to} 0$.

Definition 5.11. The threshold associated with the degree distribution pair (λ, ρ), call it $\epsilon^{\text{BP}}(\lambda, \rho)$, is defined as:

$$\epsilon^{\text{BP}}(\lambda, \rho) = \sup\{\epsilon \in [0, 1] : P^{\text{BP}}_{\mathcal{T}_\ell(\lambda,\rho)}(\epsilon) \overset{\ell\to\infty}{\to} 0\}.$$

This means that for $\epsilon > \epsilon^{\text{BP}}(\lambda, \rho)$ we can not hope to achieve reliable transmission over a BEC channel with erasure probability ϵ with a code from the LDPC(n, λ, ρ) ensemble. This fact, along with lemma 5.10, means that if $\epsilon > \epsilon^{\text{BP}}(\lambda, \rho)$, the bit erasure probability will converge to the nearest non-zero fixed point of $x = f(\epsilon, x)$. The threshold is usually determined by numerical experiments for any given code ensemble.

Stability Condition

If we expand the right-hand side of the density evolution equation into a Taylor series around 0, we get

$$x_\ell = \epsilon\lambda'(0)\rho'(1)x_{\ell-1} + \mathcal{O}(x^2_{\ell-1}).$$

If x_ℓ is sufficiently small, the convergence behaviour is determined by the term linear in x_ℓ. More precisely, the convergence depends on whether $\epsilon\lambda'(0)\rho'(1)$ is smaller or larger than 1.

Theorem 5.12 ([1], Theorem 3.65, p. 100)**.** *Assume that we are given a degree distribution pair (λ, ρ) and $\epsilon, x_0 \in [0, 1]$. Let $x_\ell(x_0)$ be defined as in lemma 5.10. Then*

1. *(Necessity) If $\epsilon\lambda'(0)\rho'(1) > 1$, then there exists a strictly positive constant $\xi = \xi(\lambda, \rho, \epsilon)$ such that $\lim_{\ell\to\infty} x_\ell(x_0) \geq \xi$ for all $x_0 \in (0, 1)$.*

2. *(Sufficiency) If $\epsilon\lambda'(0)\rho'(1) < 1$, then there exists a strictly positive constant $\xi = \xi(\lambda, \rho, \epsilon)$ such that $\lim_{\ell\to\infty} x_\ell(x_0) = 0$ for all $x_0 \in (0, \xi)$.*

So, using the stability condition, we can check if a threshold exists for a given degree distribution pair and channel parameter ϵ. In fact, the following upper bound on the threshold is implied

$$\epsilon^{\text{BP}}(\lambda, \rho) \leq \frac{1}{\lambda'(0)\rho'(1)}.$$

This concludes our quick analysis of density evolution for the BEC. We will now proceed to the more general family of Binary Memoryless Symmetric Channels (BMSC), of which the BEC is a member as well. Density evolution for these channels is a bit more complex, as we will see, due to the more complex nature of the update rules of Belief Propagation.

5.2 BMS Channels

The family of Binary Memoryless Symmetric channels includes a wide variety of channel models. Some examples are the Binary Symmetric channel (BSC) and the Binary-Input AWGN channel (BI-AWGN). In the case of the BSC, the binary input is either received correctly with probability $1-p$, or flipped with probability p. We say that this channel is a BSC with cross-over probability p, denoted BSC(p). With BMS channels, we commonly use the binary alphabet $\{+1, -1\}$ instead of $\{0, 1\}$, with $+1$ corresponding to 0 and -1 corresponding to 1.

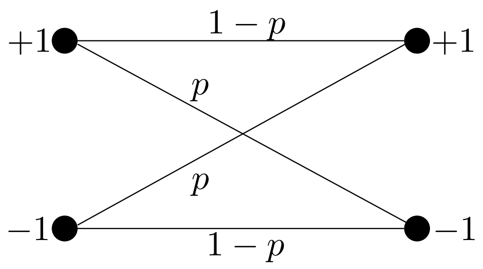


Figure 5.2: The Binary Symmetric Channel.

The BI-AWGN channel has binary input which is corrupted by additive white Gaussian noise, meaning that the output is a real number. The only parameters needed for the characterisation of this channel are the noise mean and variance. Hence, this family of channels is usually denoted BI-AWGN(μ, σ) or equivalently, since if we know the mean of the noise we can always subtract it and get a zero-mean noise, BI-AWGN(σ).

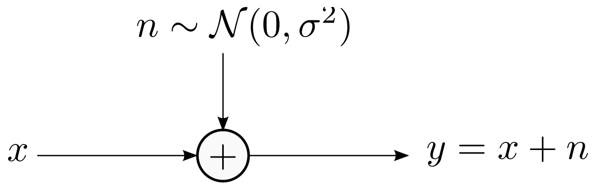


Figure 5.3: The BI-AWGN Channel.

If we denote the input as $x \in \{\pm 1\}$, the additive noise as $n \sim \mathcal{N}(0, \sigma^2)$, and the output as $y \in \mathbb{R}$, then

$$y = x + n.$$

Density evolution for this channel family is a bit more complex since the involved densities do not take on a small number of values as with the BEC channel, but are continuous functions.

5.2.1 Densities

Recall that the messages exchanged by the BP decoder are in a log-likelihood ratio form. Also recall that we have mentioned that we can restrict ourselves to the all-zero codeword for analysis, which in this case becomes the all-one codeword, according to the mapping we introduced in the previous section. Then, the distribution of the log-likelihood ratio $l(Y)$associated with a channel observation Y assuming that $X = 1$, is denoted $\mathcal{L}$ and is called an L-density. The bit error probability associated with an L-density $\mathcal{L}$ is

$$e(\mathcal{L}) = \frac{1}{2} \int_{\mathbb{R}} \mathcal{L}(x) e^{-(|x/2|+x/2)} dx.$$

Some L-densities of commonly used channel models are:

$$\mathcal{L}_{\text{BEC}(\epsilon)}(y) = \epsilon\Delta_0(y) + (1-\epsilon)\Delta_{+\infty}(y)$$
$$\mathcal{L}_{\text{BSC(p)}}(y) = p\Delta_{-\ln\frac{1-p}{p}}(y) + (1-p)\Delta_{\ln\frac{1-p}{p}}(y)$$
$$\mathcal{L}_{\text{BI-AWGN}(\sigma)}(y) = \sqrt{\frac{\sigma^2}{8\pi}}\exp\left\{-\frac{\left(y-\frac{2}{\sigma^2}\right)^2\sigma^2}{8}\right\}$$

where $\Delta_z(x) = \Delta_0(x-z)$ and $\Delta_0(x)$ is the Dirac delta centered at zero.

A function closely related to $l(y)$ is

$$d(y) \triangleq \tanh(l(y)/2) = \frac{1-e^{-l(y)}}{1+e^{-l(y)}} = p_{X|Y}(1|y) - p_{X|Y}(-1|y).$$

We see that from $l(y)$ we can compute $d(y)$, and vice versa. When we conceive of $d(y)$ as a random variable, we write $D = d(y)$. The distribution of D, conditioned on $X = 1$, is called a D-density and is denoted $\mathcal{D}$.

Another important quantity is

$$g(y) \triangleq (\text{sign}(y), \ln\coth(|l(y)|/2)) = (\text{sign}(d(y)), -\ln|d(y)|).$$

As above, when we refer to the corresponding random variable, we write $G = g(y)$. The density of G, conditioned on $X = 1$, is called a G-density and is denoted $\mathcal{G}$. Note that $g(y)$ takes values in $\{\pm 1\} \times [0, +\infty]$. A G-density $a(s,x)$ therefore has the form

$$a(s,x) = \mathbb{1}_{[s=1]}a(1,x) + \mathbb{1}_{[s=-1]}a(-1,x).$$

We will see that the convolution of A-densities represents the message distribution change at variable nodes, and the convolution of G-densities represents the message distribution change at variable nodes. A-densities are real valued, $\mathbb{R} \to \mathbb{R}$ functions, so their convolution is well-defined. G-densities, however, are defined over the product of $\mathbb{R}^+$ and $\{\pm 1\}$ so calculating their convolution is not trivial. The space $\mathbb{R}^+$ on its own has a well-defined, one-sided convolution. Now, instead of $\{\pm 1\}$, think of the set $\{0,1\}$ with modulo-2 addition. The associated convolution is the cyclic convolution of sequences of length two. Therefore, the convolution of G-densities is just the two-dimensional convolution which consists of the familiar convolution over $\mathbb{R}^+$ in one dimension and the convolution over $\{0,1\}$ in the other dimension. In other words, the new convolution is a convolution over the group $\{0,1\} \times [0, +\infty]$. Explicitly, the convolution of

$$a_1(s,x) = \mathbb{1}_{[s=1]}a_1(1,x) + \mathbb{1}_{[s=-1]}a_1(-1,x)$$

and

$$a_2(s,x) = \mathbb{1}_{[s=1]}a_2(1,x) + \mathbb{1}_{[s=-1]}a_2(-1,x)$$

is

$$a(s,x) = \mathbb{1}_{[s=1]}(a_1(1,\cdot) * a_2(1,\cdot) + a_1(-1,\cdot) * a_2(-1,\cdot)) + \mathbb{1}_{[s=-1]}(a_1(1,\cdot) * a_2(-1,\cdot) + a_1(-1,\cdot) * a_2(1,\cdot))$$

where $*$ denotes the standard (one-sided) convolution. We denote this convolution as $\circledast$, so that for the above example we have

$$a(s,x) = a_1(s,x) \circledast a_2(s,x).$$

For notation purposes, when we write $a \circledast b$, we will mean the convolution of G-densities regardless of the representation which a and b currently have. So, if a and b are, say L-densities, then by writing $a \circledast b$, we refer to the transformation of a, b into the G-densities, their convolution in the G-domain and their transformation back to the L-domain.

5.2.2 Density Evolution

Before deriving the expressions for density evolution, we first need to extend some definitions we already made for the BEC channel to the BMS channel family, namely the tree channel definition and the theorem asserting the convergence to the tree channel.

Definition 5.13. Given a BMS channel characterised by its L-density $\mathcal{L}_{\text{BMSC}}$ and a tree ensemble $\mathcal{T}_\ell = \mathcal{T}_\ell(\lambda, \rho)$, we define the associated $(\mathcal{T}_\ell, \mathcal{L}_{\text{BMSC}})$-tree channel. The channel takes binary input $X \in \{\pm 1\}$with uniform probability. The output of the channel is constructed as follows. Given X, first pick a codeword from $C^X(T)$ uniformly at random. Transmit this codeword over the BMS channel defined by $\mathcal{L}_{\text{BMSC}}$. Call the output Y. The receiver sees (T, Y) and estimates X.

Theorem 5.14 ([1], Theorem 4.94, p. 217)**.** *For a given degree distribution pair (λ, ρ) consider the sequence of associated ensembles LDPC(n, λ, ρ) of increasing blocklengths n under ℓ rounds of message-passing decoding. Then*

$$\lim_{n\to\infty} \mathbb{E}_{LDPC(n,\lambda,\rho)}\left[P_b^{BP}(G, \mathcal{L}_{BMSC}, \ell)\right] = P^{BP}_{\mathcal{T}(\lambda,\rho)}(\mathcal{L}_{BMSC}).$$

Since we have now asserted the convergence to the tree channel, we can start analysing its performance. Consider the distribution of the initial variable-to-check messages. By definition of the algorithm, these messages are equal to the received messages which have density $\mathcal{L}_{\text{BMSC}}$. Let a_ℓ denote the variable-to-check message density at iteration ℓ. Then, $a_0 = \mathcal{L}_{\text{BMSC}}$.

We will proceed by induction over ℓ. Assume that the density of the variable-to-check messages in the ℓ-th iteration is a_ℓ. Consider a check node of degree j. The update rule can be equivalently written as

$$2\tanh^{-1}\left(\prod_{k=1}^{j-1} \tanh\left(\frac{l_k}{2}\right)\right) = g^{-1}\left(\sum_{k=1}^{j-1} g(l_k)\right)$$

where summation of $g(\cdot)$ is defined as follows

$$\sum_{k=1}^{j-1}(s_k, y_k) = \left(\prod_{k=1}^{j-1} s_k, \sum_{k=1}^{j-1} y_k\right).$$

So, we see that the outgoing message of a degree-j check node is the sum of the G-representations of the $(j-1)$ independent incoming messages. It follows that the density of the outgoing message is

$$b_{\ell+1,j} = a_\ell^{\circledast(j-1)}$$

or in words, the $(j-1)$-times convolution of a_ℓ with itself. By averaging over ρ, we have

$$b_{\ell+1} = \sum_j \rho_j b_{\ell+1,j} = \sum_j \rho_j a_\ell^{\circledast(j-1)}$$

which, with a slight abuse of notation, can be written as

$$b_{\ell+1} = \rho(a_\ell).$$

Now consider a variable node of degree i. By definition of the algorithm, the variable-to-check message is the sum of the received message, which has L-density $\mathcal{L}_{\text{BMSC}}$ and the $(i-1)$ incoming check-to-variable messages, each of which has density $b_{\ell+1}$. Since all messages are statistically independent, it follows that the density of the outgoing message is the convolution of the densities of the summands, i.e.

$$a_{\ell+1,i} = \mathcal{L}_{\text{BMSC}} * b_{\ell+1}^{*(i-1)}.$$

Averaging again over the edge degree distribution, we see that the outgoing density is equal to

$$\begin{aligned} a_{\ell+1} &= \sum_i \lambda_i \mathcal{L}_{\text{BMSC}} * b_{\ell+1}^{*(i-1)} \\ &= \mathcal{L}_{\text{BMSC}} * \sum_i \lambda_i b_{\ell+1}^{*(i-1)} \\ &= \mathcal{L}_{\text{BMSC}} * \lambda(b_{\ell+1}) \\ &= \mathcal{L}_{\text{BMSC}} * \lambda(\rho(a_\ell)). \end{aligned}$$

So, the final update rule for the variable-to-check message densities for $\ell \geq 0$ reads

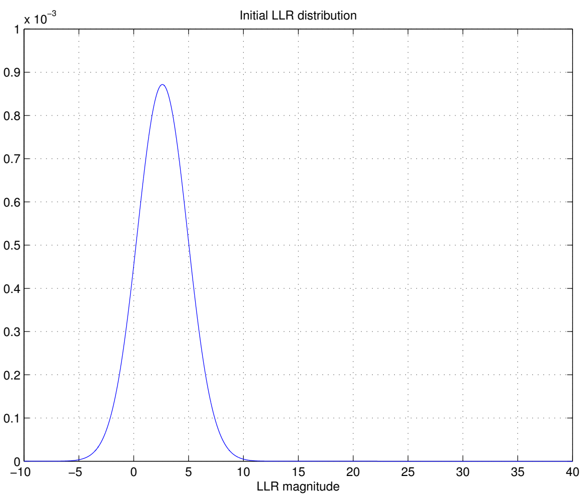
$$a_{\ell+1} = \mathcal{L}_{\text{BMSC}} * \lambda(\rho(a_\ell)), \qquad a_0 = \mathcal{L}_{\text{BMSC}}.$$

Using the same derivation, the update rule for the check-to-variable message densities for $\ell \geq 1$ reads

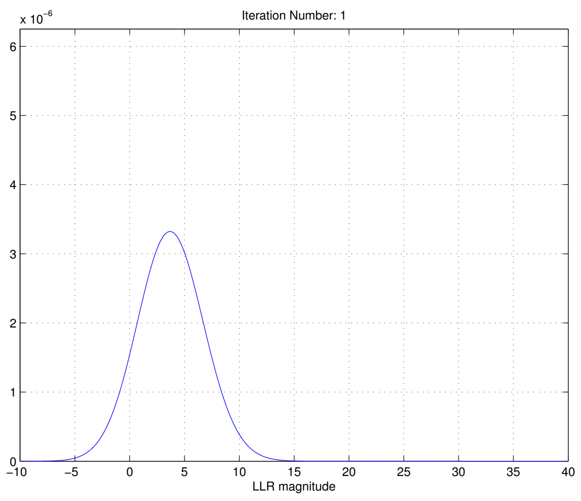
$$b_{\ell+1} = \rho(\mathcal{L}_{\text{BMSC}} * \lambda(b_\ell)), \qquad b_1 = \rho(a_0).$$

Note that, as maybe expected, if $a_0 = \mathcal{L}_{\text{BEC}(\epsilon)}(y)$, then these updates rules are equivalent to the simplified update rules we derived for the BEC channel in previous sections.

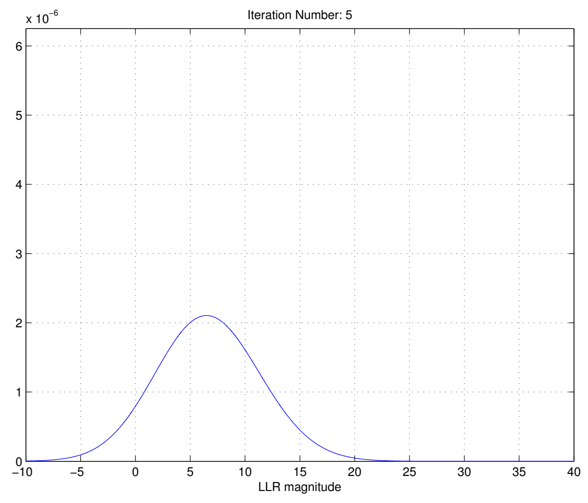
We will now illustrate density evolution for the $(3, 6)$ LDPC ensemble over the BI-AWGN channel. The threshold, in terms of noise variance, for this ensemble has been found to be $\sigma^* = 0.8747$. Since we now track whole densities and not just a number, this procedure can not be illustrated with just one graph. We will instead present the message densities after a certain number of iterations. In figure 5.4, we demonstrate density evolution for the case where $\sigma = 0.8747 = \sigma^*$, for $0, 1, 5$ and 15 iterations respectively.



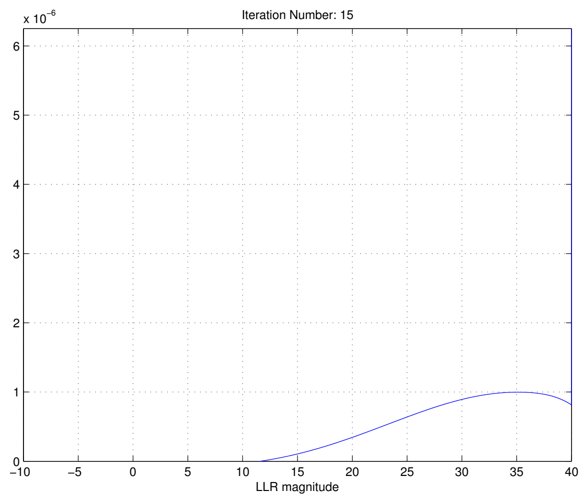
(a)



(b)



(c)

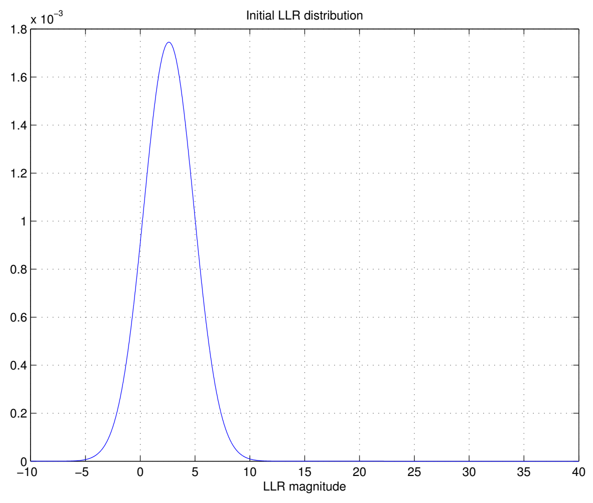


(d)

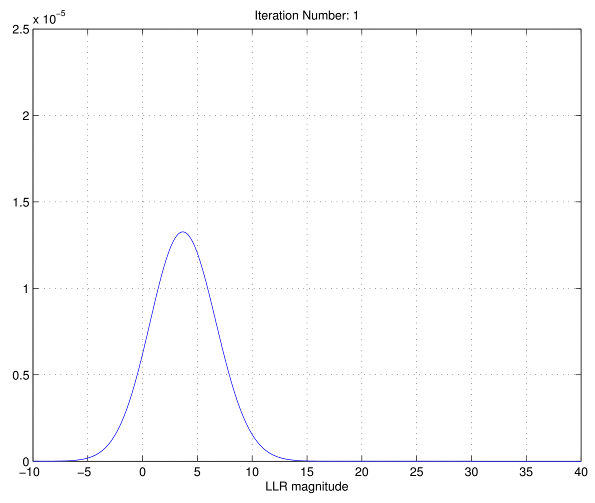
Figure 5.4: Density evolution for the BI-AWGN channel for $\sigma = \sigma^*$.

We see that, as the number of iterations increases, the message densities tend to the right, meaning that the corresponding bit error probability decreases. As the number of iterations tends to infinity, the message density will tend to a unit mass at $+\infty$. Our implementation is based on the discussion in [1, Appendix B]. For accurate results, we have to sample the densities at a large number of points, so the implementation is rather slow.

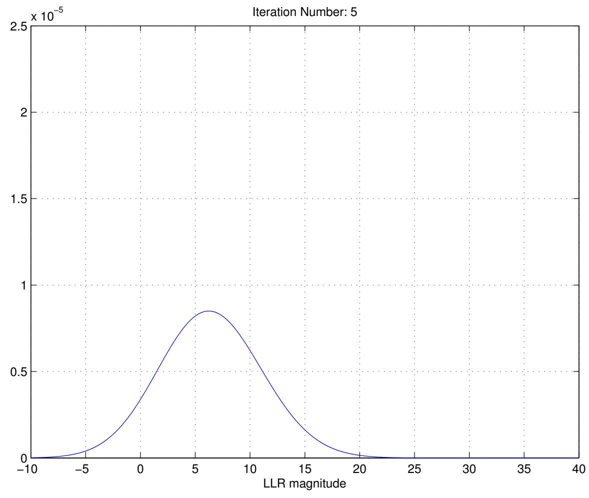
Now, if we choose a noise variance which exceeds the noise threshold of this ensemble, we get the following evolution of densities for $0, 1, 5$ and 15 iterations respectively.



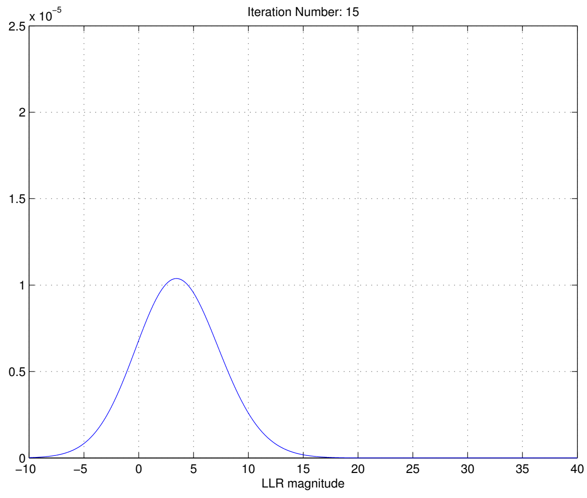
(a)



(b)



(c)



(d)

Figure 5.5: Density evolution for the BI-AWGN channel for $\sigma > \sigma^*$.

We see that the densities start moving to the right, but return to a fixed point after a certain number of iterations, leading to a non-zero error probability. In this case, even after infinite iterations, the message density will not tend to a point mass at $+\infty$.

Monotonicity

Due to the optimality of the belief propagation algorithm in the tree-like setting, is it natural to expect that the performance will improve if we increase the number of iterations or if the channel is improved. The following lemmas assert this expectation with respect to the channel and iteration number respectively.

Lemma 5.15 ([1], Lemma 4.106, p. 224)**.** *Let (λ, ρ) be a degree distribution pair and consider two BMS channels characterised by their L-densities $\mathcal{L}_{BMSC}$ and $\mathcal{L}'_{BMSC}$. If $\mathcal{L}_{BMSC}$ is a degraded version of $\mathcal{L}'_{BMSC}$, then $P^{BP}_{\mathcal{T}_\ell}(\mathcal{L}'_{BMSC}) \leq P^{BP}_{\mathcal{T}_\ell}(\mathcal{L}_{BMSC})$. In particular, if $P^{BP}_{\mathcal{T}_\ell}(\mathcal{L}_{BMSC}) \overset{\ell\to\infty}{\to} 0$, then $P^{BP}_{\mathcal{T}_\ell}(\mathcal{L}'_{BMSC}) \overset{\ell\to\infty}{\to} 0$.*

Lemma 5.16 ([1], Lemma 4.107, p. 225)**.** *For a given degree distribution pair (λ, ρ) define the operator $f(a, b) = a * \lambda(\rho(b))$. Let $\mathcal{L}_{BMSC}$ and a_0 denote two symmetric L-densities. For $\ell \geq 0$ define*

$$a_\ell = f(\mathcal{L}_{BMSC}, a_{\ell-1}).$$

If for some $\ell' \geq 0$ and $k \geq 1$ we have that $a_{\ell'}$ is a degraded version of $a_{\ell'+k}$, then for $\ell'' \geq \ell'$, ℓ'' and k fixed, the sequence $a_{\ell''+k\ell'}$ is monotone (with respect to degradation) in $\ell \geq 0$ and converges to a symmetric limit density.

Although $a_{\ell''+k\ell'}$ converges in ℓ for all ℓ' we cannot conclude that a_ℓ converges. In general, the sequence could converge to a limit cycle of length k. Such a limit cycle has never been observed and it is conjectured that limit cycles cannot occur and that a_ℓ in fact converges. The following corollary can substitute the conjecture.

Corollary 5.17 ([1], Corollary 4.108, p. 225)**.** *If in addition to the conditions of lemma 5.16 we have that $a_{\ell'}$ is a degraded version of $a_{\ell'+k+1}$, then a_ℓ converges to a limit density a_∞.*

Below is a simulation comparing the performance of a regular-$(3, 6)$ LDPC code, created using the PEG algorithm, under belief propagation decoding for maximum number of iterations equal to 20, 40 and 80. We use BPSK modulation with the mapping $\{0, 1\} \to \{-1, +1\}$. Noise variance is defined as

$$\sigma^2 = \frac{N_0}{2}$$

where

$$\mathrm{SNR}_{\mathrm{dB}} = 10 \log_{10}\left(\frac{1}{N_0}\right).$$

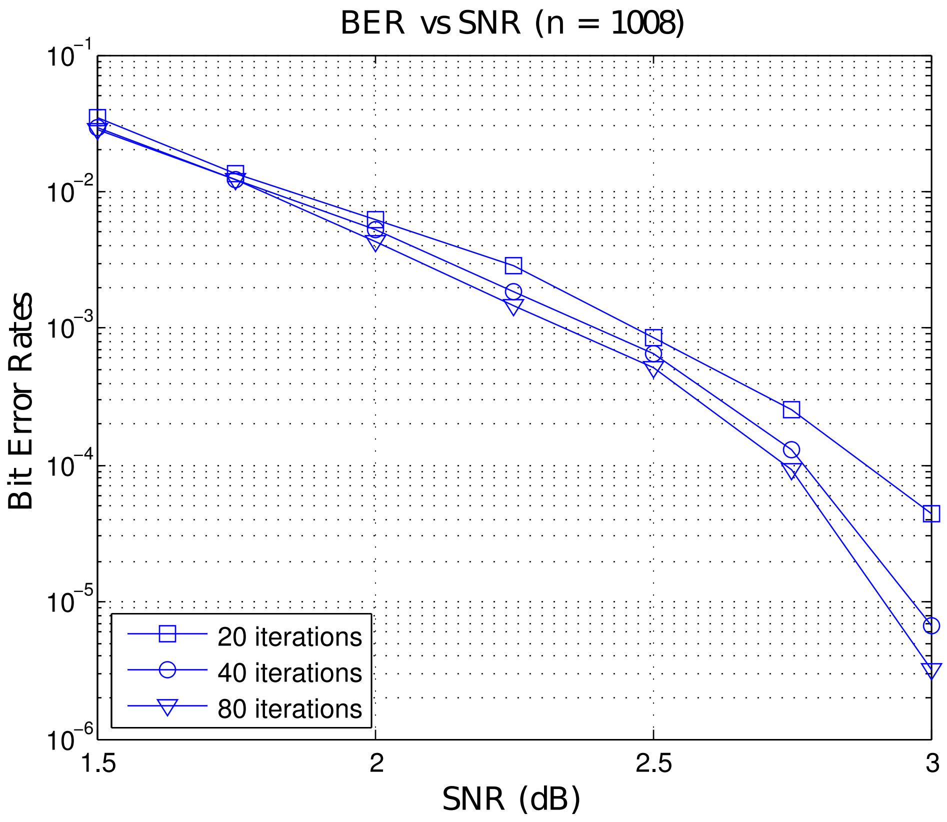


Figure 5.6: Performance of a regular LDPC code of rate 0.5 and length $n = 504$ bits for various maximum number of decoding iterations.

The monotonicity lemmas presented above hold for cycle-free codes. However, we see that even for this code, which is not cycle-free, increasing the maximum number of decoding iterations is beneficial.

Threshold

Consider a family of BMS channels $\{\mathcal{L}_{\mathrm{BMSC}}(\sigma)\}$ for $\underline{\sigma} \leq \sigma \leq \overline{\sigma}$. Assume that for a BP decoder it holds that

$$\lim_{\ell\to\infty} P^{\mathrm{BP}}_{\mathcal{T}_\ell}\left(\mathcal{L}_{\mathrm{BMSC}}(\underline{\sigma})\right) = 0 \quad \text{but} \quad \lim_{\ell\to\infty} P^{\mathrm{BP}}_{\mathcal{T}_\ell}\left(\mathcal{L}_{\mathrm{BMSC}}(\overline{\sigma})\right) > 0.$$

We also know that the BP decoder is monotone with respect to a BMS channel family. Then, there must exist a supremum of the set of σ for which $\lim_{\ell\to\infty} P^{\mathrm{BP}}_{\mathcal{T}_\ell}\left(\mathcal{L}_{\mathrm{BMSC}}(\sigma)\right) = 0$ so that for all smaller channel parameters the bit error probability is zero and for all larger channel parameters the bit error probability is non-zero. This supremum is called the threshold and is denoted σ^{BP}.

Definition 5.18. The threshold associated with the degree distribution pair (λ, ρ), call it $\sigma^{\mathrm{BP}}(\lambda, \rho)$, is defined as:

$$\sigma^{\mathrm{BP}}(\lambda, \rho) = \sup\{\sigma \in [\underline{\sigma}, \overline{\sigma}] : P^{\mathrm{BP}}_{\mathcal{T}_\ell}(\mathcal{L}_{\mathrm{BMSC}}(\sigma)) \overset{\ell\to\infty}{\to} 0\}$$

Stability Condition

In order to characterise the stability of the system, we will use the Bhattacharyya constant $\mathcal{B}(\cdot)$ which is defined as

$$\mathcal{B}(\mathcal{L}) = \int_{\mathbb{R}} \mathcal{L}(x) e^{-x/2} dx.$$

Theorem 5.19 ([1], Theorem 4.125, p. 232)**.** *Assume we are given a degree distribution pair (λ, ρ) and that transmission takes place over a BMS channel characterised by its L-density $\mathcal{L}_{BMSC}$ with Bhattacharyya constant $\mathcal{B}(\mathcal{L}_{BMSC})$. For $\ell \geq 1$ define $a_\ell(a_0) = a_\ell = \mathcal{L}_{BMSC} * \lambda(\rho(a_{\ell-1}))$ with a_0 an* arbitrary *symmetric density.*

1. *(Necessity) If $\mathcal{B}(\mathcal{L}_{BMSC})\,\lambda'(0)\rho'(1) > 1$, then there exists a strictly positive constant $\xi = \xi(\lambda, \rho, \mathcal{L}_{BMSC})$ such that $\liminf_{\ell\to\infty} e(a_\ell) \geq \xi$ for all $a_0 \neq \Delta_{+\infty}$.*
2. *(Sufficiency) If $\mathcal{B}(\mathcal{L}_{BMSC})\,\lambda'(0)\rho'(1) < 1$, then there exists a strictly positive constant $\xi = \xi(\lambda, \rho, \mathcal{L}_{BMSC})$ such that if, for some $\ell \in \mathbb{N}$, $e(a_\ell) \leq \xi$, then a_ℓ converges to $\Delta_{+\infty}$.*

Example 5.20. For the BEC, we have

$$\mathcal{B}(\mathcal{L}_{\mathrm{BEC}(\epsilon)}) = \int_{\mathbb{R}} \left[\epsilon\Delta_0(x) + (1-\epsilon)\Delta_{+\infty}(x)\right] e^{-x/2} dx = \epsilon.$$

Therefore, the stability condition reads $\epsilon\lambda'(0)\rho'(1) < 1$, which agrees with the result of the section 5.1.5 where we analysed BP for the BEC. ■

Example 5.21. For the BI-AWGN(σ) channel we have

$$\mathcal{B}(\mathcal{L}) = \int_{\mathbb{R}} \sqrt{\frac{\sigma^2}{8\pi}} \exp\left\{-\frac{\left(y - \frac{2}{\sigma^2}\right)^2 \sigma^2}{8}\right\} e^{-x/2} dx = e^{-\frac{1}{2\sigma^2}}.$$

Therefore the stability condition reads $e^{-\frac{1}{2\sigma^2}}\lambda'(0)\rho'(1) < 1$ which gives rise to the upper bound for σ^{BP}

$$\sigma^{\mathrm{BP}}(\lambda, \rho) \leq \frac{1}{\sqrt{2\ln(\lambda'(0)\rho'(1))}}.$$

■

5.3 Gaussian Approximation

Density evolution is generally an infinite-dimensional problem, since we need to track the whole densities, which are functions that can not be expressed in closed form. This, of course, can lead to an enormous computational complexity, depending on the sampling of the continuous densities we choose to make. In [16], the authors propose modelling the message of the BP algorithm as Gaussian RVs, based on the observation that, under some conditions, the message densities do indeed resemble Gaussian distributions.

This decreases the problem dimensionality to 2. In fact, we only need to track the mean of the Gaussian RV, denoted μ, since it is shown that, under a symmetry condition, it holds that

$$\sigma^2 = 2\mu$$

where σ^2 is the variance of the Gaussian RV. We will now derive the update rules for the density evolution algorithm using the Gaussian approximation.

First consider a degree-i variable node where $(i-1)$ incoming messages are added to the channel message to form the outgoing messages. Let the mean of the incoming messages at iteration $(\ell-1)$ be denoted $\mu_u^{(\ell-1)}$ and the mean of the channel message be denoted μ_{u_0}. At variable nodes, we perform simple addition to form the outgoing and the mean of a sum of RVs is the sum of their means, so the mean of the outgoing message for a degree-i variable, denoted $\mu_{v,i}^{(\ell-1)}$, will be

$$\mu_{v,i}^{(\ell-1)} = \mu_{u_0} + (i-1)\mu_u^{(\ell-1)}.$$

Consider a degree-j check node and recall that the update rule for BP for check nodes at iteration ℓ reads

$$u^{(\ell)} = 2\tanh^{-1}\left(\prod_{k=1}^{j-1}\tanh\left(\frac{v_k^{(\ell-1)}}{2}\right)\right) \tag{5.1}$$

or equivalently

$$\tanh\left(\frac{u^{(\ell)}}{2}\right) = \prod_{k=1}^{j-1}\tanh\left(\frac{v_k^{(\ell-1)}}{2}\right)$$

where $v_j^{(\ell-1)}$ are Gaussian RVs of mean $m_v^{(\ell-1)}$. Since all $v_k^{(\ell-1)}$ are equal, we drop the k index. Now, by taking means on both sides of the previous equation, we get

$$\begin{aligned} E\left[\tanh\left(\frac{u^{(\ell)}}{2}\right)\right] &= E\left[\prod_{k=1}^{j-1}\tanh\left(\frac{v_k^{(\ell-1)}}{2}\right)\right] \\ &= E\left[\left(\tanh\left(\frac{v^{(\ell-1)}}{2}\right)\right)^{(j-1)}\right] \\ &= E\left[\tanh\left(\frac{v^{(\ell-1)}}{2}\right)\right]^{(j-1)}. \end{aligned}$$

We have exploited the assumption that all messages are independent to move from the mean of a product, to a product of means. Since, by assumption, we have that $u^{(\ell)} \sim \mathcal{N}(\mu_u^{(\ell)}, 2\mu_u^{(\ell)})$, it holds by definition of the mean that

$$E\left[\tanh\left(\frac{u^{(\ell)}}{2}\right)\right] = \frac{1}{\sqrt{4\pi\mu_u^{(\ell)}}}\int_{-\infty}^{+\infty}\tanh\left(\frac{u^{(\ell)}}{2}\right)\exp\left\{-\frac{(u^{(\ell)}-\mu_u^{(\ell)})^2}{4\mu_u^{(\ell)}}\right\}du.$$

We will now define the auxiliary function $\phi(x)$

$$\phi(x) = \begin{cases} 1 - \frac{1}{\sqrt{4\pi x}}\int_{-\infty}^{+\infty}\tanh\left(\frac{u}{2}\right)\exp\left\{-\frac{(u-x)^2}{4x}\right\}du, & x > 0, \\ 1, & x = 0. \end{cases}$$

For practical purposes, the following approximation for $\phi(x)$ is commonly used

$$\hat{\phi}(x) = \begin{cases} e^{-0.4527x^{0.86}+0.0218}, & x < 10, \\ \sqrt{\frac{\pi}{x}} e^{-\frac{x}{4}} \left(1 - \frac{20}{7x}\right), & x \geq 10. \end{cases}$$

The first part of $\hat{\phi}(x)$ (for $x < 10$) is easily invertible. The second part is a fairly well-behaved function and can be inverted using a lookup table.

Using $\phi(x)$ and averaging over $\lambda(x)$, we get

$$E\left[\tanh\left(\frac{v^{(\ell-1)}}{2}\right)\right] = 1 - \sum_i \lambda_i \phi\left(\mu_{v,i}^{(\ell-1)}\right).$$

Using (5.1), we get

$$\mu_{u,j}^{(\ell)} = \phi^{-1}\left(1 - \left[1 - \sum_i \lambda_i \phi\left(\mu_{v,i}^{(\ell-1)}\right)\right]^{j-1}\right) \tag{5.2}$$

which gives the mean of the messages emanating from degree-j check nodes, based on the received messages. Further expanding using the update rule for $m_{v,i}^{(\ell-1)}$, we get

$$\mu_{u,j}^{(\ell)} = \phi^{-1}\left(1 - \left[1 - \sum_i \lambda_i \phi\left(\mu_{u_0} + (i-1)\mu_u^{(\ell-1)}\right)\right]^{j-1}\right).$$

So, we have reached a point where we can calculate the mean of messages emitted from check nodes of degree j as a function of the mean of the messages which were emitted in the previous iteration. As a last step, we need to average again over $\rho(x)$ in order to get the overall mean, so

$$\mu_u^{(\ell)} = \sum_j \rho_j \phi^{-1}\left(1 - \left[1 - \sum_i \lambda_i \phi\left(\mu_{u_0} + (i-1)\mu_u^{(\ell-1)}\right)\right]^{j-1}\right).$$

By following the same approach, we can also derive an update rule for the mean of messages emitted from variable nodes. We start with the update rule

$$\mu_{v,i}^{(\ell)} = \mu_{u_0} + (i-1)\mu_u^{(\ell)}.$$

Using (5.2) and averaging over $\rho(x)$, we get

$$\mu_v^{(\ell)} = \sum_i \lambda_i \phi\left(\mu_{u_0} + (i-1)\sum_j \rho_i \phi^{-1}(1 - (1 - \mu_v^{(\ell-1)})^{j-1})\right).$$

We have shown how we can perform density evolution under the assumption that the exchanged messages follow a Gaussian distribution, greatly reducing the computational complexity in the process. Results using the Gaussian approximation are of course not as precise as the results we can get with full density evolution, but some examples in [16] indicate that the precision-complexity trade-off is relatively favourable.

For $0 < s < \infty$ and $0 \leq t < \infty$, we define $f_j(s,t)$ and $f(s,t)$ as

$$f_j(s,t) \triangleq \phi^{-1}\left(1 - \left[1 - \sum_i \lambda_i \phi\left(s + (i-1)t\right)\right]^{j-1}\right)$$

$$f(s,t) \triangleq \sum_j \rho_j f_j(s,t).$$

Using this we can write

$$t_\ell = f(s, t_{\ell-1}).$$

The threshold now takes on the following form.

Definition 5.22. The threshold s^* is defined as the infimum of all $s \in \mathbb{R}^+$ such that $t_\ell(s)$ converges to ∞ as $\ell \to \infty$.

A sufficient convergence condition is given by the following lemma

Lemma 5.23 ([16], Lemma 2)**.** *$t_\ell(s)$ will converge to ∞ if $t < f(s,t)$ for all $t \in \mathbb{R}^+$.*

Below we see the evolution of the mean of messages emitted from check nodes for the $(3,6)$ ensemble of LDPC codes. The threshold, in terms of noise variance, for this ensemble has been found to be $\sigma^* = 0.8747$.

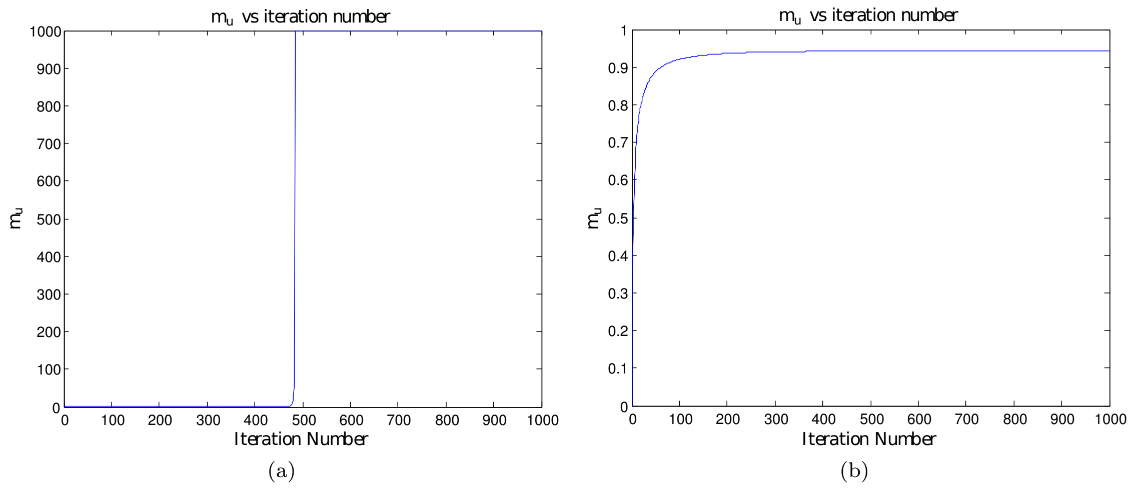


Figure 5.7: Density evolution for the BI-AWGN channel using a Gaussian approximation.

In figure 5.7(a), where the noise level is exactly on the threshold ($\sigma = 0.8747$), we see that the mean of messages converges to the maximum value of our lookup table for ϕ^{-1}. On the other hand, in figure 5.7(b), where the noise level is over the threshold ($\sigma = 0.8748$), we see that the mean of messages converges to a small fixed value, resulting in a strictly positive probability of error.

5.4 Design Using Density Evolution

We have seen that, with the help of density evolution, we can predict the asymptotic performance of a given LDPC code ensemble. In this section, we will present a way of exploiting density evolution in order to search for degree distributions with good properties. Many approaches to this problem exist, but we will present only one of the most common ones.

In this approach, we fix the target SNR (or erasure probability for the BEC) at which the code has to operate successfully. Furthermore, the check node degree distribution is chosen to be concentrated, which is justified by the fact that most good codes have concentrated check node degree distributions. We have already greatly simplified the problem, since we only need to seek for a good variable node degree distribution. The problem now consists of maximising the rate of the code for a given check node degree distribution while ensuring convergence of density evolution for the given SNR (or erasure probability). We know that the rate of an LDPC code can be expressed as

$$r = 1 - \frac{\int_0^1 \rho(x)dx}{\int_0^1 \lambda(x)dx} = 1 - \frac{\sum_i \rho_i/i}{\sum_i \lambda_i/i}.$$

So, in order to maximise the rate for fixed $\rho(x)$, we need to maximise $\sum_i \lambda_i/i$. We will focus on the case of optimisation for the BI-AWGN channel using a Gaussian approximation for density evolution. The first constraint for our maximisation problem is

$$\sum_i \lambda_i = 1.$$

For the second constraint, using lemma 5.23 we have

$$\mu_v^{(\ell)} < \mu_v^{(\ell+1)}, \qquad \ell = 1, 2, \ldots, L.$$

Since we can not run density evolution for an infinite number of iterations, we choose a maximum number of iterations L. We can choose this number to be equal to the maximum number of iterations our decoder is allowed to make. Also, since in practice we need a lookup table for ϕ^{-1}, if the maximum value the lookup table can handle is reached, we consider this as valid convergence. Using these constraints, we can formulate the following linear program

$$\begin{aligned} \max_{\lambda(x)} \quad & \textstyle\sum_i \lambda_i/i \\ \text{s.t.} \quad & \textstyle\sum_i \lambda_i = 1 \\ & \mu_v^{(\ell)} < \mu_v^{(\ell+1)}, \quad \ell = 1, 2, \ldots, L. \end{aligned}$$

Our choice of algorithm for this maximisation is the Differential Evolution algorithm, which belongs to the family of genetic algorithms. Below, we present a degree distribution which was obtained using this approach. We fixed the SNR to 0 (i.e. $\sigma^2 = 1$) and the check node degree to 8. The maximum variable node degree was set to 16. We obtained the following distribution:

$$\begin{aligned} \lambda(x) &= 0.118433x + 0.424942x^2 + 0.0003415x^3 + 0.000253x^4 + 0.001209x^5 + 0.000741x^6 \\ &\quad + 0.00255x^7 + 0.000685x^8 + 0.000169x^9 + 0.001085x^{10} + 0.00163x^{11} + 0.000895x^{12} \\ &\quad + 0.00221x^{13} + 0.007088x^{14} + 0.437768x^{15} \\ \\ \rho(x) &= x^7 \end{aligned}$$

which has a rate of 0.46. By looking up the best code at [13] for the same maximum variable node degree, we see that its threshold is $\sigma^* = 1.00776$, meaning that our code works well for noise variances just 0.00776 smaller that the best code found so far.

Similarly, for the BEC with $\epsilon = 0.42943$ and maximum variable node degree equal to 11, we found the following degree distribution

$$\lambda(x) = 0.316995x + 0.181947x^2 + 0.11862x^3 + 0.382436^{10}, \qquad \rho(x) = x^7.$$

This distribution has rate 0.55919 which is only 0.01138 away from the capacity of the channel, which is $C_{\text{BEC}} = 1 - \epsilon = 0.57057$. Of course, more constraints can be added to the problem, such as the stability condition for the BEC for example. However, even with this very simple approach, we can obtain very good codes.

We see that the design problem actually consists of finding the right parameter to maximise (or minimise) and the corresponding constraints and then finding an efficient algorithm for this maximisation, like the differential evolution algorithm. MATLAB code for solving the linear programs presented in this section is available at `http://www.telecom.tuc.gr/~alex/`.

5.5 EXIT Charts

In the section, we will take a glimpse at EXIT charts, which are a visualisation of the asymptotic performance under BP decoding. We only consider the case of the BEC which is simple.

Consider a degree distribution pair (λ, ρ). Recall that the asymptotic behaviour of such a degree distribution pair is described by $f(\epsilon, x) = \epsilon\lambda(1 - \rho(1 - x))$, which represents the evolution of the fraction of erased messages emitted by variable nodes, assuming that the system is in *state* (i.e. has current such fraction) x and that the channel parameter is ϵ. It is helpful to represent $f(\epsilon, x)$ as the composition of two functions, one which represents the "action" of the variable nodes and one which represents the "action" of the check nodes. We define $\nu_\epsilon(x) = \epsilon\lambda(x)$ and $c(x) = 1 - \rho(1 - x)$, so that $f(\epsilon, x) = \nu_\epsilon(c(x))$. Recall that the condition for convergence reads $f(\epsilon, x) < x$, $\forall x \in (0, 1)$. Observe that $\nu_\epsilon(x)$ has an inverse for $x \geq 0$ since $\lambda(x)$ is a polynomial with non-negative coefficients. The condition for convergence can hence be written as

$$c(x) < \nu_\epsilon^{-1}(x), \ \forall x \in (0, 1).$$

This has a pleasing graphical interpretation: $c(x)$ has to lie strictly below $\nu_\epsilon^{-1}(x)$ over the whole range $x \in (0, 1)$. The threshold ϵ^{BP} can then be defined as the supremum of all numbers ϵ for which this condition is fulfilled. This graphical representation is called an EXIT chart.

An equivalent graphical representation can be derived for BMS channels, but it is much more complicated and, since we do not focus on EXIT charts, will not be analysed here.

Chapter 6

Construction of LDPC Codes

After designing a code, which in fact means finding a good degree distribution for the specific channel scenario, we have to construct a code from the optimised ensemble. As already mentioned, it is vital to choose a good graph (or, equivalently, a good parity-check matrix) representation of the ensemble in order to achieve the best possible performance. Various constructions exist, each having its pros and cons. We will present some of them in the following chapters, starting with the historically first construction, namely Gallager codes, and concluding with our choice of construction, the Progressive Edge Growth algorithm.

6.1 Random Constructions

6.1.1 Gallager Codes

This is the original method for constructing regular LDPC codes, first introduced by Gallager in his 1962 dissertation [3].

To construct a parity-check matrix $\mathbf{H}$ with column weight w_c and row weight w_r, we first construct a sub-matrix $\mathbf{H}_1$ which contains a single non-zero entry in each column and w_r non-zero entries in each row. For row $i \in \{1, 2, \ldots\}$, these non-zero entries are placed in columns $(i-1)w_r + 1$ to iw_r. For this construction to be possible, we have to choose the blocklength n as a multiple of w_r. $\mathbf{H}_1$ has n columns and $\frac{n}{w_r}$ rows.

To construct the final parity-check code, we create $(w_c - 1)$ pseudorandom permutations of $\mathbf{H}_1$ and concatenate them

$$\mathbf{H} \triangleq \begin{bmatrix} \mathbf{H}_1 \\ \mathbf{H}_2 \\ \vdots \\ \mathbf{H}_{w_c} \end{bmatrix}.$$

The resulting code has row weight w_r, column weight w_c (since we concatenate w_c matrices with a single non-zero entry in each column) and design rate $r = \frac{nw_r - nw_c}{nw_r} = 1 - \frac{w_c}{w_r}$. The code's actual rate might be higher since we have no guarantee that the resulting parity-check matrix has full rank.

Example 6.1. We will demonstrate the above construction with a simple example. Suppose that we wish to construct a regular code with $w_r = 6$, $w_c = 3$ and $n = 12$. The design rate of this code is $r = 1 - \frac{w_c}{w_r} = \frac{1}{2}$.

We will first construct $\mathbf{H}_1$

$$\mathbf{H}_1 = \begin{bmatrix} 1&1&1&1&1&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&1&1&1&1&1&1 \end{bmatrix}.$$

Observe that $\mathbf{H}_1$ has indeed $\frac{n}{w_r} = 2$ rows. Also note that the row 1 contains non-zero entries in columns $(i-1)w_r+1 = 1$ to $iw_r = 6$, and row 2 contains non-zero entries in columns $(i-1)w_r+1 = 7$ to $iw_r = 12$.

To construct $\mathbf{H}$, we need two random permutations of $\mathbf{H}_1$, say

$$\mathbf{H}_2 = \begin{bmatrix} 1&0&1&0&1&0&1&0&1&0&1&0 \\ 0&1&0&1&0&1&0&1&0&1&0&1 \end{bmatrix},$$

$$\mathbf{H}_3 = \begin{bmatrix} 1&1&0&0&1&1&0&0&1&0&1&0 \\ 0&0&1&1&0&0&1&1&0&1&0&1 \end{bmatrix}.$$

By vertically concatenating the above matrices we get

$$\mathbf{H} = \begin{bmatrix} \mathbf{H}_1 \\ \mathbf{H}_2 \\ \mathbf{H}_3 \end{bmatrix} = \begin{bmatrix} 1&1&1&1&1&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&1&1&1&1&1&1 \\ 1&0&1&0&1&0&1&0&1&0&1&0 \\ 0&1&0&1&0&1&0&1&0&1&0&1 \\ 1&1&0&0&1&1&0&0&1&0&1&0 \\ 0&0&1&1&0&0&1&1&0&1&0&1 \end{bmatrix}.$$

Note that the above matrix is not full rank (many columns are pairwise equal), thus the actual rate will be higher than the design rate. ■

As the block length increases, the probability that any two columns are equal or linearly dependent decreases. Thus, the probability that the actual rate is close to the design rate increases. This construction is rather simple, but it gives no guarantee for the rank of the resulting parity-check matrix, it does not avoid small cycles and it can only create regular codes. Another drawback of this method is that we can not create codes that are encodable with linear time complexity.

An alternative approach to Gallager Codes is through the superposition of permutation matrices. The superposed matrices are generated at random, subject to the constraint that no two non-zero entries coincide, since we do not allow double edges between vertices in the code's Tanner graph. This can be made a bit more clear through a quick example:

Example 6.2. Suppose that we wish to create a length-n regular LDPC code from the $(3, 6)$ ensemble. This code has a design rate of $1/2$, so $m = n/2$. We can achieve this by constructing a parity-check matrix with the following structure.

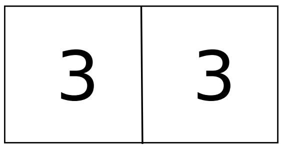


Figure 6.1: Regular parity-check matrix using superposed permutation matrices.

Each square corresponds to an $m \times m$ matrix and the number inside the square denotes the number of random permutation matrices which have been superposed. This matrix has exactly 3 non-zero entries per column and 6 non-zero entries per row. The code's actual rate might be higher than the design rate since we have no guarantee that the resulting parity-check matrix has full rank. ■

6.1.2 Poisson Construction

Using the approach mentioned above, we can construct irregular codes by splitting the parity-check matrix into areas of different degree, according to the degree distribution, and superposing the appropriate number of permutation matrices in each area.

Example 6.3. Suppose that we wish to create a length-n irregular LDPC code with the following degree distributions

$$\lambda(x) = \frac{11}{12}x^2 + \frac{1}{12}x^8, \qquad \rho(x) = x^6.$$

We can achieve this by constructing a parity-check matrix with the following structure.

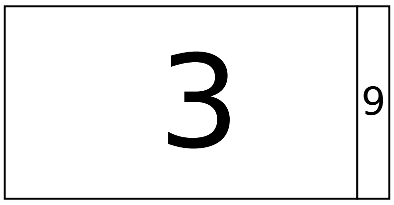


Figure 6.2: Irregular parity-check matrix using superposed permutation matrices.

This representation is a bit more abstract since the submatrices are not square. However, this matrix has exactly 3 non-zero entries per column for 11/12 of the columns and exactly 9 non-zero entries for 1/12 of the columns. To ensure that the row weight is constant and equal to 7, several methods of constructing the above type of parity-check matrix using square permutation matrices, such as super-Poisson and sub-Poisson, are proposed and analysed in [17].

6.1.3 Random Codes

Another method for constructing regular codes is randomly connecting the vertices of a Tanner graph, subject to the constraint that no length-4 cycles are created. Suppose that the code we wish to construct has column weight w_c and row w_r. The algorithm is initialised with an empty 0×0 parity-check matrix **H**.

```
repeat
    Select an $m \times 1$ column of weight $w_c$ at random, which is not already being used in H and has not
    been rejected in previous steps, and add it to H.
    if No length-4 cycle is created and all rows have weight no more than $w_r$ then
        Save new column to H and continue.
    else
        Reject new column and continue.
    end if
until $m$ columns have been added to H.
```

Figure 6.3: Random Construction for Regular Codes

To check for length-4 cycles, we can simply check if the candidate column has more than one non-zero element in common with any of the columns already in **H**. This can be seen if we recall the way in which parity-check matrices are associated with Tanner graphs. More precisely, the following type of construct

$$\mathbf{H} = \begin{bmatrix} & \vdots & & \vdots & \\ \dots & 1 & \dots & 1 & \dots \\ & \vdots & & \vdots & \\ \dots & 1 & \dots & 1 & \dots \\ & \vdots & & \vdots & \end{bmatrix}$$

results in the following construct in the corresponding Tanner graph.

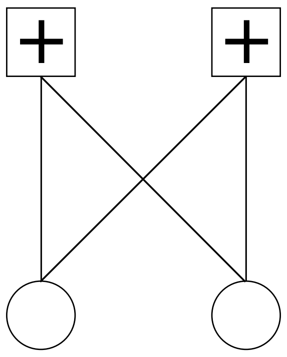


Figure 6.4: A length-4 cycle corresponding to two columns having two common non-zero elements.

We have no guarantee that the performance of this randomly constructed finite-length code will be close to the ensemble average performance, nor that the actual rate will be equal to the design rate. However, in the limit of infinite blocklengths, we have seen that both performance and actual rate are arbitrarily close to the ensemble performance and design rate respectively with high probability. This construction is widely used to construct codes of large blocklength (i.e. 10^5 bits or more) where the asymptotic analysis results are relatively valid. On the downside, encoding in linear time is not possible with this construction.

Random constructions can be extended to irregular codes by simply adding the columns according to the given degree distribution. For example, for the $(7, \lambda_H, \rho_H)$ Hamming code ensemble, we have to add 3 columns of weight 1, 3 columns of weight 2 and 1 column of weight 3, each chosen at random as with the algorithm for regular constructions.

6.2 Algebraic Constructions

6.2.1 Quasi-Cyclic Codes

Quasi-Cyclic LDPC codes have a parity-check matrix which consists of square blocks which are either full-rank circulants, or zero matrices. The $L \times L$ circulant matrix used is

$$\mathbf{P} = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ 1 & 0 & 0 & \dots & 0 \end{bmatrix}.$$

Powers of the circulant matrix (i.e. right shifts of the identity matrix) and the zero matrix (denoted $\mathbf{P}^{\infty}$) are used to construct $\mathbf{H}$. The $mL \times nL$ parity-check matrix is constructed as follows

$$\mathbf{H} = \begin{bmatrix} \mathbf{P}^{a_{11}} & \mathbf{P}^{a_{12}} & \dots & \mathbf{P}^{a_{1n}} \\ \mathbf{P}^{a_{21}} & \mathbf{P}^{a_{22}} & \dots & \mathbf{P}^{a_{2n}} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{P}^{a_{m1}} & \mathbf{P}^{a_{m2}} & \dots & \mathbf{P}^{a_{mn}} \end{bmatrix}$$

where $a_{ij} \in \{1, 2, \dots, L-1, \infty\}$[1]. It can be easily observed that the block length is nL, the number of parity checks is mL and the design rate is $\frac{nL-mL}{nL} = \frac{n-m}{n}$. This construction is mainly used for constructing regular codes, but it also carries the ability to create irregular codes, if a suitable choice of the coefficients a_{ij} is made. More precisely, since we can choose $a_{ij} = \infty$, resulting in a zero matrix, we can modify the column weight distribution of the resulting block matrix to match that of the desired variable degree distribution of the code.

As shown in [18], if we force the coefficients of the submatrices which are below the diagonal of $\mathbf{H}$ to be equal to ∞, and the coefficients of the submatrices which lie exactly on the diagonal to be equal to $L-1$, the resulting code can be efficiently encoded. As chance would have it, we are once again faced with a construction which does not guarantee the absence of short cycles, nor the equality of the actual rate with the design rate.

6.2.2 Array Codes and Modified Array Codes

Array Codes are in fact a special case of Quasi-Cyclic codes, where $a_{ij} = (i-1)(j-1)$. For a prime $L = q$ ($\mathbf{P}$ is defined as in QC-LDPC codes) and the integers $m \leq n \leq q$, we create the matrix

$$\mathbf{H} = \begin{bmatrix} \mathbf{I} & \mathbf{I} & \dots & \mathbf{I} & \dots & \mathbf{I} \\ \mathbf{I} & \mathbf{P}^{1} & \dots & \mathbf{P}^{(m-1)} & \dots & \mathbf{P}^{(n-1)} \\ \mathbf{I} & \mathbf{P}^{2} & \dots & \mathbf{P}^{2(m-1)} & \dots & \mathbf{P}^{2(n-1)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \mathbf{I} & \mathbf{P}^{(m-1)} & \dots & \mathbf{P}^{(m-1)(m-1)} & \dots & \mathbf{P}^{(m-1)(n-1)} \end{bmatrix}.$$

It can be easily observed that the block length is nq, the number of parity checks is mq, and the design rate is $\frac{nq-mq}{nq} = \frac{n-m}{n}$.

This type of construction results in an encoder which is even simpler than that of Quasi-Cyclic codes, since we do not need to store the coefficients a_{ij}. Unfortunately, no structure for quick encoding is provided. On a more pleasant note, for $j > 3$, it can be proven that no length-4 cycles exist and that the resulting parity-check matrix has full row rank, so the actual rate equals the design rate. Due to reduced randomness however, Array codes tend to perform worse than Gallager codes of the same ensemble.

Using the same modification we used for Quasi-Cyclic codes, we can construct Array codes which are encodable with linear time complexity. We simply force the submatrices which are below the diagonal of $\mathbf{H}$ to be zero matrices, and the submatrices which lie exactly on the diagonal to be identity matrices.

[1]Note that $\mathbf{P}^{(L-1)} = \mathbf{I}$.

Again, for a prime $L = q$ and the integers $m \leq n \leq q$, we create the matrix

$$\mathbf{H} = \begin{bmatrix} \mathbf{I} & \mathbf{I} & \mathbf{I} & \dots & \mathbf{I} & \dots & \mathbf{I} \\ \mathbf{0} & \mathbf{I} & \mathbf{P}^1 & \dots & \mathbf{P}^{(m-2)} & \dots & \mathbf{P}^{(n-2)} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} & \dots & \mathbf{P}^{2(m-3)} & \dots & \mathbf{P}^{2(n-3)} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots & \mathbf{I} & \dots & \mathbf{P}^{(m-1)(n-m)} \end{bmatrix}.$$

The quick encoding property is added to the existing list of convenient attributes, which still hold.

6.2.3 Finite Geometry Codes

A finite geometry is any geometric system that has only a finite number of points. Euclidean geometry over real numbers, for example, is not finite, because a Euclidean line contains infinitely many points, in fact precisely the same number of points as there are real numbers. A finite geometry can have any (finite) number of dimensions. Codes based on the points and lines of finite geometries were introduced and studied thoroughly in [14].

Let G be a finite geometry with n points and m lines which has the following fundamental structural properties:

1. Every line consists of ρ points.
2. Any two points are connected by one and only one line.
3. Every point is intersected by γ lines (i.e. every point lies on γ lines).
4. Any two lines are either parallel (i.e. they have no point in common) or they intersect at one and only one point.

There are two families of finite geometries which have the above fundamental structural properties, namely Euclidean and projective geometries over finite fields. Form an $m \times n$ matrix $\mathbf{H}_G$ whose rows and columns correspond to the lines and points of the finite geometry G respectively, where $\mathbf{h}_{ij} = 1$ if and only if the i-th line of G contains the j-th point of G. Each row has weight ρ and each column has weight γ. If $\rho\ n$ is small, $\mathbf{H}_G$ is sparse and thus defines a low-density-parity check code.

This code is regular, since the column and row weights are constant, and has a design rate of $r = 1 - \frac{\gamma}{\rho}$. However, there is no guarantee that the columns of $\mathbf{H}_G$ are linearly independent, so the actual rate may be higher. It has been proven that the girth of the graph corresponding to the resulting parity-check matrix, is 6, so no length-4 cycles exist [14]. Lower bounds for the minimum distance, which is of vital importance for the performance in the error floor region, are also derived. It is also notable that the parity-check matrices of these codes are either in quasi-cyclic or cyclic form, so an encoder which is efficient both in terms of time and memory complexity is feasible.

More information on finite geometries and finite geometry based LDPC codes can be found at [11, Chapter 17].

6.2.4 Graph Theoretic Codes

Much like finite geometries, other branches of combinatorial mathematics can be used for constructing LDPC codes. In this section, we will discuss the use of *graph theory* for the construction of a parity-check matrix.

Let $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ be a connected graph with vertex set $\mathcal{V} = \{v_1, v_2, \ldots, v_q\}$ and edge set $\mathcal{E} = \{e_1, e_2, \ldots, e_L\}$. We require that $\mathcal{G}$ does not contain self self-loops, and two vertices in $\mathcal{G}$ are connected by at most one edge (i.e. no multiple edges between two vertices).

Definition 6.4. Two paths in $\mathcal{G}$ are said to be *disjoint* if they do not have any vertex in common.

Definition 6.5. Two paths are said to be *singularly crossing* each other if they have one and only one vertex in common.

For $1 < \gamma \leq q$, let $\mathcal{P}$ be the set of paths of length $(\gamma - 1)$ in $\mathcal{G}$ that satisfies the following constraint: *Any two paths in $\mathcal{P}$ are either disjoint or singularly crossing each other.* This constraint is called the *disjoint crossing* (DC) constraint. Let $n = |\mathcal{P}|$ denote the number of paths in $\mathcal{P}$, i.e. the cardinality of the set. Let q_0 denote the number of vertices in $\mathcal{G}$ that are covered by the paths in $\mathcal{P}$. For simplicity, we call the vertices in $\mathcal{G}$ that are covered by the paths in $\mathcal{P}$, the vertices of $\mathcal{P}$. We form a $q_0 \times n$ matrix $\mathbf{H}$ whose rows correspond to the q_0 vertices in $\mathcal{P}$ and whose columns correspond to the n paths in $\mathcal{P}$, where $\mathbf{h}_{ij} = 1$ if and only if the i-th vertex v_i of $\mathcal{P}$ is on the j-th path in $\mathcal{P}$. This matrix is called the *incidence matrix* of $\mathcal{P}$.

Because the length of each path in $\mathcal{P}$ is $(\gamma - 1)$, there are γ vertices on each path. Therefore, each column of $\mathbf{H}$ has weight exactly γ. It follows from the DC constraint that no two columns or rows can have more than one 1-component in common. This means that no cycles of length 4 can exist in the corresponding Tanner graph. Now, if γ is significantly smaller than q_0, the incidence matrix $\mathbf{H}$ is sparse and thus its nullspace defines a low-density parity-check code. If, in addition, each vertex of $\mathcal{P}$ is is intersected by exactly ρ paths, all rows of $\mathbf{H}$ will have the same weight, resulting in a (γ, ρ) regular LDPC code.

A detailed description of an algorithm which can find such sets of paths can be found at [11, Chapter 16]. Some constructions of LDPC codes with large girth based on specific classes of graphs have been proposed, e.g. [19].

6.3 Progressive Edge Growth

As we have seen, the unfavourable constructs containing cycles in randomly generated graphs, have a probability which tends to 0 as the block length tends to infinity. However, for a code to be practical, its length can not be arbitrarily large. For short or moderately lengthed LDPC codes (up to, say, some thousand bits), the unfavourable constructs have relatively high probability, thus having non-negligible impact on the code's decoding performance. Our goal is to construct a graph having as large a girth as possible, given the code length and degree distribution, which is a rather hard combinatorial problem. However, a suboptimal yet simple and well performing algorithm was proposed in [9].

The basic idea is pretty straightforward: the graph is constructed in an edge-by-edge manner with each edge placed so that it has the minimum possible impact on the overall graph girth. This means that, fundamentally, an edge is placed between the variable node in question and the most distant check node in the graph. In the optimal case where the distance is infinite, i.e. no path exists between the variable node and the check node, the new edge creates no additional cycles.

Given the number of variable nodes, the number of check nodes and the variable node degree distribution, this algorithm assigns degrees to each variable node according to the degree distribution and calculates the degree distribution of the check nodes, making it as uniform as possible.

Before describing the algorithm, we need to explain the notation used. c_i and s_i denote the i-th check and variable node respectively. E_{s_i} denotes the set containing all edges incident to variable node s_i. $\mathcal{N}_{s_i}^{l}$ is called the *neighbourhood* of variable node s_i at depth l, and is defined as the set containing all check nodes reached by a subgraph spreading from variable node s_i within depth l. $\bar{\mathcal{N}}_{s_i}^l$ is the complement of $\mathcal{N}_{s_i}^l$, containing all check nodes which are *not* reached by a subgraph spreading from variable node s_i within depth l. d_{s_i} denotes the degree of variable node i.

```
for $j = 0$ to $n - 1$ do
    for $k = 0$ to $d_{s_j} - 1$ do
        if $k = 0$ then
            $E_{s_j}^0 \leftarrow$ edge $(c_i, s_j)$, where $E_{s_j}^0$ is the first edge incident to $s_j$ and $c_i$ is a check node such
            that it has the lowest check-node degree under the current graph setting $E_{s_0} \cup E_{s_1} \cup \ldots \cup E_{s_{j-1}}$.
        else
            Expand a subgraph from symbol node $s_j$ up to depth $l$ under the current graph setting such
            that the cardinality of $\mathcal{N}_{s_j}^l$ stops increasing but is less than $m$, or $\bar{\mathcal{N}}_{s_j}^l \neq \emptyset$ but $\bar{\mathcal{N}}_{s_j}^{l+1} = \emptyset$,
            then $E_{s_j}^k \leftarrow$ edge $(c_i, s_j)$ where $E_{s_j}^k$ is the $k$-th edge incident to $s_j$ and $c_i$ is a check node
            picked from the set $\bar{\mathcal{N}}_{s_j}^l$ having the lowest check-node degree.
        end if
    end for
end for
```

Figure 6.5: The Progressive Edge Growth Algorithm

Since the above pseudocode might not make much sense at first, a brief explanation is in order. For each variable node, the first connection is chosen at random from the set of check nodes which have the fewest connections up to that time, since a single connection can not create a cycle. For the remaining $(d_{s_j} - 1)$ edges, we seek for check nodes which are unreachable from s_j. This happens if the cardinality of $\mathcal{N}_{s_j}^l$ stops increasing. If the cardinality reaches m, the furthest check nodes will be the ones that were reached with the last step, so we chose a minimum degree node from that set, at random.

We will now present a simulation result which compares a randomly generated code with a code generated with the PEG algorithm. Both codes are created from the $(3, 6)$ regular ensemble. For the random code we use the random construction described in chapter 6.1.3, while for the second code we use the PEG algorithm. The maximum number of iterations of the belief propagation algorithm is set to 120. We use BPSK modulation with the mapping $\{0, 1\} \to \{-1, +1\}$. Noise variance is defined as

$$\sigma^2 = \frac{N_0}{2r}$$

where

$$\text{SNR}_{\text{dB}} = 10 \log_{10} \left(\frac{1}{N_0} \right).$$

and r denotes the code's rate.

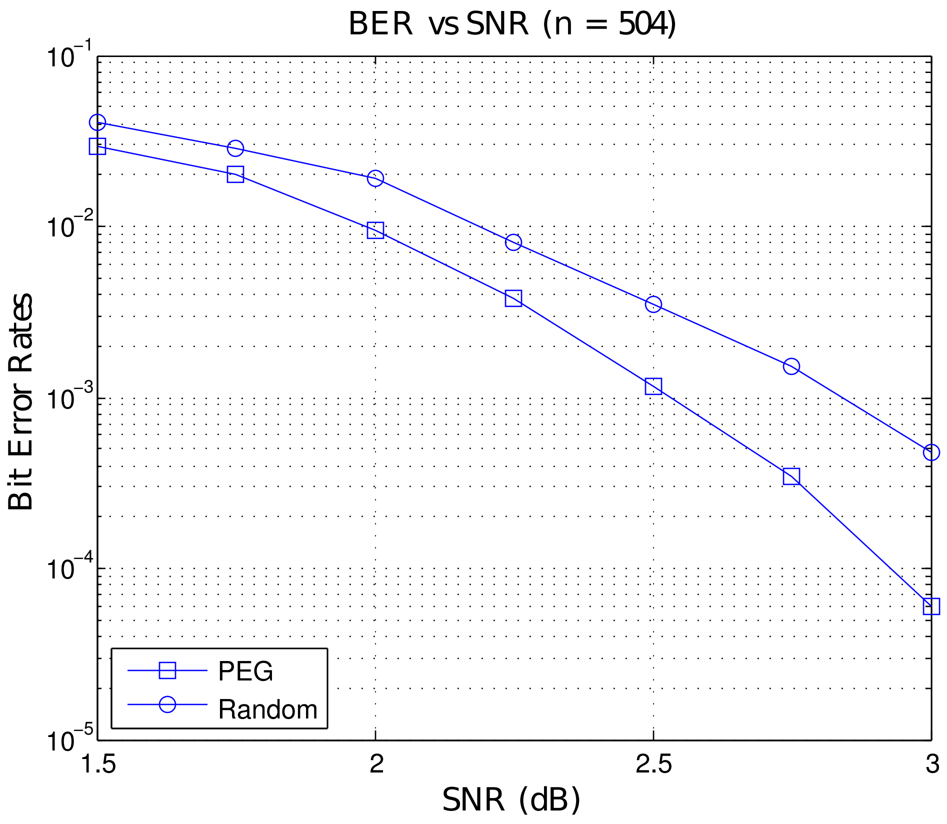


Figure 6.6: A random and a PEG created code, both with length $n = 504$ bits and rate 0.5.

It is clear that the PEG generated code has much better performance than the random code. More specifically, for SNR $\geq$ 2.5dB, the gain of PEG over the random construction is one order of magnitude. As the blocklength is increased, this gap will close since the assumption that the random construction creates cycle-free codes becomes valid with high probability. However, for practical applications where the blocklength can not be arbitrarily large, the PEG algorithm is able to create very good codes from any ensemble.

6.3.1 Linear Time Complexity Encoding (Modified PEG)

With a slight modification, this algorithm can produce codes which are in upper triangular form, possessing the very pleasant property of having a linear time encoding algorithm. This modification only affects the $(m-1)$ first variable nodes, since they correspond to the $m \times m$ submatrix of $\mathbf{H}$ which we want to be upper triangular, henceforth denoted $\mathbf{H}^p$. For the remaining variable nodes, the unmodified algorithm is used. The modification is quite simple, we only allow connections which result in "1"s over the diagonal line of $\mathbf{H}^p$ while making sure that the diagonal line has strictly non-zero elements, so that $\mathbf{H}^p$ is guaranteed to have full rank.

```
for $j = 0$ to $m - 1$ do
    for $k = 0$ to $d_{s_j} - 1$ do
        if $k = 0$ then
            $E^0_{s_j} \leftarrow$ edge $(c_j, s_j)$, where $E^0_{s_j}$ is the first edge incident to $s_j$.
            This edge corresponds to the "1" in the diagonal line of matrix $\mathbf{H}^p$.
        else
            Expand a subgraph from symbol node $s_j$ up to depth $l$ under the current graph setting such
            that the cardinality of $\mathcal{N}^l_{s_j} \cap \{c_0, c_1, \ldots, c_{j-1}\}$ stops increasing , or $\bar{\mathcal{N}}^l_{s_j} \cap \{c_0, c_1, \ldots, c_{j-1}\} \neq \emptyset$
            but $\bar{\mathcal{N}}^{l+1}_{s_j} \cap \{c_0, c_1, \ldots, c_{j-1}\} = \emptyset$, then $E^k_{s_j} \leftarrow$ edge $(c_i, s_j)$ where $E^k_{s_j}$ is the $k$-th edge incident
            to $s_j$ and $c_i$ is a check node picked from the set $\bar{\mathcal{N}}^l_{s_j} \cap \{c_0, c_1, \ldots, c_{j-1}\}$ having the lowest
            check-node degree.
        end if
    end for
end for
```

Figure 6.7: The Modified Progressive Edge Growth Algorithm

Note that the outer loop runs from 0 to $(m - 1)$. For iterations m to $(n - 1)$ we use algorithm 6.5, as already stated. Moreover, the degrees of the first $(m - 1)$ variable nodes may have to be modified, since variable node 1 can not have a degree higher than 1, variable node 2 can not have degree higher than 2, etc.

6.3.2 Further Modifications

A slight modification of the PEG algorithm was proposed in [10], where use of the Approximate Cycle Extrinsic (ACE) message degree metric is made to improve performance in the high SNR region, without any sacrifice in the low SNR region. More specifically, at each round where a cycle can not be avoided, the edge is placed in such a way as to increase the connectivity of the newly created cycle with the rest of the graph. The ACE metric is merely the sum of the current degrees of the variable nodes contained in the cycle. The idea behind this is that if a cycle can receive extrinsic information from many check nodes, it will not impose as much harm as it would if it was more isolated, since the available information will not be simply recycled at each iteration round, but a lot of new information will be available as well. Using this approach, the minimum weight of the resulting code is increased and trapping sets are improved, explaining the increase in performance in the high SNR region where these properties become of significance for the decoding operation.

Inspired by the fact that most of the optimised degree distributions found at [13] do not have regular check node degree distributions, we applied another slight modification to the PEG algorithm to allow for irregular check node degree distributions. The algorithm now assigns degrees to the check nodes as well. The design rates defined by the variable and check node degree distributions and the parameters m and n have to be compatible[2]. We resort to compatibility rather than equality, because degree distributions are usually given with a precision of at least 5 decimal digits, making equality comparison with a rational number rather meaningless. More precisely:

$$1 - \frac{\int_0^1 \rho(x)dx}{\int_0^1 \lambda(x)dx} \approx 1 - \frac{m}{n}$$

The check node degrees are then slightly modified, if needed, to make the total number of edges emanating from the check nodes equal to the total number of edges emanating from the variable nodes, slightly

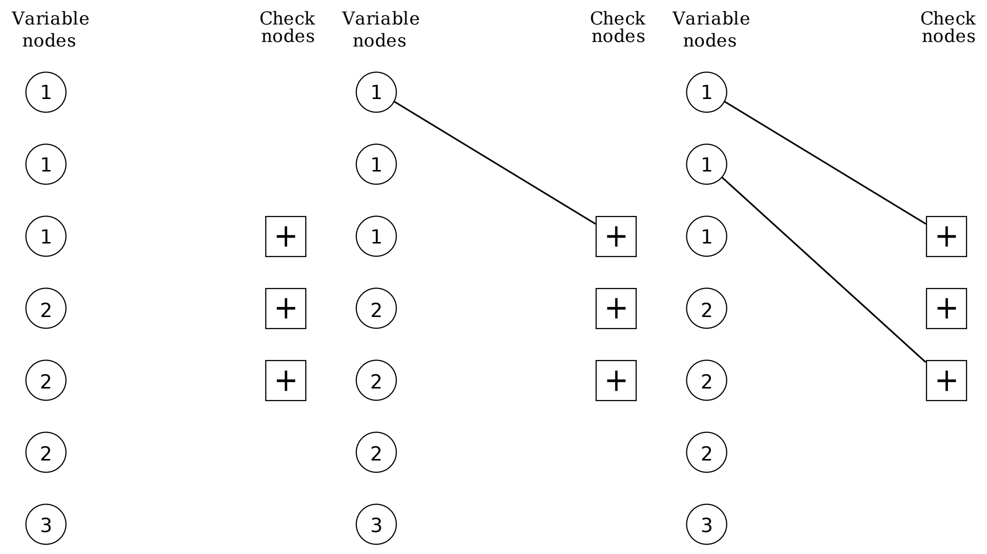
[2]For our implementation's purposes it suffices that they are within 0.01 of each other.

increasing the rate in the process. This increase, however, is negligible for codes with lengths of some thousands of bits. Source code for the modified algorithm along with some documentation can be found at `http://www.telecom.tuc.gr/~alex/`.

Example 6.6. We will call our trusted companion to arms once more, demonstrating an instance of the Modified Progressive Edge Growth algorithm for the $(7, \lambda_H, \rho_H)$ family of Hamming codes.

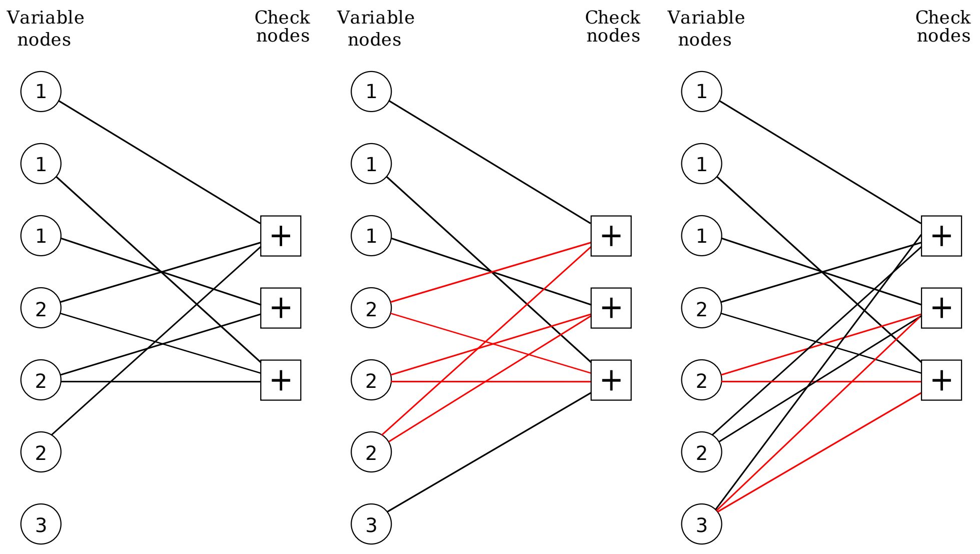
We have already seen that this code has 3 variable nodes of degree 1, 3 variable nodes of degree 2, and 1 check node of degree 3. The number of check nodes is 3. Thus, we create an empty graph with 7 variable nodes and 3 check nodes, and proceed as follows.

We start by assigning degrees to the variable nodes of the empty graph (a). Then, we can proceed to the first variable node, which is of degree 1. Since no connections have been made yet, and we also want the resulting parity-check matrix to be upper triangular, the sole choice we have is check node 1 (b). Moving on to variable node 2, we can choose between check nodes 1 and 2. We choose check node 2, since it has the lower degree (c). The connection of variable node 3 is made similarly. After reaching variable node 4, we can proceed with the unmodified Progressive Edge Growth algorithm, since the square part of our parity-check matrix is already completed and upper triangular. All connections of variable nodes 4 and 5, and the first connection of variable node 6 are straightforward: we always choose one of the check nodes with the lowest degree at every step of the algorithm, since there is no risk of creating cycles yet (d).



(a) (b) (c)

The next step is more interesting. We do not have many choices, since the only check node that has minimum degree in the given graph setting is check node 2, so we can not avoid the creation of a length 6 cycle. The first connection of variable node 7 can also be established as usual (e). In the final step, since either a connection to check node 1 or to check node 2 creates a cycle of length 4, a choice is made at random. For the final connection no selection has to be made, since we have only one choice. The final graph along with one of the many length 4 cycles of this graph is demonstrated in figure (f).



(d) (e) (f)

Note that this graph differs significantly from the graph in figure 2.2, despite the fact that they describe the same code. They both belong to the $(7, \lambda_H, \rho_H)$ ensemble of graphs. ■

6.3.3 Error-Minimisation Progressive Edge Growth (EMPEG)

A further improved version of the PEG algorithm was proposed in [20]. It is mainly aimed at the BEC, but also performs well on other channel types, such as the BI-AWGN channel. Instead of using ACE or other approximate metrics, it directly tries to optimise the code's error performance by exploiting the knowledge about a certain type of combinatorial structure, namely *minimal cycle sets*. Each edge placed by the EMPEG algorithm is chosen in such a way as to minimise the contribution to the expected block error probability. This contribution can be calculated by enumerating the minimal cycle sets that are formed if the edge in question is added to the graph. In order to continue, we first need to introduce some basic definitions.

Definition 6.7. An *extrinsic* check node to a set of variable nodes is a check node which is singly connected to the set.

Definition 6.8. An *intrinsic* check node to a set of variable nodes is a check node which is connected to the set at least twice.

Definition 6.9. The *Extrinsic Message Degree* (EMD) of a set of variable nodes is the number of extrinsic check nodes connected to the set. The EMD of a path in the graph is the EMD of the set of variable nodes contained in that path. Accordingly, the EMD of a cycle in the graph is the EMD of the set of variable nodes contained in that cycle.

Definition 6.10. A stopping set $\mathcal{S}$ is a subset of the set of variable nodes $\mathcal{V}$, such that no extrinsic check nodes are connected to $\mathcal{S}$.

That is, if all variable nodes in a stopping set happen to be erasures, their values can not be recovered. The probability that all variable nodes of a stopping set are erased is ϵ^n, where ϵ denotes the channel erasure probability and n denotes the cardinality of the set. Thus, in order to generate a good code, small stopping sets, which have high probability of full erasure, should be avoided.

Definition 6.11. A set of variable nodes forms a minimal cycle if 1) it forms a cycle, 2) no subset of the variable nodes forms a cycle.

Definition 6.12. A (d, ω, τ) cycle set is a set of the d variable nodes which forms a minimal cycle and has ω intrinsic check nodes and τ extrinsic check nodes.

Based on its definition, a cycle set has the following properties:

1. Every intrinsic check node of the set is connected to the set exactly twice.
2. Any variable node in the cycle set is connected to exactly two intrinsic check nodes.
3. The number of extrinsic check nodes of the cycle is equal to $\sum_{v_i}(d_{v_i} - 2)$ where d_{v_i} is the degree of the i-th variable node in the cycle set, i.e. for the induced cycle of a cycle set, EMD = ACE.
4. The number of intrinsic check nodes of a size d cycle set is always $\omega = d$. Thus, a (d, d, τ)-cycle set can be denoted as a (d, τ)-cycle set.

Definition 6.13. A set of variable nodes form a minimal cycle with respect to edge e if: 1) it forms a cycle that contains e, 2) no subset of the variable nodes forms a cycle that contains e.

Definition 6.14. A (d, ω, τ) e-cycle set is a set of d variable nodes, which forms a minimal cycle with respect to e and has ω intrinsic check nodes and τ extrinsic check nodes.

The properties of an e-cycle set are:

1. Every intrinsic check node of the set is connected to the set exactly twice.
2. Intrinsic checks are always connected between consecutive variable nodes in the minimal cycle formed by the e-cycle set.
3. The number of extrinsic check nodes of the cycle set is equal to $\sum_{v_i}(d_{v_i} - |S_i|)$ where v_i is the i-th variable node in the set, d_{v_i} is its degree and S_i is the set of check nodes that it shares with adjacent variable nodes in the minimal cycle formed by the set.
4. If we do not allow 4-cycles in the graph, we have only simple (d, d, τ) e-cycle sets.

Definition 6.15. A (d, τ) minimal e-cycle set is an e-cycle set, such that there is no other (d', τ') e-cycle set for which $(d'\tau') < (d, \tau)$.

Enumeration of e-cycle sets, which is vital to the EMPEG algorithm, has high complexity. Minimal e-cycle sets provide the highest contribution to the expected block error probability, so they are enumerated instead of enumerating all possible e-cycle sets. An efficient algorithm for the enumeration can be found at [20, Appendix A].

After minimal e-cycles have been enumerated, an estimate of their expected contribution to the block error probability has to be calculated. This can be done as follows. For ease of analysis, only regular check node degree distributions are considered. Let $P^{\epsilon}_{\text{CS}}(d, \omega, \tau)$ and $P^{\epsilon}_{\text{CS}}(d, \tau)$ denote the contribution to the expected block error probability of a (d, ω, τ) e-cycle set and a (d, τ) cycle set over a BEC(ϵ) respectively. We have:

$$P^{\epsilon}_{\text{CS}}(d, \omega, \tau) = \sum_{s=d}^{N} N_s(d, \omega, \tau)\epsilon^s$$

$$P^{\epsilon}_{\text{CS}}(d, \tau) = P^{\epsilon}_{\text{CS}}(d, d, \tau)$$

where $N_s(d,\omega,\tau)$ is the expected number of stopping sets of size s which contain the given (d,ω,τ)-cycle set. An equation for computing $N_s(d,\omega,\tau)$ is given in [20, Equation 4]. A recursive algorithm for pre-computing $P^{\epsilon}_{\mathrm{CS}}(d,\omega,\tau)$ for various values of (d,ω,τ) is also provided [20, Equation 10].

Let $W = \{\forall(d,\tau) : W_{(d,\tau)}\}$ denote the spectrum of cycle sets of a graph, where $W_{(d,\tau)}$ is the number of (d,τ)-cycle sets in the graph. We denote by LDPC(λ,ρ,W) the expurgated ensemble of the ensemble LDPC(λ,ρ) containing all graphs with cycle sets spectrum W. We say that LDPC(λ,ρ,W) is an *optimised* ensemble if

$$\forall(d,\tau) : N_{s|W}(d,d,\tau) \leq N_s(d,d,\tau).$$

An upper bound for the expected block error probability of optimised ensembles can be derived based on the observation that the support set of every erasure pattern leading to a block error includes a stopping set which includes a cycle set. Thus, if, for each cycle set we sum the probabilities of all the erasure patterns with support set including the cycle set, we take into account every error pattern at least once.

Theorem 6.16 ([20], Theorem 4.1)**.** *The expected block error probability over BE with erasure probability ϵ of the optimised ensemble LDPC(λ,ρ,W), with $\rho(x) = x^{d_c-1}$, is upper bounded by:*

$$\begin{aligned}
\mathbb{E}_{LDPC(\lambda,\rho,W)}[P_B(G,\epsilon)] &\leq \sum_{(d,\tau)} W_{(d,\tau)} \sum_{s=d}^{N} N_{s|W}(d,d,\tau)\epsilon^s \\
&\leq \sum_{(d,\tau)} W_{(d,\tau)} \sum_{s=d}^{N} N_s(d,d,\tau)\epsilon^s \\
&= \sum_{(d,\tau)} W_{(d,\tau)} P^{\epsilon}_{CS}(d,\tau).
\end{aligned}$$

The EMPEG algorithm is initialised with an unconnected graph G where variable and check nodes are assigned with edge sockets according to a degree distribution (λ,ρ). The edges are then connected as follows:

While there exists a variable node v_i with unconnected sockets, do:

1. For every check node c_j with an unconnected socket:
   a. Enumerate minimal e-cycle sets that will be formed if the edge $e = (v_i, c_j)$ is connected.
   b. Compute an estimate $P_B(G,e,\epsilon)$ of the expected contribution of $e = (v_i, c_j)$ to the block error probability of the code over a BEC with erasure probability ϵ.
2. Chose an edge, randomly, from the set of edges for which P_B is minimal and add it to the graph G.

Figure 6.8: The EMPEG algorithm.

Since the values for $P^{\epsilon}_{\mathrm{CS}}(d,\omega,\tau)$ are precomputed, the complexity of this algorithm is only slightly higher than that of the simple PEG algorithm.

6.4 Code Extension

The following techniques can be used for expanding the graph of an existing LDPC code in order to create a new code. A pleasant side-effect of this expansion is that many cycles of the corresponding Tanner graph are broken in the process, which is beneficial to the performance of the decoding process.

6.4.1 Column Splitting

Let **C** be an LDPC code of length n. We can extend this code by splitting each column of its parity-check matrix into $q \geq 2$ columns, where q is the extension factor. The resulting matrix will be even sparser than the original, so the resulting code will definitely be an LDPC code. If columns are split properly, the resulting codes can be very good. Column splitting increases the code's length and decreases its rate. Many codes of different rates and lengths can be constructed from a single *mother* code.

Example 6.17. Consider a rate-3/4 code $\mathcal{C}$ with the following parity-check matrix

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

We can apply column splitting by choosing to split each column into 2 new columns and distributing the 1s in a rotating manner. For example, the first column of $\mathcal{C}$ will be split into the following 2 columns

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}.$$

By extending the remaining columns, we get the following extended parity-check matrix

$$\mathbf{H}_{\text{ext}} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

of design rate $r = \frac{8}{12}$. The resulting code in this case is obviously not very good, since many variable nodes in $\mathbf{H}_{\text{ext}}$ are connected to only one check node. Note that many unfavourable constructs which result in length-4 cycles have been eliminated from the extended code. ■

6.4.2 Row Splitting

Let $\mathcal{C}$ be an LDPC code with an $m \times n$ parity-check matrix $\mathbf{H}$. We can extend this code by splitting each row of $\mathbf{H}$ into $q \geq 2$ rows, where q is the extension factor. The resulting matrix will be even sparser than the original, so the resulting code will definitely be an LDPC code. Row splitting increases the code's number of parity check equations, so the rate is decreased.

Example 6.18. Consider the same code as in the previous example. We can apply row splitting by choosing to split each row into 2 new rows and distributing the 1s in a rotating manner. For example, the first rows of $\mathcal{C}$ will be split into the following 2 rows

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

As in the previous example, if we form the overall extended parity-check matrix we will see that many of the unfavourable constructs have been eliminated. ■

By combining column and row splitting, we can create an extended code with the same rate as the original code but with many of its short cycles broken into longer ones. We can also choose to only extend a fraction of the columns or rows, in order to get an extension factor less than 2.

6.4.3 Breaking Cycles in Tanner Graphs

By column splitting, the degrees of the variable nodes in the extended code are reduced. Thus, the resulting graph has less cycles than the original graph. An example of a length-4 cycle being broken by column splitting can be seen below:

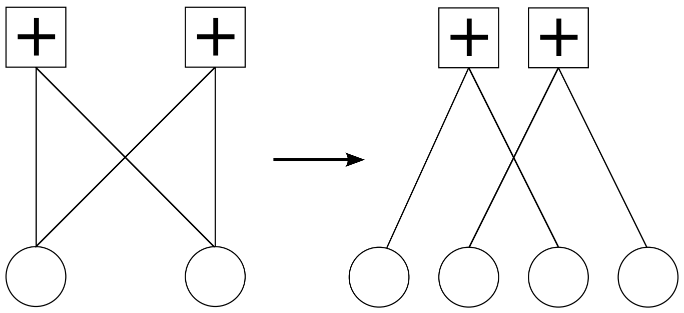


Figure 6.9: Breaking a length-4 cycle by column splitting.

The 2 original variable nodes have been split into 4 nodes, with each being connected to the 2 common check nodes by only one edge. Accordingly, by row splitting, the degrees of the check nodes in the extended code are reduced. Thus, the resulting graph has less cycles than the original graph. An example of a length-4 cycle being broken by row splitting can be seen below.

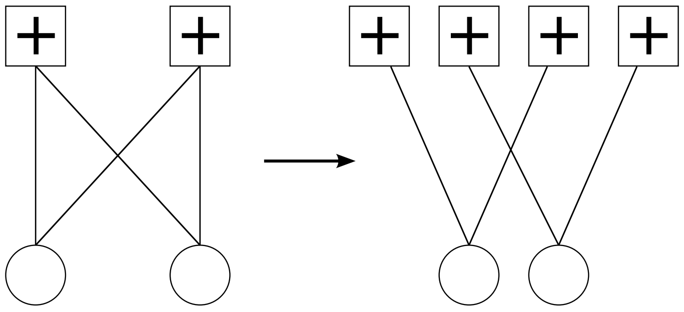


Figure 6.10: Breaking a length-4 cycle by row splitting.

The 2 original check nodes have been split into 4 nodes, with each being connected to the 2 common variable nodes by only one edge.

6.5 Concluding Remarks

The following table sums up the basic properties of each construction presented here.

Construction Properties				
	4-cycle Avoidance	Rate Guarantee	Quick Encoding	Irregularity
Gallager Construction				
Poisson Construction				✓
Random Construction	✓			✓
Quasi-Cyclic Codes			✓	✓
Array Codes	✓	✓		
Modified Array Codes	✓	✓	✓	
Finite Geometry Codes	✓		✓	
Graph Theoretic Codes	✓			
Progressive Edge Growth	✓	✓		✓
Modified PEG	✓	✓	✓	✓
Error-Minimization PEG	✓	✓	✓	✓

Table 6.1: Code Construction Properties

Naturally, not all aspects of the constructions can be represented in this table. For example, implementation complexity varies significantly, with the Gallager Construction probably having the lowest. Another aspect is hardware complexity of the encoder, in which Quasi-Cyclic and Array codes dominate. The best choice of construction depends on the particular scenario parameters, as usual in engineering. In our case, for example, we needed a way to construct irregular codes, since they perform better than regular ones. Poisson and random constructions were eliminated from our list of choices, since, in our opinion, the simplicity of their implementation does not balance their lack of useful properties which is depicted in the above table. Error-minimisation PEG applies mainly to the BEC channel, so the only remaining construction was the modified PEG algorithm, which we further modified, as already mentioned, to tailor it to our needs.

Chapter 7

Related Classes of Codes

7.1 Turbo Codes

Turbo codes were first introduced by Berrou et al. in 1993 [21]. They are nowadays competing with LDPC codes, which provide similar performance.

Turbo codes consist of two convolutional codes in parallel, with the input of the second being an interleaved version of the input of the first. Convolutional codes differ from block codes in the sense that they do not break the message stream into fixed-size blocks. Instead, redundancy is added continuously to the whole stream. The encoder keeps M previous input bits in memory. Each output bit of the encoder then depends on the current input bit as well as the M stored bits. Convolutional codes can also be though of as block codes, if we choose an appropriate way to terminate the data stream once no more information bits are present. The way of termination can greatly affect the code's performance, so it has to be chosen with great caution.

The generic structure of a Turbo encoder can be seen in the figure below.

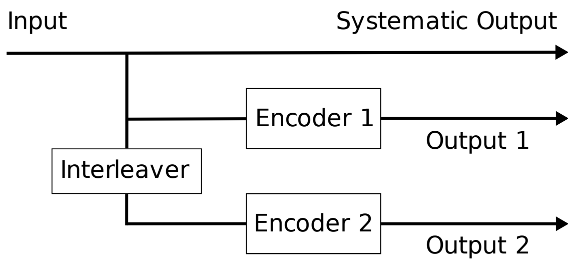


Figure 7.1: Generic Structure of a Turbo Encoder.

While the component codes can in practice be any type of code, convolutional codes are mostly used in accordance with [21]. Some further generic attributes of Turbo codes are:

1. The two parallel codes are usually identical.

2. The code is in systematic form.

3. Interleaving is done according to pseudo-random input to output mappings.

The structure of the interleaver is crucial to the performance of the code. Its purpose is mainly to help avoid low weight codewords. No universally best interleaver exists, the best choice is always dependent on the code design. Many interleaver structures have been proposed, ranging from relatively deterministic and with poor performance, to pseudorandom approaches with excellent performance.

Convolutional codes are usually decoded using the Viterbi algorithm. In the case of Turbo codes, the presence of the interleaver makes the resulting trellis diagram of the overall code extremely complex. Fortunately, decoding of Turbo codes can be achieved with several alternative approaches, which perform well in practice. Usually, two soft decision decoders are used which interact with one another in an iterative fashion, since both output streams contain the same parity bits, but in a different order. Popular choices for the soft-decision decoders include the Viterbi decoder and the BJCR algorithm. If we choose to view the Turbo code as a block code, iterative decoding using belief propagation can also be performed. A concatened code setup with a Turbo code as the inner code and an LDPC code as the outer code is briefly discussed in [11, Chapter 16].

As with LDPC codes, methods for asymptotic behaviour analysis of Turbo codes, namely EXIT charts and Density Evolution, exist. These methods can be used to search for good component codes as well as good interleavers for any given channel situation.

More information on Turbo codes can be found at [11, Chapter 16] and [1, Chapter 6], with the latter providing more detailed bibliographical guidance in the chapter's concluding section.

7.2 Fountain Codes

Fountain codes are record-breaking sparse-graph codes for channels with erasures, such as the Internet, where files are transmitted in multiple small packets, each of which is either received without error or not received. Standard file transfer protocols simply chop a file up into K packet-sized pieces, then repeatedly transmit each packet until it is successfully received (e.g. Selective Repeat ARQ). A back channel is required for the transmitter to find out which packets need retransmitting. In contrast, fountain codes make packets that are random functions of the whole file. The transmitter sprays packets at the receiver without any knowledge of which packets are received. Once the receiver has received any K' packets, where K' is just slightly greater than the original file size K, the whole file can be recovered.

An interesting observation is that in situations where very good channel coding has been applied, the overall channel, including the channel code, behaves much like an erasure channel. Most of the time, the channel code will work well and correct any errors. A small fraction of frames however might be erroneous even after decoding, so the decoder reports a failure. It is interesting to investigate whether we could apply some additional, simple, coding scheme to correct this type of failure using the rest of the frames which have been received correctly.

The existing class of erasure correcting codes, namely Reed-Solomon codes, have numerous disadvantages with the most prominent being high encoding and decoding complexity. A more elegant way, called a Fountain code, was introduced in [22], and further improved by the Luby Transform (LT) [23] and Raptor [24] coding approaches.

Consider a source which emits a metaphorical endless fountain of encoded data packets[1]. Fountain codes are *rateless* in the sense that as many coded packets as needed can be transmitted, without the need for a constant and predefined code rate. Suppose that the file we wish to send consists of K frames.

[1]Hence the name *Fountain* codes.

Then, as we will see, the receiver can, with high probability, correctly decode the K frames if he receives correctly *any* K' frames, with K' being just a fraction larger than K.

Fountain codes are very useful in situations of file transfers over networks, file storage on unreliable media, data broadcasting with many simultaneous receivers and more.

A slightly more detailed overview of Fountain codes than the one presented here can be found at [25].

7.2.1 The Random Linear Fountain

The first approach to Fountain codes was the Random Linear Fountain. Consider a file of size K packets $s_1, s_2, \ldots, s_K$. At each clock cycle, labelled by n, the encoder generates K random bits $\{G_{kn}\}$, and the transmitted packet t_n is set to the bitwise sum, modulo 2, of the source packets for which G_{kn} is 1. So

$$t_n = \sum_{k=1}^{K} s_k G_{kn}.$$

Each new set of K random bits can be thought of as a new column in an ever growing generator matrix $\mathbf{G}$.

Assume that the receiver holds out a metaphorical bucket and collects K' packets. Also assume that the generator matrix $\mathbf{G}$ corresponding to the K' received packets is known at the receiver. If $K' < K$ then the receiver has definitely not got enough information to decode the received packets. If $K = K'$ and the $K \times K$ matrix $\mathbf{G}$ is invertible, then the receiver can decode the packets as follows

$$s_k = \sum_{k=1}^{K} t_n G^{-1}_{nk}.$$

So the probability of decoding is equal to the probability of the $K \times K$ matrix $\mathbf{G}$ having full rank. This probability is the product of K probabilities, with each of them being the probability that a new column of $\mathbf{G}$ is linearly independent of the preceding columns. The first factor is $(1 - s^{-K})$, the probability that the first column of $\mathbf{G}$ is not the all-zero column. The second factor is $(1 - 2^{-(K-1)})$, the probability that the second column is equal neither to the all-zero column nor to the first column of $\mathbf{G}$, whichever this is. Iterating, the probability that $\mathbf{G}$ is invertible is

$$\left(1 - 2^{-K}\right)\left(1 - 2^{-(K-1)}\right)\ldots\left(1 - \frac{1}{2}\right) \approx 0.289 \text{ for any } K > 10.$$

This is not a great result, but it can be improved by letting $K' = K + E$ where E is a small number of excess packets (and columns in $\mathbf{G}$). We now have a $K \times K'$ matrix and are looking for any $K \times K$ invertible matrix within it. Let us call the probability that such a $K \times K$ invertible matrix exists $(1 - \delta)$, meaning that δ denotes the probability of failure. It is shown that, for any K, the probability of failure is bounded by:

$$\delta(E) \leq 2^{-E}.$$

This means that, in order to achieve a success rate of $1-\delta$, the number of packets required is approximately $K + \log_2 \frac{1}{\delta}$. This is good news, since it means that as the file size K increases, random Fountain codes can get arbitrarily close to the Shannon limit. On the other hand, the expected encoding cost is approximately $K/2$ operations per packet, which gives us a quadratic complexity over $K' = K + E$ packets. On the receiver side the situation is even worse, since inversion of a $K \times K$ matrix is needed, requiring approximately K^3 operations. While the performance of these codes can be very good, their encoding and decoding complexities can be prohibitively large for practical applications.

7.2.2 LT Codes

The LT code family, devised by Michael Luby in [23], retains the good performance of the random linear fountain code, while drastically reducing the encoding and decoding complexities.

Each encoded packet t_n is created from the source file $s_1, s_2, \ldots, s_K$ as follows:

1. Randomly choose the degree d_n of the packet from a degree distribution $\rho(d)$. The appropriate choice of ρ depends on the source file size K and will be discussed later.

2. Choose, uniformly at random, d_n distinct input packets and set t_n equal to the bitwise sum, modulo 2, of those d_n packets.

The introduction of a degree distribution and the fact that parity check equations are formed between the data packets, already strongly remind us of LDPC codes. In fact, the above encoding procedure defines a graph connecting encoded packets with data packets. If the mean degree d_{avg} is significantly smaller than K, then this graph is sparse. We already know that decoding of codes on sparse graphs can be done very efficiently via message-passing algorithms.

The message-passing algorithm is very simple in this case, since any received packet is either completely unknown if it is erased by channel, or known with absolute certainty if it is received correctly. For the following description of the decoding procedure, the encoded packets t_n are called check nodes.

```
repeat
    1. Find a check node $t_n$ that is connected to only one source packet $s_k$ (if no such packet exists,
    the algorithm halts here).
    2. Set $s_k = t_n$.
    3. Add $s_k$ to all checks $t_{n'}$ that are connected to $s_k$:
            $t_{n'} = t_{n'} \oplus s_k$ for all $n'$ such that $\mathbf{G}_{n'k} = 1$
    4. Remove all edges connected to $s_k$.
until {All $s_k$ have been determined.}
```

Figure 7.2: The LT Decoding Algorithm.

Designing the degree distribution is vital, since we wish to ensure that at each iteration at least one check node of degree 1 exists so the decoding algorithm can continue. This can be achieved by having many check nodes of small degree. On the other hand, this induces the risk that some source packets might not be connected to any check nodes, so some check nodes of large degree are definitely in order. However, these should be constrained to the absolute minimum needed, since encoding and decoding complexities increase linearly with the number of edges in the graph. For a given degree distribution $\rho(d)$, the statistics of the decoding process can be predicted by an appropriate version of the density evolution algorithm.

Through a random binning argument, it can be shown that the number of edges must be at least of order $K \log_e K$. This means that the encoding and decoding complexity of an LT code will be at least $K \log_e K$. It can also be shown that if enough packets have been received and decoding is possible, the average degree of each packet must be at least $\log_e K$. Ideally, we would like the graph to have only one check node of degree 1 at each iteration. This can be achieved, in expectation, by the *ideal soliton* *distribution*

$$\rho(1) = \frac{1}{K},$$
$$\rho(d) = \frac{1}{d(d-1)}, \quad \text{for } d = 2, 3, \ldots, K.$$

However, due to significant fluctuations around the average performance, this distribution provides poor performance in practice.

A slightly modified distribution, called the *robust soliton distribution*, can ensure that at each iteration the expected number of degree 1 check nodes is:

$$S = c \log_e \left(\frac{K}{\delta} \right) \sqrt{K}$$

rather than 1. The parameter δ is a bound on the probability that the decoding will fail after receiving K' packets, and c is a constant of order 1. We define the function:

$$\tau(d) = \begin{cases} \frac{s}{K}\frac{1}{d}, & d = 1, 2, \ldots, (K/S) - 1, \\ \frac{s}{K} \log_e(S/K), & d = K/S, \\ 0, & d > K/S. \end{cases}$$

We then add the above distribution to the ideal soliton distribution and normalise the result to get the robust soliton distribution $\mu(d)$:

$$\mu(d) = \frac{\rho(d) + \tau(d)}{Z}$$

where $Z = \sum_d [\rho(d) + \tau(d)]$. The number of encoded packets required at the receiving end to ensure that the decoding can run to completion, with probability at least $(1 - \delta)$, is $K' = KZ$.

In practice, LT codes can be tuned so that a file of original size $K \approx 10^4$ packets is recovered with an overhead of about 5%. It is characteristic of a good LT code that very little decoding is possible until slightly more than K packets have been received, at which point an avalanche of decoding occurs.

7.2.3 Raptor Codes

Raptor codes lower the encoding and decoding complexity even more by concatenating a weakened LT code with an outer code that patches the gaps in the LT code. They make use of a distribution with average degree $d_{\text{avg}} \approx 3$ which ensures that the decoding process will not get stuck, but a fraction of packets will not be connected to the graph, so they can not be recovered. Again, from the random binning argument, we can show that the expected fraction of non-recoverable frames is about:

$$\tilde{f} = e^{-d_{\text{avg}}}$$

which is approximately equal to 0.05 for $d_{\text{avg}} \approx 3$. If K is relatively large, the law of large numbers applies, assures us that the fraction of packets not recovered in any particular realisation will be very close to $\tilde{f}$.

Raptor codes use the following trick: a K-packet file is first precoded into $\tilde{K} \approx K/(1 - \tilde{f})$ packets with an excellent outer code that can correct erasures if the erasure rate is $\tilde{f}$ or less. We then transmit this slightly enlarged file using a weak LT code that, once slightly more than K packets have been received, can recover $(1 - \tilde{f})\tilde{K}$ of the precoded packets, which is roughly K packets. Then we use the outer code to recover the original file. In [24], an irregular LDPC code is used as the outer code, which has a linear time decoding algorithm, as we know.

Chapter 8

Relay Channel

8.1 Introduction

The relay channel was first introduced in 1971 by van der Meulen [26] and further studied from an information theoretic point of view in 1979 by Cover and El Gamal [27].

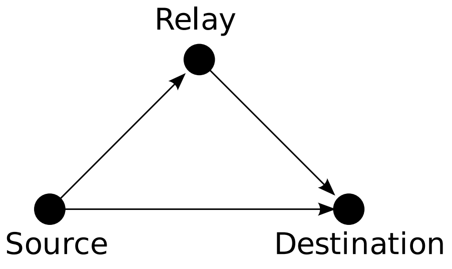


Figure 8.1: The Relay Channel.

The source wishes to send information to the destination and is aided by the relay, which has no information of its own to transmit. Several schemes of utilising this channel exist, each with a different capacity. Some examples include Amplify-and-Forward (AaF), where the relay simply acts as a repeater, Estimate-and-Forward (EaF) (or Demodulate-and-Forward, DEaF), where hard decisions are made at the relay and forwarded to the receiver, and Decode-and-Forward (DaF), where the relay decodes the source's codeword and uses it to send extra information to the destination in order to help it decode the source's codeword. The capacity of the general relay channel is still not known, however, Decode-and-Forward outperforms any other scheme proposed so far when the source-relay channel is strong.

In this chapter, we focus on the case of half-duplex Decode-and-Forward where we have two timeslots. In the first timeslot, called the *broadcast* timeslot (BC), the source transmits a codeword which is received by both the relay and the destination. However, only the relay can reliably decode the codeword. In the second timeslot, called the *multiple access* timeslot (MAC), the relay uses the information it got from the source in order to aid the destination with the decoding of the original codeword, while the source may transmit additional information utilising the capacity of the MAC channel formed by the source, the relay and the destination. The normalised durations of these timeslots are denoted t and $t' = (1-t)$, for the BC and MAC mode, respectively.

Let X, V, W and Y denote the source transmitted signal, the relay received signal, the relay transmitted signal and the destination received signal, respectively. Subscript 1 denotes the BC timeslot and subscript 2 denotes the MAC timeslot. Using these notational conventions, the half-duplex relay channel can be described as:

$$\begin{aligned} V_1 &= h_{\mathrm{SR}} X_1 + N_{R_1}, \\ Y_1 &= h_{\mathrm{SD}} X_1 + N_{D_1}, \\ Y_2 &= h_{\mathrm{SD}} X_2 + h_{\mathrm{RD}} W_2 + N_{D_2} \end{aligned}$$

where h_{SR} denotes the source-relay channel gain, h_{SD} denotes the source-destination channel gain and h_{RD} denotes the relay-destination channel gain. The rate achieved by Decode-and-Forward is [27]:

$$R_{\mathrm{DF}} = \sup_t \min\{tI(X_1; V_1) + t'I(X_2; Y_2|W_2), tI(X_1; Y_2) + t'I(X_2, W_2; Y_2)\}. \tag{8.1}$$

The optimal value for t can be found by equating the two arguments of $\min(\cdot)$.

8.2 A First Approach to Coding

We will now describe the coding scheme which was introduced in [28], and achieves rate (8.1), in detail. A total of N symbols are transmitted, of which tN are transmitted in BC mode, and the rest are sent in MAC mode. The information at the source is first divided into two independent parts (ω, v). In BC mode, the source encodes ω to generate a tN symbol long codeword $c_{\mathrm{SR}_1} \in \mathcal{C}_{\mathrm{SR}_1}$ with rate

$$R_{\mathrm{SR}_1} = I(X_1; V_1).$$

The codeword c_{SR_1}, corrupted by noise, is received by both the relay and the destination. The relay decodes c_{SR_1} reliably since R_{SR_1} equals the capacity of the SR link. The destination can not decode c_{SR_1} since the capacity of the SD link is lower than that of the SR link[1]. The destination stores c_{SR_1} for use at the end of MAC mode. Now, the destination already has $tNI(X_1; Y_1)$ bits of information in the form of the codeword c_{SR_1}. It needs an additional amount of $tN(I(X_1; V_1) - I(X_1; Y_1))$ bits to reliably decode c_{SR_1}. These additional bits are jointly sent by the source and the relay in a codeword $c_{\mathrm{RD}_2} \in \mathcal{C}_{\mathrm{RD}_2}$ of rate

$$R_{\mathrm{RD}_2} = \frac{t}{t'}(I(X_1; V_1) - I(X_1; Y_1)).$$

The second part of information v is also sent in MAC mode using a codeword $c_{\mathrm{SD}_2} \in \mathcal{C}_{\mathrm{SD}_2}$ to utilise the remaining capacity of the MAC channel. This information is sent only by the source, since the relay does not have access to it. Its rate is bounded by the capacity of the MAC channel and is thus equal to

$$R_{\mathrm{SD}_2} = \min\{I(X_2, W_2; Y_2) - \frac{t}{t'}(I(X_1; V_1) - I(X_1; Y_1)), I(X_2; Y_2|W_2)\}.$$

The decoding at the end of MAC mode proceeds as follows. First, c_{RD_2} and c_{SD_2} are decoded, using either successive or joint decoding, depending on the point we wish to achieve in the two user MAC channel capacity region. After c_{RD_2} and c_{SD_2} have been decoded, the destination can decode the corrupted codeword c_{SD_1} using the information in c_{RD_2} as side information. If all codes are binary codes, this side information is simply more parity information which lowers the rate of c_{SD_1} to $I(X_1; Y_1)$, permitting reliable decoding.

An interesting observation is that if we choose the correlation r to be equal either to 0 or to 1, then the rate loss, in comparison to a system that uses optimum correlation,[2] is negligible. Guided by this observation, the authors of [28] consider the code design problem only for $r = 1$ and $r = 0$.

[1]Otherwise relaying yields no gain over point to point transmission.

[2]Optimum correlation lies somewhere in $[0, 1]$.

Code design for $r = 1$.

In BC mode, the source S uses an LDPC code $\mathcal{C}_{\mathrm{SR}_1}$ with an $Nt \times Nt(1 - R_{\mathrm{SR}_1})$ parity check matrix to transmit information. The relay decodes this codeword. The destination D stores it for future decoding. In MAC mode, S and R use the BC mode codeword as the basis for cooperation. Both nodes multiply the BC codeword with an $Nt \times Nt(R_{\mathrm{SR}_1} - R_{\mathrm{SD}_1})$ matrix to generate $Nt(R_{\mathrm{SR}_1} - R_{\mathrm{SD}_1})$ These additional parity bits are then channel coded using an LDPC code $\mathcal{C}_{\mathrm{RD}_2}$ with an $Nt' \times Nt'(1 - R_{\mathrm{RD}_2})$ parity check matrix and transmitted from both S and R in a phase synchronised manner to the destination. The bits communicated by $\mathcal{C}_{\mathrm{RD}_2}$, in addition to the parity bits of the original code $\mathcal{C}_{\mathrm{SD}_1}$ that is decodable by D.

Code design for $r = 0$.

The BC mode remains unchanged, the two coding schemes differ only in MAC mode. In MAC mode, the source and the relay transmit independent information. The relay first generates additional parity bits from the BC mode codeword by multiplying with an $Nt \times Nt(R_{\mathrm{SR}_1} - R_{\mathrm{SD}_1})$ matrix (same as for $r = 1$). It then uses an LDPC code $\mathcal{C}_{\mathrm{RD}_2}$ with an $Nt' \times Nt'(1 - R_{\mathrm{RD}_2})$ parity check matrix to encode and transmit the additional parity information in MAC mode. The information carried by this codeword enables decoding of the BC mode codeword at the end of MAC mode. The source, in MAC mode, uses an LDPC code $\mathcal{C}_{\mathrm{SD}_2}$ with an $Nt' \times Nt'(1 - R_{\mathrm{SD}_2})$ parity check matrix to send new information to the destination. At the end of MAC mode, D uses successive decoding to recover both the additional parity bits and the new source information. Finally, the additional parity bits received in MAC mode are used to decode the received BC mode codeword at D.

The main challenge of this scheme lies in designing $\mathcal{C}_{\mathrm{SR}_1}$ and $\mathcal{C}_{\mathrm{SD}_1}$. More particularly, $\mathcal{C}_{\mathrm{SR}_1}$ is a subcode of $\mathcal{C}_{\mathrm{SD}_1}$, since $\mathcal{C}_{\mathrm{SD}_1}$ simply contains additional parity bits. Additionally, $\mathcal{C}_{\mathrm{SR}_1}$ has to be a capacity approaching code for the source-relay channel and $\mathcal{C}_{\mathrm{SD}_1}$ has to be a capacity approaching code for the source-destination channel. In [28] density evolution is used to optimise the code profiles, with some additional constraints which express the relation between the two code profiles. A more detailed description follows.

The graph of $\mathcal{C}_{\mathrm{SR}_1}$ has to be a subgraph of $\mathcal{C}_{\mathrm{SD}_1}$. Both $\mathcal{C}_{\mathrm{SR}_1}$ and $\mathcal{C}_{\mathrm{SD}_1}$ produce codewords of length tN symbols. Therefore, $\mathcal{C}_{\mathrm{SR}_1}$ has $tN(1 - R_{\mathrm{SR}_1})$ check nodes and $\mathcal{C}_{\mathrm{SD}_1}$ has $tN(1 - R_{\mathrm{SD}_1})$ check nodes that are a superset of those belonging to $\mathcal{C}_{\mathrm{SR}_1}$. As a result, the check degree distributions must satisfy a relationship given by the following theorem.

Theorem 8.1 ([28], Theorem 5.1). *If $\mathcal{C}_{SR_1}$ has a check node degree distribution $\rho^N_{SR_1}(x) = \sum_{i=2}^{d_{c_1}} \rho^N_{SR_{1,i}} x^{i-1}$, and $\mathcal{C}_{SD_1}$ has a check node degree distribution $\rho^N_{SD_1}(x) = \sum_{i=2}^{d_{c_2}} \rho^N_{SD_{1,i}} x^{i-1}$ then the following relationship must hold:*

$$\frac{(1 - R_{SR_1})}{(1 - R_{SD_1})} \rho^N_{SR_{1,i}} \leq \rho^N_{SD_{1,i}}, \quad \forall i = 2, 3, \dots, \max(d_{c_1}, d_{c_2}).$$

A corresponding relationship holds for the variable nodes as well and is stated in the following theorem.

Theorem 8.2 ([28], Theorem 5.2). *If $\mathcal{C}_{SR_1}$ has a variable node degree distribution $\lambda^N_{SR_1}(x) = \sum_{i=2}^{d_{v_1}} \lambda^N_{SR_{1,i}} x^{i-1}$, and $\mathcal{C}_{SD_1}$ has a variable node degree distribution $\lambda^N_{SD_1}(x) = \sum_{i=2}^{d_{v_2}} \lambda^N_{SD_{1,i}} x^{i-1}$ then the following relationship must hold:*

$$\sum_{i=j}^{\max(d_{v_1}, d_{v_2})} \lambda^N_{SR_{1,i}} \leq \sum_{i=j}^{\max(d_{v_1}, d_{v_2})} \lambda^N_{SD_{1,i}}, \quad \forall i = 2, 3, \dots, \max(d_{v_1}, d_{v_2}).$$

Another difference from the single-user case is that we now have two noise variances, $\sigma^2_{\mathrm{SD}_1}$ and $\sigma^2_{\mathrm{SR}_1}$. These variances, however, are related by the relative channel strengths

$$\sigma^2_{\mathrm{SD}_1} = \frac{\gamma_{\mathrm{SR}}}{\gamma_{\mathrm{SD}}}\sigma^2_{\mathrm{SR}_1}.$$

So, it suffices to treat one of the variances as a variable since the second can be completely determined by the first. These constraints, in addition to the usual constraints of density evolution, are used to search for good codes.

8.3 Bilayer Codes

In [28], classic density evolution was used even though the code structure is not that of a classic, single-user LDPC code. While this approach yielded relatively good results, the authors of [29] argued that the design can be improved by adapting the density evolution algorithm to this new code structure. A new class of LDPC codes, called *Bilayer* LDPC codes, emerged from this work.

The SNR at which the source-relay code must work is denoted SNR_+ and the SNR at which the source-destination code must work is denoted SNR_-. All channels are assumed to be Gaussian, so the capacities of the source-relay link and the source-destination link are

$$R_+ = \frac{1}{2}\log(1 + \mathrm{SNR}_+)$$

$$R_- = \frac{1}{2}\log(1 + \mathrm{SNR}_-)$$

respectively. The overall DaF rate can then be expressed as

$$R = \min(R_+, R_1 + R_-)$$

where R_1 denotes the relay-destination link capacity.

8.3.1 Expurgated Bilayer Codes

Let $\tilde{\mathcal{X}}$ be a linear $(n, n-k_1)$ LDPC code of rate $(n-k_1)/n$. The codebook $\tilde{\mathcal{X}}$ should be a capacity approaching code for the source-relay channel at SNR_+ with a rate R_+. Let k_2 be the number of extra random parity bits on the source codeword X_1, generated by the relay and provided to the destination. Then, a subcode of $\tilde{\mathcal{X}}$ which satisfies two sets of parities: k_1 zero parities enforced by the source's codebook and k_2 extra presumably nonzero parity bits provided by the relay, should form an $(n, n-k_1-k_2)$ capacity approaching code for decoding at the destination, i.e. at SNR_- with a rate R_-.

An ensemble of expurgated bilayer LDPC codes is defined as follows. The bilayer graph of the code consists of three sets of nodes and two sets of edges. The three sets of nodes correspond to one set of variable nodes, and two sets of check nodes: the k_1 lower check nodes corresponding to the check nodes in the lower subgraph and the k_2 upper check nodes corresponding to the check nodes in the upper subgraph. Edges are grouped in two sets: those connecting the variable nodes to the lower check nodes and those connecting the variable nodes to the upper check nodes.

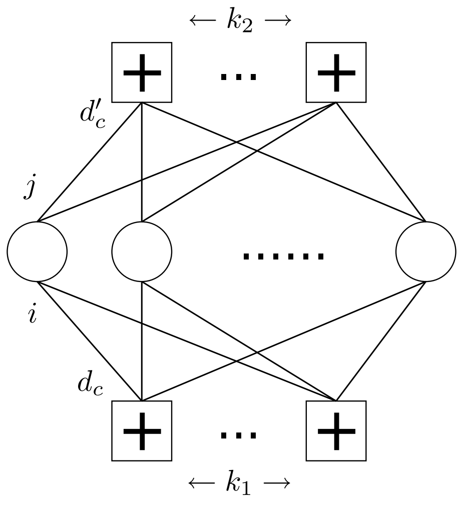


Figure 8.2: An expurgated bilayer LDPC code.

Definition 8.3. An edge is called a lower (upper) edge if it connects a variable node to a lower (upper) check node.

Definition 8.4. The lower (upper) degree of a variable node is defined as the number of lower (upper) edges connected to it.

Definition 8.5. The lower (upper) degree of an edge is defined as the lower (upper) degree of the variable node it is connected to.

The minimum lower variable degree is 2, since the lower code should be a valid LDPC code on its own. The minimum upper variable degree is 0, since some variable nodes may not be connected to an upper check degree.

Definition 8.6. A variable node has degree (i, j) if it has lower degree i and upper degree j.

Definition 8.7. An edge has degree (i, j) if it is connected to an (i, j) variable node.

Regular lower and upper check degrees are assumed in [29]. They are denoted d_c and d'_c, respectively. The ensemble of bilayer LDPC codes can be described by a variable node degree distribution

$$\lambda_{i,j}, \quad i \geq 2, j \geq 0$$

with

$$\sum_{i \geq 2, j \geq 0} \lambda_{i,j} = 1$$

and a parameter $0 < \eta < 1$, which defines the percentage of lower edges in the bilayer graph. Conventional density evolution has to be modified in order to correctly predict the performance of a bilayer code, since it is statistically different from a conventional LDPC code. In a bilayer code there is a distinction between messages exchanged between variable nodes and lower check nodes, and between variable nodes and upper check nodes. Therefore, evolution of two densities should be tracked. If we denote the density of the incoming messages at check nodes at the beginning of round ℓ as p^ℓ for lower check nodes and q^ℓ for upper check nodes, their updated versions, namely p'^ℓ and q'^ℓ, can be calculated using the conventional density evolution update rules. For variable nodes, the update rules for bilayer density evolution can be written as follows:

$$p_{i,j}^{\ell+1} = (*^{(i-1)} p'^\ell) * (*^j q'^\ell) * p_c, \quad i \geq 2, j \geq 0$$

$$q_{i,j}^{\ell+1} = (*^{i} p'^{\ell}) * (*^{(j-1)} q'^{\ell}) * p_c, \quad i \geq 2, j \geq 1$$

where p_c denotes the channel log-likelihood ratio. Then, the input densities to the lower and upper check nodes at the beginning of iteration $(\ell+1)$ can be computed as follows:

$$p^{(\ell+1)} = \sum_{i\geq 2, j\geq 0} \frac{i}{i+j} \lambda_{i,j} p_{i,j}^{(\ell+1)}$$

$$q^{(\ell+1)} = \sum_{i\geq 2, j\geq 0} \frac{j}{i+j} \lambda_{i,j} q_{i,j}^{(\ell+1)}.$$

Note that $\frac{i}{i+j}$ is the probability that a degree (i,j) edge is a lower edge and $\frac{j}{i+j}$ is the probability that a degree (i,j) edge is an upper edge.

The lower graph degree (i,j) error profile function $e_{i,j}^{1}(p^{\ell}, q^{\ell})$ is defined as the message error probability corresponding to the density $p_{i,j}^{(\ell+1)}$, after one density evolution iteration with input message densities p^{ℓ} and q^{ℓ}. Recall that this can be computed using $e(\mathcal{L})$ defined in chapter 5.2.1. Similarly, $e_{i,j}^{2}(p^{\ell}, q^{\ell})$ is defined as the message error probability corresponding to the density $q_{i,j}^{(\ell+1)}$, after one density evolution iteration with input message densities p^{ℓ} and q^{ℓ}. Let $e(p^{(\ell+1)}, q^{(\ell+1)})$ denote the overall message error probability in the bilayer graph corresponding to the message densities $p^{(\ell+1)}$ and $q^{(\ell+1)}$. Then:

$$e(p^{(\ell+1)}, q^{(\ell+1)}) = \sum_{i\geq 2, j\geq 0} \lambda_{i,j} \left(\frac{i}{i+j} e_{i,j}^{1}(p^{\ell}, q^{\ell}) + \frac{j}{i+j} e_{i,j}^{2}(p^{\ell}, q^{\ell}) \right).$$

Using these constraints, a rate maximisation problem can be formulated as follows. One approach is to fix the lower graph code to be a capacity approaching LDPC code at SNR$_+$ and searching for a variable degree distribution $\lambda_{i,j}$ that is consistent with the lower graph code and is capacity approaching at SNR$_-$. The check degrees d_c and d'_c are fixed, so the rate of the bilayer graph is related to the parameter η, since η depends on the number of check nodes in the graph via

$$\eta = \frac{d_c k_1}{d_c k_1 + d'_c k_2}.$$

By fixing the lower graph, we effectively fix n, k_1 and λ_i. Thus, the rate of the bilayer code, defined by $1-(k_1+k_2)/n$, can be maximized by minimising k_2 or equivalently maximising η. The distribution λ_i is related to $\lambda_{i,j}$ as follows:

$$\lambda_i = \frac{1}{\eta} \sum_{j\geq 0} \frac{i}{i+j} \lambda_{i,j}.$$

For fixed λ_i, the above equation can be written in a linear form:

$$\sum_{j\geq 0} \frac{i}{i+j} \lambda_{i,j} - \eta \lambda_i = 0.$$

The final linear program has the following form:

$$\begin{aligned}
\max\nolimits_{\lambda_{i,j},\eta} \quad & \eta \\
\text{s.t.} \quad & \textstyle\sum_{j\geq 0} \frac{i}{i+j} \lambda_{i,j} - \eta \lambda_i = 0 \\
& \textstyle\sum_{i\geq 2, j\geq 0} \lambda_{i,j} \left(\frac{i}{i+j} e_{i,j}^{1}(p^{\ell}, q^{\ell}) + \frac{j}{i+j} e_{i,j}^{2}(p^{\ell}, q^{\ell}) \right) < e(p^{\ell}, q^{\ell}) \quad \ell = 1, 2, \ldots, L \\
& \textstyle\sum_{i\geq 2, j\geq 0} \lambda_{i,j} = 1.
\end{aligned}$$

A good initialisation degree distribution can be found by solving the following linear program:

$$\begin{aligned} \min_{\lambda_{i,j},\eta} \quad & \eta \\ \text{s.t.} \quad & \textstyle\sum_{i\geq 2, j\geq 0} \lambda_{i,j} = 1 \\ & \textstyle\sum_{j\geq 0} \frac{i}{i+j}\lambda_{i,j} - \eta\lambda_i = 0 \end{aligned}$$

which will result in an initial code with maximum number of parity checks k_2, which ensures quick decoding convergence.

The optimal check degree for a conventional LDPC code is often concentrated around a mean value. Thus, this scheme results in good codes when the gap between SNR_+ and SNR_- is small, meaning that d_c and d'_c will be close to their optimal values. If, however, the gap between SNR_+ and SNR_- is large, the optimal check degree d'_c is much smaller than d_c, resulting in a larger gap to capacity. For these cases, a second family of bilayer LDPC codes, namely *lengthened* bilayer LDPC codes, are also introduced in [29].

8.3.2 Lengthened Bilayer Codes

Lengthening refers to the process of increasing the codeword length while keeping the number of parity checks constant. Designing a lengthened bilayer code corresponds to finding an overall graph so that the lower graph corresponds to a good LDPC code at R_- optimised for SNR_-, while the overall bilayer graph represents a good LDPC code at rate R_+ optimised for SNR_+. Many features of the lengthened bilayer LDPC code are the dual of the expurgated bilayer code: the roles of variable nodes check nodes are interchanged, the source encodes its data over the overall graph instead of the lower graph. Expurgated codes work well for small gaps between SNR_+ and SNR_+, while lengthened codes work best for larger gaps.

The source encodes its data over the overall graph. This is in contrast with expurgated codes where the source encodes its data over the lower graph first. The relay first decodes the source codeword over the bilayer graph. It then helps the destination by sending the values of upper n_2 variable nodes in the next block using the following scheme. The relay generates a set of $k_2 = (1 - R_-)n_2$ extra parity bits for the upper variable nodes, using the parity check matrix of a separate conventional LDPC code $\mathcal{C}_2$ of rate R_- optimised for SNR_-. The relay forwards these extra parity bits to the destination using another codebook of rate R_1. The destination first decodes the set of k_2 extra parity bits for upper n_2 variable bits provided by the relay. These k_2 parity bits are used to decode the upper n_2 variable nodes of the source codeword, which are then removed from the graph, reducing the resulting code's rate and thus allowing the destination to decode the remaining n_1 variable nodes. Decoding over the overall graph can also be performed, slightly increasing overall performance.

An ensemble of lengthened bilayer LDPC codes is defined as follows. The nodes are grouped into one set of check nodes and two sets of variable nodes: the lower variable nodes corresponding to the variable nodes in the lower subgraph, and the upper variable nodes corresponding to the variable nodes in the upper subgraph. The edges are grouped in two sets: those connecting check nodes to the lower variable nodes, and those connecting check nodes to the upper variable nodes.

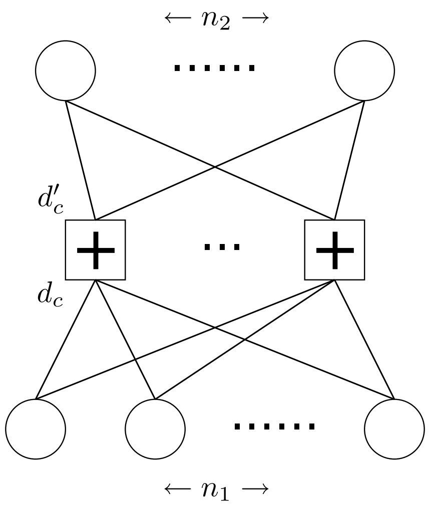


Figure 8.3: A lengthened bilayer LDPC code.

Definition 8.8. An edge is called a lower (upper) edge if it connects a check node to a lower (upper) variable node.

Assuming regular check degrees, each check node in the lengthened bilayer graph has d_c edges in the lower subgraph d'_c edges in the upper subgraph.

Definition 8.9. An edge is said to have a variable degree i if it is connected to a degree-i variable node.

The degrees of variable nodes are defined as with conventional LDPC codes. The ensemble of lengthened bilayer LDPC codes is defined by the lower variable node degree distribution, the upper variable node degree distribution and two regular check degrees d_c and d'_c.

Since upper and lower edges have different degree distributions, the messages passed along them have different densities. Let p^ℓ and q^ℓ denote the message densities in the lower and upper parts of the graph at the beginning of the ℓ-th decoding iteration. Let p'^ℓ and q'^ℓ denote the evolved versions of p^ℓ and q^ℓ after check updates. p'^ℓ and q'^ℓ can be computed as follows

$$p'^\ell = (\circledast^{(d_c-1)} p^\ell) \circledast (\circledast^{d'_c} q^\ell)$$

$$q'^\ell = (\circledast^{d_c} p^\ell) \circledast (\circledast^{(d'_c-1)} q^\ell).$$

Let $p_i^{(\ell+1)}$ ($q_i^{(\ell+1)}$) denote the output message density after a variable update at a variable node of degree i in the lower (upper) subgraph, with an input message density p'^ℓ (q'^ℓ). We have

$$p_i^{(\ell+1)} = *^{(i-1)} p'^\ell * p_c, \quad i \geq 2$$

$$q_i^{(\ell+1)} = *^{(i-1)} q'^\ell * p_c, \quad i \geq 2$$

where p_c is the channel message density. The message densities in the lower and upper subgraphs after the variable update, i.e. at the beginning of the $(\ell+1)$-th decoding iteration, can be computed as follows

$$p^{(\ell+1)} = \sum_{i \geq 2} \lambda_i^1 p_i^{(\ell+1)}$$

$$q^{(\ell+1)} = \sum_{i\geq 2} \lambda_i^2 q_i^{(\ell+1)}$$

where λ^1 and λ^2 are the lower and upper variable node degree distributions, respectively.

Let $e(p^{(\ell+1)}, q^{(\ell+1)})$ denote the message error probability of the message densities $p^{(\ell+1)}$ and $q^{(\ell+1)}$ at the beginning of the $(\ell+1)$-th decoding iteration. Let $e_i^1(p^\ell, q^\ell)$ $(e_i^2(p^\ell, q^\ell))$ denote the message error probability corresponding to $p_i^{(\ell+1)}$ $(q_i^{(\ell+1)})$, which is the message density of degree-i lower (upper) nodes after one density evolution iteration with input message densities p^ℓ and q^ℓ. The overall message error probability at the beginning of the $(\ell+1)$-th iteration, $e(p^{(\ell+1)}, q^{(\ell+1)})$, can be found as a linear combination of $e_i^1(p^\ell, q^\ell)$ and $e_i^2(p^\ell, q^\ell)$ functions as follows:

$$e(p^{(\ell+1)}, q^{(\ell+1)}) = \sum_{i\geq 2} \eta\lambda_i^1 e_i^1(p^\ell, q^\ell) + (1-\eta)\lambda_i^2 e_i^2(p^\ell, q^\ell)$$

where $\eta = \frac{d_c}{d_c + d_c'}$ denotes the percentage of lower edges in the graph.

The design of a lengthened bilayer LDPC code involves finding a pair of variable degree distributions λ^1 and λ^2 and a pair of check degrees d_c and d_c' for the lower and upper subgraphs in the bilayer structure, such that the overall graph is capacity approaching at SNR$_+$, while the lower graph is capacity approaching at SNR$_-$. The check degrees are again fixed, as is the lower variable degree distribution λ^1. The degree distribution λ^1 is found independently so that it is capacity approaching at SNR$_-$. The design problem is now reduced to finding an upper variable degree distribution λ^2 such that the overall lengthened graph represents a capacity approaching code at SNR$_-$. The rate of the overall lengthened code is $\frac{1-k}{n_1+n_2}$, where k denotes the number of check nodes, n_1 is the number of lower variable nodes, and n_2 is the number of upper variable nodes. The number of upper variable nodes n_2 is given by

$$n_2 = d_c' k \sum_{i\geq 2} \frac{\lambda_i^2}{i}.$$

Thus, fixing the lower graph code and d_c', the rate of the overall graph can be maximised by maximising the aforementioned sum. To ensure convergence of the overall code, we make use of the error profile function derived previously. More specifically, fixing η, d_c, and d_c', the linear programming update for λ^2 proposed in [29] can be formulated as follows:

$$\begin{array}{ll} \max_{\lambda_i^2} & \sum_{i\geq 2} \frac{\lambda_i^2}{i} \\ \text{s.t.} & \sum_{i\geq 2} \eta\lambda_i^1 e_i^1(p^\ell, q^\ell) + (1-\eta)\lambda_i^2 e_i^2(p^\ell, q^\ell) < e(p^\ell, q^\ell), \quad \ell = 1, 2, \ldots, L \\ & \sum_{i\geq 2} \lambda_i^2 = 1. \end{array}$$

As an initialisation, $\lambda_{\max(d_v)}^2 = 1$ is used. This code class covers the cases where the gap between SNR$_+$ and SNR$_-$ is large. Thus, expurgated and lengthened bilayer LDPC codes are complementary code structures which cover the entire range of SNRs.

8.3.3 Gaussian Approximation for Bilayer Codes

The main complexity of the overall optimisation scheme in both cases is due to density evolution. Hence, a Gaussian approximation was applied to bilayer density evolution in [30], for expurgated bilayer codes. The analysis is very similar to the analysis which was made for conventional density evolution. We denote the mean of message updates at the ℓ-th iteration at the input of lower (upper) check nodes as $\mu_v^{1,\ell}$ $(\mu_v^{2,\ell})$.

Likewise, we denote the mean of message updates at the ℓ-th iteration at the output of lower (upper) check nodes as $\mu_u^{1,\ell}$ ($\mu_u^{2,\ell}$). Using the message update rule at a degree (i,j) variable node, we have

$$\mu_{v,i,j}^{1,\ell} = \mu_{u_0} + (i-1)\mu_{u,i,j}^{1,(\ell-1)} + j\mu_{u,i,j}^{2,(\ell-1)}$$
$$\mu_{v,i,j}^{2,\ell} = \mu_{u_0} + i\mu_{u,i,j}^{1,(\ell-1)} + (j-1)\mu_{u,i,j}^{2,(\ell-1)}$$

where μ_{u_0} denotes the channel LLR mean. Using the same approach as with conventional density evolution, we get

$$\begin{aligned}\mu_v^{1,\ell} &= \phi\Big(\mu_{u_0} + (i-1)\phi^{-1}\Big(1-[1-\mu_v^{1,(\ell-1)}]^{(d_c-1)}\Big) \\ &\quad + j\phi^{-1}\Big(1-[1-\mu_v^{1,(\ell-1)}]^{(d_c'-1)}\Big)\Big) \\ &= g(\mu_v^{1,(\ell-1)}, \mu_v^{2,(\ell-1)})\end{aligned}$$

$$\begin{aligned}\mu_v^{2,\ell} &= \phi\Big(\mu_{u_0} + (j-1)\phi^{-1}\Big(1-[1-\mu_v^{2,(\ell-1)}]^{(d_c-1)}\Big) \\ &\quad + i\phi^{-1}\Big(1-[1-\mu_v^{1,(\ell-1)}]^{(d_c'-1)}\Big)\Big) \\ &= h(\mu_v^{1,(\ell-1)}, \mu_v^{2,(\ell-1)}).\end{aligned}$$

The overall mean of message updates at variable nodes can then be written as

$$\begin{aligned}\mu_v^{\ell} &= \sum_{i\geq 2, j\geq 0} \lambda_{i,j}\left(\frac{i}{i+j}\mu_v^{1,\ell} + \frac{j}{i+j}\mu_v^{2,\ell}\right) \\ &= \sum_{i\geq 2, j\geq 0} \lambda_{i,j}\left(\frac{i}{i+j} g(\mu_v^{1,(\ell-1)}, \mu_v^{2,(\ell-1)}) + \frac{j}{i+j} h(\mu_v^{1,(\ell-1)}, \mu_v^{2,(\ell-1)})\right).\end{aligned}$$

We can also calculate μ_v^ℓ using η as follows

$$\mu_v^\ell = \eta\mu_v^{1,\ell} + (1-\eta)\mu_v^{2,\ell}$$

resulting in the following expression for the convergence condition

$$\sum_{i\geq 2, j\geq 0} \lambda_{i,j}\left(\frac{i}{i+j} g(\mu_v^{1,(\ell)}, \mu_v^{2,(\ell)}) + \frac{j}{i+j} h(\mu_v^{1,(\ell)}, \mu_v^{2,(\ell)})\right) < \eta\mu_v^{1,\ell} + (1-\eta)\mu_v^{2,\ell}.$$

These constraints can now be used to express a linear optimisation problem, which can be solved using standard optimisation tools. More precisely, the authors of [30] chose to maximise the rate of the lower graph LDPC code with the following linear program.

$$\begin{aligned}\max_i \quad & \textstyle\sum_i \frac{\lambda_i^2}{i} \\ \text{s.t.} \quad & \textstyle\sum_{i\geq 2, j\geq 0} \lambda_{i,j}\left(\frac{i}{i+j} g(\mu_v^{1,(\ell)}, \mu_v^{2,(\ell)}) + \frac{j}{i+j} h(\mu_v^{1,(\ell)}, \mu_v^{2,(\ell)})\right) < \eta\mu_v^{1,\ell} + (1-\eta)\mu_v^{2,\ell}, \quad \ell = 1,2,\ldots,L \\ & \textstyle\sum_i \lambda_i \phi\left(\mu_{u_0} + (i-1)\phi^{-1}\left(1-[1-\mu_v^\ell]^{(d_c-1)}\right)\right) < \mu_v^\ell \\ & \textstyle\lambda_i - \frac{1}{\eta}\sum_{j\geq 0}\frac{i}{i+j}\lambda_{i,j} = 0.\end{aligned}$$

Naturally, the codes designed using the Gaussian approximation perform slightly worse than those designed using full density evolution. However, the complexity-performance trade-off is relatively favourable. A similar approach can be used for lengthened codes. However, in [30], they are not considered.

8.4 Related Work

In this section, some recent related work is presented. Recall that in the previous sections we always fixed the check node degrees to d_c and d'_c. A rather straightforward improvement was made in [31] where irregular check node degree distributions were allowed, slightly improving performance. In [32], rate-compatible LDPC codes are used in order to create two codes that can operate well at two different SNRs. A different approach was made in [33], where Multi-Edge type Bilayer-Expurgated codes are introduced. Furthermore, in [34] the relay channel is formulated as two virtual MISO and MIMO systems and a method of factor graph decoupling is used to design codes.

Chapter 9

Brief Review, Conclusion and Future Work

In this chapter, we will very briefly review the previous chapters and propose some future work.

9.1 Brief Review

We first introduced the notion of linear codes. Then, we introduced LDPC codes and we also learned that their parity-check matrices can be represented as graphs. Degree distributions were our next checkpoint, and the chapter was concluded with the definition of code ensembles.

In the following chapter, we considered the encoding problem for LDPC codes and showed that, when the parity-check matrix has a certain upper-triangular structure, encoding can be achieved with (almost) linear time complexity.

The next item on our list was decoding of LDPC codes. We started by stating the MAP decoding problem, which is NP-complete, and then introduced some suboptimal decoding algorithms which approximate the MAP decoder, such as Bit-Flipping decoding, One-Step Majority Logic decoding (along with their weighted variants) and Belief Propagation. The chapter was concluded by an extensive example which demonstrated all of the above decoding algorithms.

The problem of designing good LDPC codes was considered next. We introduced our main tool, the density evolution algorithm, accompanied by the Gaussian Approximation simplification which greatly reduces computational complexity. The chapter ended with a section explaining how density evolution is exploited in order to find degree distributions with good properties, along with a brief reference to EXIT charts.

In the sequel, we presented many different methods of constructing the parity-check matrix of an LDPC code with a given degree distribution, along with their pros and cons. Examples were also provided for most methods. A section was dedicated to the Progressive Edge Growth (PEG) algorithm and its variants, since it is a very good way of constructing short and medium lengthed LDPC codes, which can easily be used for practical applications.

Our discussion of LDPC codes was concluded in chapter 7, where some related code families, such as Turbo codes and Linear Fountain codes, were briefly described.

The final chapter consisted of applications of LDPC codes in the case of the relay channel. We started with the definition of the relay channel and then presented the rate achieved by the cooperative protocol Decode-and-Forward (DaF). We then focused on LDPC coding schemes which can approach this rate. Bilayer LDPC codes were introduced, along with bilayer density evolution and the corresponding bilayer Gaussian approximation.

9.2 Conclusion and Future Work

We saw that LDPC codes have a very strong theoretical background and come with very powerful design tools which can be modified appropriately to fit almost any scenario. They are one of two modern code classes which have proven to be capacity achieving with reasonable computational complexity. Consequently, they are a very good choice for applications ranging from secrecy and data storage up to satellite communications. This fact is also proven by the (largely incomplete) list of recent applications of LDPC codes which we presented in the first chapter.

Recently, great progress has been made in the area of coding for the relay channel. Of course, there is always room for improvement. All publications we were able to get our hands on, consider schemes with full correlation in MAC mode. However, by taking a close look at [28, Figure 5], we can see that, for SNRs greater than zero, the scheme with zero correlation can achieve higher rates; thus, we propose the exploration of this possibility. Zero correlation results in a MAC channel and the problem now consists of finding a good way of utilising it. We could jointly optimise the source and relay codes for the MAC channel. Alternatively, we could use Costa precoding at the source, since the interference from the relay is known, as it is a function of the BC codeword. Another interesting topic is the consideration of schemes with lower complexity than DaF, such as Estimate-and-Forward (EaF), in order to reduce the computational load of the relay.

In a slightly different direction, we also propose the modification of the Progressive Edge Growth algorithm to enable it to create bilayer LDPC codes for usage with the relay channel. In all related work, very long codes are considered (i.e. lengths greater than 10^5 bits), which are not suitable for practical applications since the decoding delay will be very large. By modifying the PEG algorithm, we will be able to construct moderately lengthed LDPC codes for the relay channel with good performance, which will be of great practical use.

Bibliography

[1] T. Richardson and R. Urbanke, *Modern Coding Theory.* Cambridge University Press, 2008. xi, 10, 15, 30, 37, 38, 40, 41, 42, 45, 47, 48, 50, 76

[2] C. E. Shannon, "A mathematical theory of communication," *Bell Systems Technical Journal*, vol. 27, pp. 379–423, 1948. 1

[3] R. Gallager, "Low-density parity-check codes," *IRE Transactions on Information Theory*, pp. 21–28, Jan. 1962. 1, 19, 57

[4] D. MacKay and R. Neal, "Good codes based on very sparse matrices," in *Lecture Notes in Computer Science*, pp. 100–111, Cryptography and Coding, 5th IMA Conference, Springer, 1995. 1

[5] T. Richardson and R. Urbanke, "The capacity of low-density parity-check codes under message-passing decoding," *IEEE Trans. Inf. Theory*, vol. 47, pp. 599–618, Feb. 2000. 1

[6] T. Richardson and R. Urbanke, "Design of capacity approaching irregular low-density parity-check codes," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 619–637, Feb. 2001. 1

[7] M. Luby, M. Mitzenmacher, A. Shokrollahi, and D. Spielman, "Analysis of low density codes and improved designs using irregular graphs," in *Proc. 30th Annu. ACM Symp. Theory of Computing*, pp. 249–258, 1998. 1, 6

[8] T. Richardson and R. Urbanke, "Efficient encoding of low-density parity-check codes," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 638–656, Feb. 2001. 1, 14

[9] X.-Y. Hu, E. Eleftheriou, and D. M. Arnold, "Regular and irregular progressive edge growth Tanner graphs," *IEEE Trans. Inf. Theory*, vol. 51, no. 1, pp. 386–398, Jan. 2005. 2, 63

[10] H. Xiao and A. H. Banihashemi, "Improved progressive edge growth (PEG) construction of irregular LDPC codes," *IEEE Communications Letters*, vol. 8, no. 12, pp. 715–717, Dec. 2004. 2, 66

[11] S. Lin and D. J. Costello Jr., *Error Control Coding.* Pearson Prentice Hall, 2nd ed., 2004. 4, 5, 62, 63, 76

[12] M. R. Tanner, "A recursive approach to low complexity codes," *IEEE Trans. Inf. Theory*, vol. 27, pp. 533–547, Sept. 1981. 6

[13] "LDPCopt - A fast and accurate degree distribution optimizer for LDPC code ensembles." `http://ipgdemos.epfl.ch/ldpcopt/`. 11, 54, 66

[14] Y. Kou, S. Lin, and M. P. C. Fossorier, "Low-density parity-check codes based on finite geometries: A rediscovery and new results," *IEEE Trans. Inf. Theory*, vol. 47, no. 7, pp. 2711–2736, Nov. 2001. 20, 21, 62

[15] V. D. Kolesnik, “Probability decoding of majority codes,” *Prob. Peredachi Inform.*, vol. 7, pp. 3–12, July 1971. 21

[16] S. Y. Chung, T. J. Richardson, and R. L. Urbanke, “Analysis of sum-product decoding of low-density parity-check codes using a Gaussian approximation,” *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 657–670, Feb. 2001. 50, 52, 53

[17] D. J. C. MacKay, S. T. Wilson, and M. C. Davey, “Comparison of constructions of irregular Gallager codes,” in *Proceedings of the 36th Allerton Conference on Communication, Control, and Computing, Sept. 1998*, pp. 220–229, Allerton House, 1998. 59

[18] S. Myung, K. Yang, and J. Kim, “Quasi-cyclic LDPC codes for fast encoding,” *IEEE Trans. Inf. Theory*, vol. 51, no. 8, pp. 2894–2901, Aug. 2005. 61

[19] J. Rosenthal and P. O. Vontobel, “Construction of regular and irregular LDPC codes using Ramamujan graphs and ideas from Margulis,” in *Proceedings of the IEEE International Symposium of Information Theory, Washington, D.C.*, p. 4, IEEE, June 2001. 63

[20] E. Sharon and S. Litsyn, “Constructing LDPC codes by error minimization progressive edge growth,” *IEEE Trans. Commun.*, vol. 56, no. 3, pp. 359–368, Mar. 2008. 68, 69, 70

[21] C. Berrou, A. Glavieux, and P. Thitimajshima, “Near shannon limit error-correcting coding and decoding: Turbo-codes(1),” in *Proceedings of the IEEE International Conference on Communications, Geneva, Switzerland*, pp. 1064–1070, IEEE, May 1993. 75

[22] J. W. Byers, M. Luby, A. Rege, and M. Mitzenmacher, “A digital fountain approach to reliable distribution of bulk data,” in *Proceedings of the ACM SIGCOMM’98, 2-4 Sept.*, ACM, 1998. 76

[23] M. Luby, “LT codes,” in *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, pp. 271–282, IEEE, Nov. 2002. 76, 78

[24] A. Shokrollahi, “Raptor codes.” Technical report, Laboratoire d’algorithmique, Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 2003. 76, 79

[25] D. J. MacKay, “Capacity approaching codes design and implementation - Fountain codes,” *IEE Proc.-Commun.*, vol. 152, no. 6, pp. 1060–1061, Dec. 2005. 77

[26] E. C. van der Meulen, “Three-terminal communication channels,” *Advanced Applied Probability*, vol. 3, pp. 120–154, 1971. 81

[27] T. M. Cover and A. A. E. Gamal, “Capacity theorems for the relay channel,” *IEEE Trans. Inf. Theory*, vol. 25, no. 5, pp. 572–584, Sept. 1979. 81, 82

[28] A. Chakrabarti, A. de Baynast, A. Sabharwal, and B. Aazhang, “Low density parity check codes for the relay channel,” *IEEE Journal Sel. Areas in Commun.*, vol. 25, no. 2, pp. 280–291, Feb. 2007. 82, 83, 84, 94

[29] P. Razaghi and W. Yu, “Bilayer low-density parity-check codes for decode-and-forward in relay channels,” *IEEE Trans. Inf. Theory*, vol. 53, no. 10, pp. 3723–3739, Oct. 2007. 84, 85, 87, 89

[30] J. P. Cances and V. Meghdadi, “Optimized low density parity check codes designs for half duplex relay channels,” *IEEE Trans. on Wireless Comm.*, vol. 8, no. 7, pp. 3390–3395, July 2009. 89, 90

[31] M. H. Azmi, J. Yuan, J. Ning, and H. Q. Huynh, “Improved bilayer LDPC codes using irregular check node degree distribution,” in *Proceedings of the IEEE International Symposium of Information Theory, Toronto, Canada*, pp. 141–145, IEEE, July 6–11 2008. 91

[32] C. Li, G. Yue, X. Wang, and M. A. Khojastepour, “LDPC code design for half-duplex cooperative relay,” *IEEE Trans. on Wireless Comm.*, vol. 7, no. 11, pp. 4558–4567, Nov. 2008. 91

[33] M. H. Azmi and J. Yuan, “Design of multi-edge type bilayer-expurgated LDPC codes,” in *Proceedings of the IEEE International Symposium of Information Theory, Seoul, Korea*, pp. 1988–1992, IEEE, June 28 - July 3 2009. 91

[34] C. Li, G. Yue, M. A. Khojastepour, X. Wang, and M. Madihian, “LDPC-coded cooperative relay systems: Performance analysis and code design,” *IEEE Trans. Commun.*, vol. 56, no. 3, pp. 485–496, Mar. 2008. 91