Introducing Low-Density Parity-Check Codes

Sarah J. Johnson
School of Electrical Engineering and Computer Science
The University of Newcastle
Australia
email: sarah.johnson@newcastle.edu.au

Topic 1: Low-Density Parity-Check Codes

1.1 Introduction

Low-density parity-check (LDPC) codes are forward error-correction codes, first proposed in the 1962 PhD thesis of Gallager at MIT. At the time, their incredible potential remained undiscovered due to the computational demands of simulation in an era when vacumm tubes were only just being replaced by the first transistors. They remained largely neglected for over 35 years. In the mean time the field of forward error correction was dominated by highly structured algebraic block and convolutional codes. Despite the enormous practical success of these codes, their performance fell well short of the theoretically achievable limits set down by Shannon in his seminal 1948 paper. By the late 1980s, despite decades of attempts, researchers were largely resigned to this seemingly insurmountable theory–practice gap.

The relative quiescence of the coding field was utterly transformed by the introduction of "turbo codes," proposed by Berrou, Glavieux and Thitimajshima in 1993, wherein all the key ingredients of successful error correction codes were replaced: turbo codes involve very little algebra, employ iterative, distributed algorithms, focus on average (rather than worst-case) performance, and rely on soft (or probabilistic) information extracted from the channel. Overnight, the gap to the Shannon limit was all but eliminated, using decoders with manageable complexity.

As researchers struggled through the 1990s to understand just why turbo codes worked as well as they did, two researchers, McKay and Neal, introduced a new class of block codes designed to posses many of the features of the new turbo codes. It was soon recognized that these block codes were in fact a rediscovery of the LDPC codes developed years earlier by Gallager. Indeed, the algorithm used to decode turbo codes was subsequently shown to be a special case of the decoding algorithm for LDPC codes presented by Gallager so many years before.

New generalizations of Gallager's LDPC codes by a number of researchers including Luby, Mitzenmacher, Shokrollahi, Spielman, Richardson and Urbanke, produced new irregular LDPC codes which easily outperform the best turbo codes, as well as offering certain practical advantages and an arguably cleaner setup for theoretical results. Today, design techniques for LDPC codes exist which enable the construction of codes which approach the Shannon's capacity to within hundredths of a decibel.

So rapid has progress been in this area that coding theory today is in many ways unrecognizable from its state just a decade ago. In addition to the strong

theoretical interest in LDPC codes, such codes have already been adopted in satellite-based digital video broadcasting and long-haul optical communication standards, are highly likely to be adopted in the IEEE wireless local area network standard, and are under consideration for the long-term evolution of third-generation mobile telephony.

1.2 Error correction using parity-checks

Here we will only consider binary messages and so the transmitted messages consist of strings of 0's and 1's. The essential idea of forward error control coding is to augment these *message* bits with deliberately introduced redundancy in the form of extra *check* bits to produce a *codeword* for the message. These check bits are added in such a way that codewords are sufficiently distinct from one another that the transmitted message can be correctly inferred at the receiver, even when some bits in the codeword are corrupted during transmission over the channel.

The simplest possible coding scheme is the single parity check code (SPC). The SPC involves the addition of a single extra bit to the binary message, the value of which depends on the bits in the message. In an even parity code, the additional bit added to each message ensures an even number of 1s in every codeword.

Example 1.1.

The 7-bit ASCII string for the letter *S* is 1010011, and a parity bit is to be added as the eighth bit. The string for *S* already has an even number of ones (namely four) and so the value of the parity bit is 0, and the codeword for *S* is 10100110.

More formally, for the 7-bit ASCII plus even parity code we define a codeword c to have the following structure:

$$\mathbf{c} = [c_1\ c_2\ c_3\ c_4\ c_5\ c_6\ c_7\ c_8],$$

where each c_i is either 0 or 1, and every codeword satisfies the constraint

$$c_1 \oplus c_2 \oplus c_3 \oplus c_4 \oplus c_5 \oplus c_6 \oplus c_7 \oplus c_8 = 0. \tag{1.1}$$

Equation (1.1) is called a *parity-check equation*, in which the symbol $\oplus$ represents modulo-2 addition.

Example 1.2.

A 7-bit ASCII letter is encoded with the single parity check code from Example 1.1. The resulting codeword was sent though a noisy channel and the string $\mathbf{y} = [1\ 0\ 0\ 1\ 0\ 0\ 1\ 0]$ was received. To check if $\mathbf{y}$ is a valid codeword we test $\mathbf{y}$ with (1.1).

$$y_1 \oplus y_2 \oplus y_3 \oplus y_4 \oplus y_5 \oplus y_6 \oplus y_7 \oplus y_8 = 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 = 1.$$

Since the sum is 1, the parity-check equation is not satisfied and $\mathbf{y}$ is not a valid codeword. We have detected that at least one error occurred during the transmission.

While the inversion of a single bit due to channel noise can easily be detected with a single parity check code, this code is not sufficiently powerful to indicate which bit, or indeed bits, were inverted. Moreover, since any even number of bit inversions produces a string satisfying the constraint (1.1), patterns of even numbers of errors go undetected by this simple code. Detecting more than a single bit error calls for increased redundancy in the form of additional parity bits and more sophisticated codes contain multiple parity-check equations and each codeword must satisfy every one of them.

Example 1.3.

A code C consists of all length six strings

$$\mathbf{c} = [c_1\ c_2\ c_3\ c_4\ c_5\ c_6],$$

which satisfy all three parity-check equations:

$$\begin{aligned} c_1 \oplus c_2 \oplus c_4 &= 0 \\ c_2 \oplus c_3 \oplus c_5 &= 0 \\ c_1 \oplus c_2 \oplus c_3 \oplus c_6 &= 0 \end{aligned} \tag{1.2}$$

Codeword constraints are often written in matrix form and so the constraints of (1.2) become

$$\underbrace{\begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}}_{H} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}. \tag{1.3}$$

The matrix H is called a *parity-check matrix*. Each row of H corresponds to a parity-check equation and each column of H corresponds to a bit in the codeword. Thus for a binary code with m parity-check constraints and length n codewords the parity-check matrix is an $m \times n$ binary matrix. In matrix form a string $\mathbf{y} = [c_1\ c_2\ c_3\ c_4\ c_5\ c_6]$ is a valid codeword for the code with parity-check matrix H if and only if it satisfies the matrix equation

$$H\mathbf{y}^T = 0. \tag{1.4}$$

1.2.1 Encoding

To distinguish between the message bits and parity bits in the codeword in Example 1.3 we re-write the code parity-check constraints so that each one solves for a different codeword bit.

Example 1.4.

The code constraints from Example 1.3 can be re-written as

$$\begin{aligned} c_4 &= c_1 \oplus c_2 \\ c_5 &= c_2 \oplus c_3 \\ c_6 &= c_1 \oplus c_2 \oplus c_3 \end{aligned} \tag{1.5}$$

The codeword bits c_1, c_2, and c_3 contain the three bit message, c_1, c_2, and c_3, while the codeword bits c_4, c_5 and c_6 contain the three parity-check bits. Written this way the codeword constraints show how to encode the message.

Example 1.5.

Using the constraints in (1.5) the message 110 produces the parity-check bits

$$\begin{aligned} c_4 &= 1 \oplus 1 = 0, \\ c_5 &= 1 \oplus 0 = 1, \\ c_6 &= 1 \oplus 1 \oplus 0 = 0, \end{aligned}$$

and so the codeword for this message is $\mathbf{c} = [1\ 1\ 0\ 0\ 1\ 0]$.

Again these constraints can be written in matrix form as follows:

$$\left[\ c_1\ c_2\ c_3\ c_4\ c_5\ c_6\ \right] = \left[\ c_1\ c_2\ c_3\ \right] \underbrace{\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}}_{\mathbf{G}}, \tag{1.6}$$

where the matrix G is called the *generator matrix* of the code. The message bits are conventionally labeled by $\mathbf{u} = [u_1, u_2, \cdots u_k]$, where the vector $\mathbf{u}$ holds the k message bits. Thus the codeword $\mathbf{c}$ corresponding to the binary message $\mathbf{u} = [u_1 u_2 u_3]$ can be found using the matrix equation

$$\mathbf{c} = \mathbf{u}G. \tag{1.7}$$

For a binary code with k message bits and length n codewords the generator matrix, G, is a $k \times n$ binary matrix. The ratio k/n is called the *rate* of the code.

A code with k message bits contains 2^k codewords. These codewords are a subset of the total possible 2^n binary vectors of length n.

Example 1.6.

Substituting each of the $2^3 = 8$ distinct messages $c_1\ c_2\ c_3 = 000, 001, \ldots, 111$ into equation (1.7) yields the following set of codewords for the code from Example 1.3:

$$\begin{array}{cccc} [0\,0\,0\,0\,0\,0] & [0\,0\,1\,0\,1\,1] & [0\,1\,0\,1\,1\,1] & [0\,1\,1\,1\,0\,0] \\ [1\,0\,0\,1\,0\,1] & [1\,0\,1\,1\,1\,0] & [1\,1\,0\,0\,1\,0] & [1\,1\,1\,0\,0\,1] \end{array} \tag{1.8}$$

This code is called *systematic* because the first k codeword bits contain the message bits. For systematic codes the generator matrix contains the $k \times k$ identity, I_k, matrix as its first k columns. (The identity matrix, I_k, is a $k \times k$ square binary matrix with '1' entries on the diagonal from the top left corner to the bottom right corner and '0' entries everywhere else.)

A generator matrix for a code with parity-check matrix H can be found by performing Gauss-Jordan elimination on H to obtain it in the form

$$H = [A, I_{n-k}], \tag{1.9}$$

where A is an $(n-k) \times k$ binary matrix and I_{n-k} is the identity matrix of order $n-k$. The generator matrix is then

$$G = [I_k, A^T]. \tag{1.10}$$

The row space of G is orthogonal to H. Thus if G is the generator matrix for a code with parity-check matrix H then

$$GH^T = 0.$$

Before concluding this section we note that a block code can be described by more than one set of parity-check constraints. A set of constraints is valid for a code provided that equation (1.4) holds for all of the codewords in the code. For low-density parity-check codes the choice of parity-check matrix is particularly important.

Example 1.7.

The code C in Example 1.3 can also be described by four parity-check equations:

$$\begin{aligned} c_1 \oplus c_2 \oplus c_4 &= 0 \\ c_2 \oplus c_3 \oplus c_5 &= 0 \\ c_1 \oplus c_2 \oplus c_3 \oplus c_6 &= 0 \\ c_3 \oplus c_4 \oplus c_6 &= 0 \end{aligned} \tag{1.11}$$

The extra constraint in Example 1.7 is the linear combination of the 1-st and 3-rd parity-check equations in and so the new equation is said to be *linearly dependent* on the existing parity-check equations. In general, a code can have any number of parity-check constraints but only $n-k$ of them will be linearly independent, where k is the number of message bits in each codeword. In matrix notation $n-k$ is the rank of H

$$n - k = \text{rank}_2(H), \tag{1.12}$$

where $\text{rank}_2(H)$ is the number of rows in H which are linearly dependent over GF(2).

1.2.2 Error detection and correction

Suppose a codeword has been sent down a binary symmetric channel and one or more of the codeword bits may have been flipped. The task, outline in this section and the following, is to detect any flipped bits and, if possible, to correct them.

Firstly, we know that every codeword in the code must satisfy (1.4), and so errors can be detected in any received word which does not satisfy this equation.

Example 1.8.

The codeword $\mathbf{c} = [1\ 0\ 1\ 1\ 1\ 0]$ from the code in Example 1.3 was sent through a binary symmetric channel and the the string $\mathbf{y} = [1\ 0\ 1\ 0\ 1\ 0]$ received. Substitution into equation (1.4) gives

$$H\mathbf{y}^T = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}. \tag{1.13}$$

The result is nonzero and so the string $\mathbf{y}$ is not a codeword of this code. We therefore conclude that bit flipping errors must have occurred during transmission.

The vector

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

is called the syndrome of $\mathbf{y}$. The syndrome indicates which parity-check constraints are not satisfied by $\mathbf{y}$.

Example 1.9.

The result of Equation 1.13, in Example 1.8, (i.e. the syndrome) indicates that the first parity-check equation in H is not satisfied by $\mathbf{y}$. Since this parity-check equation involves the 1-st, 2-nd and 4-th codeword bits we can conclude that at least one of these three bits has been inverted by the channel.

Example 1.8 demonstrates the use of a block code to detect transmission errors, but suppose that the channel was even noisier and three bits were flipped to produce the string $\mathbf{y} = [0\ 0\ 1\ 0\ 1\ 1]$. Substitution into (1.4) tells us that $\mathbf{y}$ is a valid codeword and so we cannot detect the transmission errors that have occurred. In general, a block code can only detect a set of bit errors if the errors don't change one codeword into another.

The *Hamming distance* between two codewords is defined as the number of bit positions in which they differ. For example the codewords $[1\ 0\ 1\ 0\ 0\ 1\ 1\ 0]$ and $[1\ 0\ 0\ 0\ 0\ 1\ 1\ 1]$ differ in two positions, the third and eight codeword bits, so the Hamming distance between them is two. The measure of the ability of a code to detect errors is the *minimum Hamming distance* or just *minimum distance* of the code. The minimum distance of a code, $d_{\min}$, is defined as the smallest Hamming distance between any pair of codewords in the code. For the code in Example 1.3, $d_{\min} = 3$, so the corruption of three or more bits in a codeword can result in another valid codeword. A code with minimum distance $d_{\min}$, can always detect t errors whenever

$$t < d_{\min}. \tag{1.14}$$

To go further and correct the bit flipping errors requires that the decoder determine which codeword was most likely to have been sent. Based only on

knowing the binary received string, $\mathbf{y}$, the best decoder will choose the codeword closest in Hamming distance to $\mathbf{y}$. When there is more than one codeword at the minimum distance from $\mathbf{y}$ the decoder will randomly choose one of them. This decoder is called the *maximum likelihood* (ML) decoder as it will always chose the codeword which is most likely to have produced $\mathbf{y}$.

Example 1.10.

In Example 1.8 we detected that the received string $\mathbf{y} = [1\ 0\ 1\ 0\ 1\ 0]$ was not a codeword of the code in Example 1.3. By comparing $\mathbf{y}$ with each of the codewords in this code, (1.8), the ML decoder will choose $\mathbf{c} = [1\ 0\ 1\ 1\ 1\ 0]$, as the closest codeword as it is the only codeword Hamming distance 1 from $\mathbf{y}$.

The minimum distance of the code in Example 1.8 is 3, so a single bit flipped always results in a string $\mathbf{y}$ closer to the codeword which was sent than any other codeword and thus can always be corrected by the ML decoder. However, if two bits are flipped in $\mathbf{y}$ there may be a different codeword which is closer to $\mathbf{y}$ than the one which was sent, in which case the decoder will choose an incorrect codeword.

Example 1.11.

The codeword $\mathbf{c} = [1\ 0\ 1\ 1\ 1\ 0]$ from the code in Example 1.3 was transmitted through a channel which introduced two flipped bits producing the string $\mathbf{y} = [0\ 0\ 1\ 0\ 1\ 0]$. By comparison of $\mathbf{y}$ with each of the codewords of this code, (1.8), the ML decoder will choose $\mathbf{c} = [0\ 0\ 1\ 0\ 1\ 1]$ as the closest decoder as it is Hamming distance one from $\mathbf{y}$. In this case the ML decoder has actually added errors rather than corrected them.

In general, for a code with minimum distance $d_{\min}$, e bit flips can always be corrected by choosing the closest codeword whenever

$$e \leq \lfloor (d_{\min} - 1)/2 \rfloor, \tag{1.15}$$

where $\lfloor x \rfloor$ is the largest integer that is at most x.

The smaller the code rate the smaller the subset of 2^n binary vectors which are codewords and so the better the minimum distance that can be achieved by a code with length n. The importance of the code minimum distance in determining its performance is reflected in the description of block codes by the three parameters $(n, k, d_{\min})$.

Error correction by directly comparing the received string to every other codeword in the code, and choosing the closest, is called maximum likelihood decoding because it is guaranteed to return the most likely codeword. However, such an exhaustive search is feasible only when k is small. For codes with thousands of message bits in a codeword it becomes far too computationally expensive to directly compare the received string with every one of the 2^k codewords in the code. Numerous ingenious solutions have been proposed to make this task less complex, including choosing algebraic codes and exploiting their structure to speed up the decoding or, as for LDPC codes, devising decoding methods which are not ML but which can perform very well with a much reduced complexity.

1.3 Low-density parity-check (LDPC) codes

As their name suggests, LDPC codes are block codes with parity-check matrices that contain only a very small number of non-zero entries. It is the sparseness of H which guarantees both a decoding complexity which increases only linearly with the code length and a minimum distance which also increases linearly with the code length.

Aside from the requirement that H be sparse, an LDPC code itself is no different to any other block code. Indeed existing block codes can be successfully used with the LDPC iterative decoding algorithms if they can be represented by a sparse parity-check matrix. Generally, however, finding a sparse parity-check matrix for an existing code is not practical. Instead LDPC codes are designed by constructing a sparse parity-check matrix first and then determining a generator matrix for the code afterwards.

The biggest difference between LDPC codes and classical block codes is how they are decoded. Classical block codes are generally decoded with ML like decoding algorithms and so are usually short and designed algebraically to make this task less complex. LDPC codes however are decoded iteratively using a graphical representation of their parity-check matrix and so are designed with the properties of H as a focus.

An LDPC code parity-check matrix is called *(w_c,w_r)-regular* if each code bit is contained in a fixed number, w_c, of parity checks and each parity-check equation contains a fixed number, w_r, of code bits.

Example 1.12.

A regular parity-check matrix for the code in Example 1.3 with $w_c = 2, w_r = 3$ and $\text{rank}_2(H) = 3$, which satisfies (1.4) is

$$H = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}. \tag{1.16}$$

For an irregular parity-check matrix we designate the fraction of columns of weight i by v_i and the fraction of rows of weight i by h_i. Collectively the set $\mathbf{v}$ and $\mathbf{h}$ is called the *degree distribution* of the code.

Example 1.13.

The parity-check matrix in Equation 1.3 is irregular with degree distribution $v_1 = 1/2, v_2 = 1/3, v_3 = 1/6, h_3 = 2/3$ and $h_4 = 1/3$.

A regular LDPC code will have,

$$m \cdot w_r = n \cdot w_c, \tag{1.17}$$

ones in its parity-check matrix. Similarly, for an irregular code

$$m(\sum_i h_i \cdot i) = n(\sum_i v_i \cdot i). \tag{1.18}$$

1.3.1 LDPC constructions

The construction of binary LDPC codes involves assigning a small number of the values in an all-zero matrix to be 1 so that the rows and columns have the required degree distribution.

The original LDPC codes presented by Gallager are regular and defined by a banded structure in H. The rows of Gallager's parity-check matrices are divided into w_c sets with M/w_c rows in each set. The first set of rows contains w_r consecutive ones ordered from left to right across the columns. (i.e. for $i \leq M/w_c$, the i-th row has non zero entries in the $((i-1)K+1)$-th to i-th columns). Every other set of rows is a randomly chosen column permutation of this first set. Consequently every column of H has a '1' entry once in every one of the w_c sets.

Example 1.14.

A length 12 (3,4)-regular Gallager parity-check matrix is

$$H = \left[\begin{array}{cccccccccccc} 1&1&1&1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&1&1&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0&1&1&1&1 \\ \hline 1&0&1&0&0&1&0&0&0&1&0&0 \\ 0&1&0&0&0&0&1&1&0&0&0&1 \\ 0&0&0&1&1&0&0&0&1&0&1&0 \\ \hline 1&0&0&1&0&0&1&0&0&1&0&0 \\ 0&1&0&0&0&1&0&1&0&0&1&0 \\ 0&0&1&0&1&0&0&0&1&0&0&1 \end{array}\right].$$

Another common construction for LDPC codes is a method proposed by MacKay and Neal. In this method columns of H are added one column at a time from left to right. The weight of each column is chosen to obtain the correct bit degree distribution and the location of the non-zero entries in each column chosen randomly from those rows which are not yet full. If at any point there are rows with more positions unfilled then there are columns remaining to be added, the row degree distributions for H will not be exact. The process can be started again or back tracked by a few columns, until the correct row degrees are obtained.

Example 1.15.

A length 12 (3,4)-regular MacKay Neal parity-check matrix is

$$H = \left[\begin{array}{cccccccccccc} 1&0&0&0&0&1&0&1&0&1&0&0 \\ 1&0&0&1&1&0&0&0&0&0&\mathbf{1}&0 \\ 0&1&0&0&1&0&1&0&1&0&0&0 \\ 0&0&1&0&0&1&0&0&0&0&\mathbf{1}&1 \\ 0&0&1&0&0&0&1&1&0&0&0&1 \\ 0&1&0&0&1&0&0&0&1&0&\mathbf{1}&0 \\ 1&0&0&1&0&0&1&0&0&1&0&0 \\ 0&1&0&0&0&1&0&1&0&1&0&0 \\ 0&0&1&1&0&0&0&0&1&0&0&1 \end{array}\right].$$

When adding the 11-th column, shown in bold, the unfilled rows were the 2-nd 4-th, 5-th, 6-th and 9-th from which the 2-nd, 4-th and 6-th were chosen.

Another type of LDPC codes called *repeat-accumulate codes* have weight-2 columns in a step pattern for the last m columns of H. This structure makes the repeat-accumulate codes systematic and allows them to be easily encoded.

Example 1.16.

A length 12 rate-1/4 repeat-accumulate code is

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

The first three columns of H correspond to the message bits. The first parity-bit (the fourth column of H) can be encoded as $c_4 = c_1$, the second as $c_5 = c_4 \oplus c_1$ and the next as $c_6 = c_5 \oplus c_2$ and so on. In this way each parity-bit can be computed one at a time using only the message bits and the one previously calculated parity-bit.

Since LDPC codes are often constructed pseudo-randomly we often talk about the set (or *ensemble*) of all possible codes with certain parameters (for example a certain degree distribution) rather than about a particular choice of parity-check matrix with those parameters.

LDPC codes are often represented in graphical form by a *Tanner graph*. The Tanner graph consists of two sets of vertices: n vertices for the codeword bits (called *bit nodes*), and m vertices for the parity-check equations (called *check nodes*). An edge joins a bit node to a check node if that bit is included in the corresponding parity-check equation and so the number of edges in the Tanner graph is equal to the number of ones in the parity-check matrix.

Example 1.17.

The Tanner graph of the parity-check matrix Example 1.12 is shown in Fig. 1.1. The bit vertices are represented by circular nodes and the check vertices by square nodes.

The Tanner graph is sometimes drawn vertically with the bit nodes on the left and check nodes on the right with bit nodes sometimes referred to as *left nodes* or *variable nodes* and the check nodes as *right nodes* or *constraint nodes*. For a systematic code the message bit nodes can be distinguished from the parity bit nodes by placing them on separate sides of the graph.

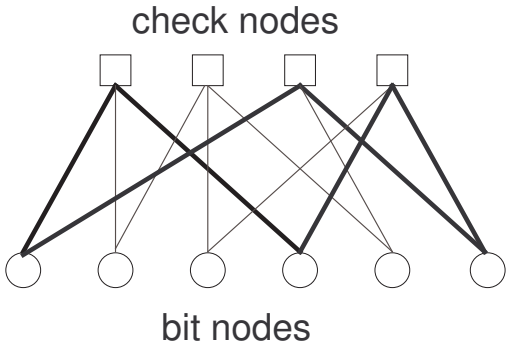

Figure 1.1: The Tanner graph representation of the parity-check matrix in (1.16). A 6-cycle is shown in bold.

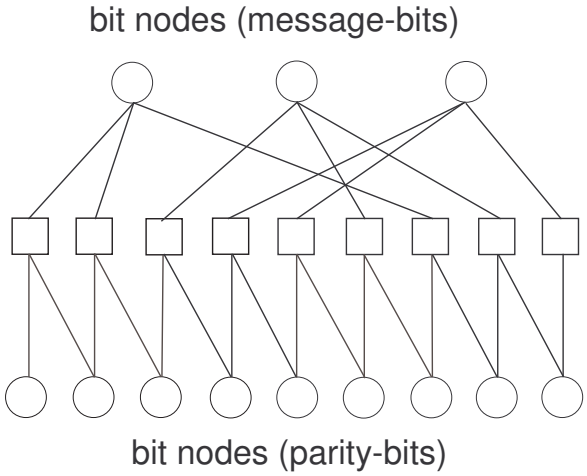

Figure 1.2: The Tanner graph representation of the parity-check matrix in Example 1.16

Example 1.18.

The Tanner graph of the parity-check matrix in Example 1.16 is shown in Fig. 1.2. The message bit nodes are shown at the top of the graph and the parity bit nodes at the bottom.

A *cycle* in a Tanner graph is a sequence of connected vertices which start and end at the same vertex in the graph, and which contain other vertices no more than once. The length of a cycle is the number of edges it contains, and the *girth* of a graph is the size of its smallest cycle.

Example 1.19.

A cycle of size 6 is shown in bold in Fig. 1.1.

The Mackay Neal construction method for LDPC codes can be adapted to avoid cycles of length 4, called 4-cycles, by checking each pair of columns in H to see if they overlap in two places. The construction of 4-cycle free codes using this method is given in Algorithm 1. Input is the code length n, rate r, and column and row degree distributions $\mathbf{v}$ and $\mathbf{h}$. The vector α is a length n vector which contains an entry i for each column in H of weight i and the vector β is a length m vector which contains an entry i for each row in H of weight i.

Algorithm 1 MacKay Neal LDPC Codes

procedure MN CONSTRUCTION(n,r,$\mathbf{v}$,$\mathbf{h}$) ▷ Required length, rate and degree distributions
  $H =$ all zero $n(1-r) \times n$ matrix ▷ Initialization
  $\alpha = []$;
  for $i = 1 : \max(\mathbf{v})$ **do**
    for $j = 1 : v_i \times n$ **do**
      $\alpha = [\alpha, i]$
    end for
  end for
  $\beta = []$
  for $i = 1 : \max(\mathbf{h})$ **do**
    for $j = 1 : h_i \times m$ **do**
      $\beta = [\beta, i]$
    end for
  end for

  for $i = 1 : n$ **do** ▷ Construction
    $\mathbf{c} =$ random subset of β, of size α_i
    for $j = 1 : \alpha_i$ **do**
      $H(c_j, i) = 1$
    end for
    $\alpha = \alpha - \mathbf{c}$
  end for

  repeat
    for $i = 1 : n - 1$ **do** ▷ Remove 4-cycles
      for $j = i + 1 : n$ **do**
        if $|H(:, i) \bigcup H(:, j)| > 1$ **then**
          permute the entries in the j-th column
        end if
      end for
    end for
  until cycles removed
end procedure

Removing the 4-cycles does have the effect of disturbing the row degree distribution. For long codes H will be very sparse and so 4-cycles very uncommon and the effect on the row degrees will be negligible. However, for short codes 4-cycle free parity-check matrices can be constructed much more effectively by

using algebraic methods, as we will see later.

Alternatively, the Mackay Neal construction method for LDPC codes can be adapted to avoid 4-cycles, without disturbing the row degree distribution, by checking each column before it is added to see if it will cause a cycle with any of the already chosen columns and rejecting it if it does.

Example 1.20.

If a 4-cycle free code was required in Example 1.15 the fourth column would have been discarded, and a new one chosen, because it causes a 4-cycle with the first column in H.

1.3.2 Encoding

Earlier we noted that a generator matrix for a code with parity-check matrix H can be found by performing Gauss-Jordan elimination on H to obtain it in the form

$$H = [A, I_{n-k}],$$

where A is a $(n-k) \times k$ binary matrix and I_{n-k} is the size $n-k$ identity matrix. The generator matrix is then

$$G = [I_k, A^T].$$

Here we will go into this process in more detail using an example.

Example 1.21.

We wish to encode the length 10 rate-1/2 LDPC code

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

First, we put H into *row-echelon form* (i.e. so that in any two successive rows that do not consist entirely of zeros, the leading 1 in the lower row occurs further to the right than the leading 1 in the higher row).

The matrix H is put into this form by applying *elementary row operations* in $GF(2)$, which are; interchanging two rows or adding one row to another modulo 2. From linear algebra we know that by using only elementary row operations the modified parity-check matrix will have the same codeword set as the original, (as the new system of linear equations will have an unchanged solution set).

The 1-st and 2-nd columns of H already have ones on the diagonal and entries in these columns below the diagonal are removed by replacing the 4-th row with the modulo-2 sum of the 1-st and 4-th rows. The 3-rd column of H does not have a one on the diagonal but this can be obtained by swapping the

3-rd and 5-th rows. Finally, replacing the 5-th row with the modulo two sum of the 5-th and 4-th rows gives H_r in row-echelon form:

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

Next the parity-check matrix is put into *reduced* row-echelon form (i.e. so that any column that contains a leading one has zeros everywhere else). The 1-st column is already correct and the entry in the 2-nd column above the diagonal is removed by replacing the 1-st row with the modulo-2 sum of the 1-st and 2-nd rows. Similarly the entry in the 3-nd column above the diagonal is removed by replacing the 2-nd row with the modulo-2 sum of the 2-nd and 3-rd rows. To clear the 4-th column the 1-st row is replace with the modulo-2 sum of the 1-st and 4-th rows. Finally, to clear the 5-th column involves adding the 5-th row to the 1-st, 2-nd and 4-th rows gives H_{rr} in reduced row-echelon form:

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

Lastly, using column permutations we put the parity-check matrix into standard form (where the last m columns of H_{std} are the m columns of H_{rr} which contain the leading ones):

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

In this final step column permutations have been used and so the codewords of H_{std} will be permuted versions of the codewords corresponding to H. A solution is to keep track of the column permutation used to create H_{std}, which in this case is

$$\Pi = \begin{bmatrix} 6 & 7 & 8 & 9 & 10 & 1 & 2 & 3 & 4 & 5 \end{bmatrix},$$

and apply the inverse permutation to each H_{std} codeword before it is transmitted.

Alternatively, if the channel is memoryless, and so the order of codeword bits is unimportant, a far easier option is to apply Π to the original H to give a parity-check matrix

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

with the same properties as H but which shares the same codeword bit ordering as H_{std}.

Finally, a generator G for the code with parity-check matrices H_{std} and H' is given by

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

All of this processing can be done off-line and just the matrices G and H' provided to the encoder and decoder respectively. However, the drawback of this approach is that, unlike H, the matrix G will most likely not be sparse and so the matrix multiplication

$$\mathbf{c} = \mathbf{u}G,$$

at the encoder will have complexity in the order of n^2 operations. As n is large for LDPC codes, from thousands to hundreds of thousands of bits, the encoder can become prohibitively complex. Later we will see that structured parity-check matrices can be used to significantly lower this implementation complexity, however for arbitrary parity-check matrices a good approach is to avoid constructing G at all and instead to encode using back substitution with H as is demonstrated in the following.

(Almost) linear-time encoding for LDPC codes

Rather than finding a generator matrix for H, an LDPC code can be encoded using the parity-check matrix directly by transforming it into upper triangular form and using back substitution. The idea is to do as much of the transformation as possible using only row and column permutations so as to keep as much of H as possible sparse.

Firstly, using only row and column permutations, the parity-check matrix is put into *approximate lower triangular form*:

$$H_t = \begin{bmatrix} A & B & T \\ C & D & E \end{bmatrix},$$

where the matrix T is a lower triangular matrix (that is T has ones on the diagonal from left to right and all entries above the diagonal zero) of size $(m-g) \times (m-g)$. If H_t is full rank the matrix B is size $m-g \times g$ and A is size $m-g \times k$. The g rows of H left in C, D, and E are called the *gap* of the approximate representation and the smaller g the lower the encoding complexity for the LDPC code.

Example 1.22. ———————————————————

We wish to encode the message $\mathbf{u} = [1\ 1\ 0\ 0\ 1]$ with the same length 10

rate-1/2 LDPC code from Example 1.21:

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

Instead of putting H into reduced row-echelon form we put it into approximate lower triangular form using only row and column swaps. For this H we swap the 2-nd and 3-rd rows and 6-th and 10-th columns to obtain:

$$H_t = \left[\begin{array}{ccccc|ccc|ccc} 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ \hline 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{array}\right].$$

with a gap of two.

Once in upper triangular format, Gauss-Jordan elimination is applied to clear E which is equivalent to multiplying H_t by

$$\begin{bmatrix} I_{m-g} & 0 \\ -ET^{-1} & I_g \end{bmatrix},$$

to give

$$\tilde{H} = \begin{bmatrix} I_{m-g} & 0 \\ -ET^{-1} & I_g \end{bmatrix} H_t = \begin{bmatrix} A & B & T \\ \tilde{C} & \tilde{D} & 0 \end{bmatrix}$$

where

$$\tilde{C} = -ET^{-1}A + C,$$

and

$$\tilde{D} = -ET^{-1}B + D.$$

Example 1.23. ---

Continuing from Example 1.22 we have

$$T^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

and

$$\begin{bmatrix} I_{m-g} & 0 \\ -ET^{-1} & I_g \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix},$$

to give

$$\tilde{H} = \left[\begin{array}{ccccc|cc|ccc} 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ \hline 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \end{array}\right].$$

When applying Gauss-Jordan elimination to clear E only $\tilde{C}$ and $\tilde{D}$ are effected, the rest of the parity-check matrix remains sparse.

Finally, to encode using $\tilde{H}$ the codeword $\mathbf{c} = [c_1 c_2, \ldots, c_n]$ is divided into three parts, $\mathbf{c} = [\mathbf{u}, \mathbf{p_1}, \mathbf{p_2}]$, where $\mathbf{u} = [u_1, u_2, \ldots, u_k]$ is the k-bit message, $\mathbf{p_1} = [p_{1_1}, p_{1_2}, \ldots, p_{1_g}]$, holds the first g parity bits and $\mathbf{p_2} = [p_{2_1}, p_{2_2}, \ldots, p_{2_{m-g}}]$ holds the remaining parity bits.

The codeword $\mathbf{c} = [\mathbf{u}, \mathbf{p_1}, \mathbf{p_2}]$ must satisfy the parity-check equation $\mathbf{c}\tilde{H}^T = \mathbf{0}$ and so

$$A\mathbf{u} + B\mathbf{p_1} + T\mathbf{p_2} = \mathbf{0}, \tag{1.19}$$

and

$$\tilde{C}\mathbf{u} + \tilde{D}\mathbf{p_1} + \mathbf{0}\mathbf{p_2} = \mathbf{0}. \tag{1.20}$$

Since E has been cleared, the parity bits in $\mathbf{p_1}$ depend only on the message bits, and so can be calculated independently of the parity bits in $\mathbf{p_2}$. If $\tilde{D}$ is invertible, $\mathbf{p_1}$ can be found from (1.20):

$$\mathbf{p_1} = \tilde{D}^{-1}\tilde{C}\mathbf{u}. \tag{1.21}$$

If $\tilde{D}$ is not invertible the columns of $\tilde{H}$ can be permuted until it is. By keeping g as small as possible the added complexity burden of the matrix multiplication in Equation 1.21, which is $\bigcirc(g^2)$, is kept low.

Once $\mathbf{p_1}$ is known $\mathbf{p_2}$ can be found from (1.19):

$$\mathbf{p_2} = -T^{-1}(A\mathbf{u} + B\mathbf{p_1}), \tag{1.22}$$

where the sparseness of A, B and T can be employed to keep the complexity of this operation low and, as T is upper triangular, $\mathbf{p_2}$ can be found using back substitution.

Example 1.24.

Continuing from Example 1.23 we partition the length 10 codeword $\mathbf{c} = [c_1, c_2, \ldots, c_{10}]$ as $\mathbf{c} = [\mathbf{u}, \mathbf{p_1}, \mathbf{p_2}]$ where $\mathbf{p1} = [c_6, c_7]$ and $\mathbf{p2} = [c_8, c_9, c_{10}]$. The parity bits in $\mathbf{p_1}$ are calculated from the message using Equation 1.21:

$$\mathbf{p_1} = \tilde{D}^{-1}\tilde{C}\mathbf{u} = \left[\begin{array}{cc} 1 & 0 \\ 1 & 1 \end{array}\right] \left[\begin{array}{ccccc} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{array}\right] \left[\begin{array}{c} 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{array}\right] = \left[\begin{array}{cc} 1 & 0 \end{array}\right]$$

As T is upper-triangular the bits in $\mathbf{p_2}$ can then be calculated using back substitution.

$$\begin{aligned} p_{2_1} &= u_1 \oplus u_2 \oplus u_4 \oplus u_5 = 1 \oplus 1 \oplus 0 \oplus 1 = 1 \\ p_{2_2} &= u_4 \oplus p_{1_1} \oplus p_{2_1} = 0 \oplus 1 \oplus 1 = 0 \\ p_{2_3} &= u_2 \oplus u_3 \oplus u_5 \oplus p_{1_2} = 1 \oplus 0 \oplus 1 \oplus 0 = 0 \end{aligned}$$

and the codeword is $\mathbf{c} = [1\ 1\ 0\ 0\ 1\ 1\ 0\ 1\ 0\ 0]$.

Again column permutations were used to obtain H_t from H and so either H_t, or H with the same column permutation applied, will be used at the decoder. Note that since the parity-check matrix used to compute G in Example 1.21 is a column permuted version of H_t, the set of codewords generated by both encoders will not be the same.

1.4 Bibliographic notes

LDPC codes were first introduced by Gallager in his 1962 thesis [1]. In his work, Gallager used a graphical representation of the bit and parity-check sets of regular LDPC codes, to describe the application of iterative decoding. The systematic study of codes on graphs however is largely due to Tanner who, in 1981, formalized the use of bipartite graphs for describing families of codes [2].

Irregular LDPC codes were first proposed by a group of researchers in the late 90's [3,4] and it is these codes which can produce performances within a fraction of a decibel from capacity [5].

The encoding algorithm presented here is from [6] and the two pseudo-random constructions we have considered can be found in [1] and [7]. For more detail on classical block codes we like the error correction texts [8], and [9] or, for those interested in a more mathematical treatment, [10] and [11].

Topic 2: Message-Passing Decoding

The class of decoding algorithms used to decode LDPC codes are collectively termed *message-passing* algorithms since their operation can be explained by the passing of messages along the edges of a Tanner graph. Each Tanner graph node works in isolation, only having access to the information contained in the messages on the edges connected to it. The message-passing algorithms are also known as *iterative decoding* algorithms as the messages pass back and forward between the bit and check nodes iteratively until a result is achieved (or the process halted). Different message-passing algorithms are named for the type of messages passed or for the type of operation performed at the nodes.

In some algorithms, such as bit-flipping decoding, the messages are binary and in others, such as *belief propagation* decoding, the messages are probabilities which represent a level of belief about the value of the codeword bits. It is often convenient to represent probability values as log likelihood ratios, and when this is done belief propagation decoding is often called sum-product decoding since the use of log likelihood ratios allows the calculations at the bit and check nodes to be computed using sum and product operations.

2.1 Message-passing on the binary erasure channel

On the binary erasure channel (BEC) a transmitted bit is either received correctly or completely erased with some probability ε. Since the bits which are received are always completely correct the task of the decoder is to determine the value of the unknown bits.

If there exists a parity-check equation which includes only one erased bit the correct value for the erased bit can be determined by choosing the value which satisfies even parity.

Example 2.1.

The code in example 1.3 includes the parity-check equation

$$c_1 \oplus c_2 \oplus c_4 = 0.$$

If the value of bit c_1 is known to be '0' and the value of bit c_2 is known to be '1', then the value of bit c_4 must be '1' if c_1, c_2 and c_4 are part of a valid codeword for this code.

In the message-passing decoder each check node determines the value of an erased bit if it is the only erased bit in its parity-check equation.

The messages passed along the Tanner graph edges are straightforward: a bit node sends the same outgoing message M to each of its connected check nodes. This message, labeled M_i for the i-th bit node, declares the value of the bit '1', '0' if it is known or 'x' if it is erased. If a check node receives only one 'x' message, it can calculate the value of the unknown bit by choosing the value which satisfies parity. The check nodes send back different messages to each of their connected bit nodes. This message, labeled $E_{j,i}$ for the message from the j-th check node to the i-th bit node, declares the value of the i-bit '1', '0' or 'x' as determined by the j-th check node. If the bit node of an erased bit receives an incoming message which is '1' or '0' the bit node changes its value to the value of the incoming message. This process is repeated until all of the bit values are known, or until some maximum number of decoder iterations has passed and the decoder gives up.

We use the notation B_j to represent the set of bits in the j-th parity-check equation of the code. So for the code in Example 1.12 we have

$$B_1 = \{1, 2, 4\}, B_2 = \{2, 3, 5\}, B_3 = \{1, 5, 6\}, B_4 = \{3, 4, 6\}.$$

Similarly, we use the notation A_i to represent the parity-check equations which check on the i-th bit of the code. So for the code in Example 1.12 we have

$$A_1 = \{1, 3\}, A_2 = \{1, 2\}, A_3 = \{2, 4\}, A_5 = \{1, 4\}, A_5 = \{2, 3\}, A_6 = \{3, 4\}.$$

Algorithm 2 outlines message-passing decoding on the BEC. Input is the received values from the detector, $\mathbf{y} = [y_1, \ldots, y_n]$ which can be '1', '0' or 'x', and output is $M = [M_1, \ldots, M_n]$ which can also take the values '1', '0' or 'x'.

Example 2.2.

The LDPC code from Example 1.12 is used to encode the codeword

$$\mathbf{c} = [0\ 0\ 1\ 0\ 1\ 1].$$

$\mathbf{c}$ is sent though an erasure channel and the vector

$$\mathbf{y} = [0\ 0\ 1\ x\ x\ x]$$

is received. Message-passing decoding is used to recover the erased bits.

Initialization is $M_i = r_i$ so

$$M = [0\ 0\ 1\ x\ x\ x].$$

For Step 1 the check node messages are calculated. The 1-st check node is joined to the 1-st, 2-nd and 4-th bit nodes, and so has incoming messages '1', '0' and 'x. Since the check node has one incoming 'x' message, from the 4-th bit node, its outgoing message on this edge, $E_{1,4}$, will be the value of the 4-th codeword bit:

$$\begin{aligned} E_{1,4} &= M_1 \oplus M_2 \\ &= 0 \oplus 0 \\ &= 0. \end{aligned}$$

The 2-nd check includes the 2-nd, 3-rd and 5-th bits, and so has incoming messages '0', '1' and 'x. Since the check node has one incoming 'x' message, from

Algorithm 2 Erasure Decoding

procedure DECODE(**y**)

  $I = 0$ ▷ Initialization
  for $i = 1 : n$ **do**
    $M_i = y_i$
  end for
  repeat

    for $j = 1 : m$ **do** ▷ Step 1: Check messages
      for all $i \in B_j$ **do**
        if all messages into check j other than M_i are known **then**
          $E_{j,i} = \sum_{i' \in B_j, i' \neq i} (M_{i'} \mod 2)$
        else
          $E_{j,i} =$ 'x'
        end if
      end for
    end for

    for $i = 1 : n$ **do** ▷ Step 2: Bit messages
      if $M_i =$ 'unknown' **then**
        if there exists a $j \in A_i$ s.t. $E_{j,i} \neq$ 'x' **then**
          $M_i = E_{j,i}$
        end if
      end if
    end for

    if all M_i known or $I = I_{\max}$ **then** ▷ Test
      Finished
    else
      $I = I + 1$
    end if
  until Finished
end procedure

the 5-th bit node, its outgoing message on this edge, $E_{2,5}$, will be the value of the 5-th codeword bit:

$$\begin{aligned} E_{2,5} &= M_2 \oplus M_3 \\ &= 0 \oplus 1 \\ &= 1. \end{aligned}$$

The 3-rd check includes the 1-st, 5-th and 6-th bits, and so has incoming messages '0', 'x' and 'x'. Since this check node receives two 'x' messages, it cannot be used to determine the value of any of the bits. In this case the outgoing messages from the check node are all 'x'. Similarly, the 4-th check includes the 3-rd, 4-th and 6-th bits and so receives two 'x' messages and thus also cannot used to determine the value of any of the bits.

In Step 2 each bit node that has an unknown value uses its incoming mes-

sages to update its value if possible. The 4-th bit is unknown and has incoming message, of '0' ($E_{1,4}$) and 'x' ($E_{4,4}$) and so it changes its value to '0'. The 5-th bit is also unknown and has an incoming messages of '1' ($E_{2,5}$) and 'x' ($E_{3,5}$) and so it changes its value to '1'. The 6-th bit is also unknown but it has incoming messages of 'x' ($E_{3,6}$) and 'x' ($E_{4,6}$) so it cannot change its value. At the end of Step 2 we thus have

$$M = [0\ 0\ 1\ 0\ 1\ x].$$

For the test, there is a remaining unknown bit (the 6-th bit) and so the algorithm continues.

Repeating Step 1 the 3-rd check node is joined to the 1-st, 5-th and 6-th bit nodes, and so this check node has one incoming 'x' message, M_6. The outgoing message from this check to the 6-th bit node, $E_{3,6}$, is the value of the 6-th codeword bit.

$$\begin{aligned} E_{3,6} &= M_1 \oplus M_5 \\ &= 1 \oplus 0 \\ &= 1. \end{aligned}$$

The 4-th check node is joined to the 3-rd, 4-th and 6-th bit nodes, and so the this check node has one incoming 'x' message, M_6. The outgoing message from this check to the 6-th bit node, $E_{4,6}$, is the value of the 6-th codeword bit.

$$\begin{aligned} E_{3,6} &= M_3 \oplus M_4 \\ &= 0 \oplus 1 \\ &= 1. \end{aligned}$$

In Step 2 the 6-th bit is unknown and has incoming messages, $E_{3,6}$ and $E_{4,6}$ with value '1' and so it changes its value to '1'. Since the received bits from the channel are always correct the messages from the check nodes will always agree. (In the bit-flipping algorithm we will see a strategy for when this is not the case.)
This time at the test there are no unknown codeword bits and so the algorithm halts and returns

$$M = [0\ 0\ 1\ 0\ 1\ 1]$$

as the decoded codeword. The received string has therefore been correctly determined despite half of the codeword bits having been erased. Fig 2.1 shows graphically the messages passed in the message-passing decoder.

Since the received bits in an erasure channel are either correct or unknown (no errors are introduced by the channel) the messages passed between nodes are always the correct bit values or 'x'. When the channel introduces errors into the received word, as in the binary symmetric or AWGN channels, the messages in message-passing decoding are instead the best guesses of the codeword bit values based on the current information available to each node.

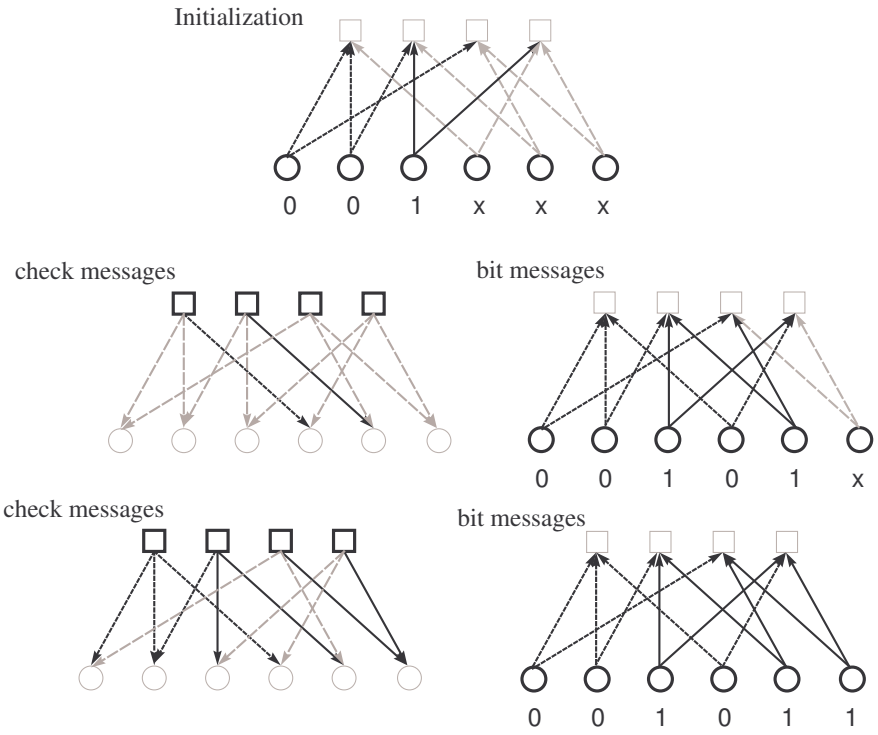

Figure 2.1: Message-passing decoding of the received string $y = [0\ 0\ 1\ x\ x\ x]$. Each subfigure indicates the decision made at each step of the decoding algorithm based on the messages from the previous step. For the messages, a dotted arrow corresponds to the messages "bit $= 0$" while a solid arrow corresponds to "bit $= 1$", and a light dashed arrow corresponds to "bit x".

2.2 Bit-flipping decoding

The *bit-flipping* algorithm is a hard-decision message-passing algorithm for LDPC codes. A binary (hard) decision about each received bit is made by the detector and this is passed to the decoder. For the bit-flipping algorithm the messages passed along the Tanner graph edges are also binary: a bit node sends a message declaring if it is a one or a zero, and each check node sends a message to each connected bit node, declaring what value the bit is based on the information available to the check node. The check node determines that its parity-check equation is satisfied if the modulo-2 sum of the incoming bit values is zero. If the majority of the messages received by a bit node are different from its received value the bit node changes (flips) its current value. This process is repeated until all of the parity-check equations are satisfied, or until some maximum number of decoder iterations has passed and the decoder gives up.

The bit-flipping decoder can be immediately terminated whenever a valid codeword has been found by checking if all of the parity-check equations are satisfied. This is true of all message-passing decoding of LDPC codes and has two important benefits; firstly additional iterations are avoided once a solution has been found, and secondly a failure to converge to a codeword is always detected.

The bit-flipping algorithm is based on the principal that a codeword bit involved in a large number of incorrect check equations is likely to be incorrect itself. The sparseness of H helps spread out the bits into checks so that parity-check equations are unlikely to contain the same set of codeword bits. In Example 2.4 we will show the detrimental effect of overlapping parity-check equations.

The bit-flipping algorithm is presented in Algorithm 3. Input is the hard decision on the received vector, $\mathbf{y} = [y_1, \ldots, y_n]$, and output is $M = [M_1, \ldots, M_n]$.

Example 2.3.

The LDPC code from Example 1.12 is used to encode the codeword

$$\mathbf{c} = [0\ 0\ 1\ 0\ 1\ 1].$$

$\mathbf{c}$ is sent though a BSC channel with crossover probability $p = 0.2$ and the received signal is

$$\mathbf{y} = [1\ 0\ 1\ 0\ 1\ 1].$$

Initialization is $M_i = r_i$ so

$$M = [1\ 0\ 1\ 0\ 1\ 1].$$

For Step 1 the check node messages are calculated. The 1-st check node is joined to the 1-st, 2-nd and 4-th bit nodes, $B_1 = [1, 2, 4]$, and so the message for the 1-st check is

$$\begin{aligned} E_{1,1} &= M_2 \oplus M_4 \\ &= 0 \oplus 0 \\ &= 0, \end{aligned}$$

$$\begin{aligned} E_{1,2} &= M_1 \oplus M_4 \\ &= 1 \oplus 0 \\ &= 1, \end{aligned}$$

Algorithm 3 Bit-flipping Decoding

procedure DECODE($\mathbf{y}$)

  $I = 0$ ▷ Initialization
  for $i = 1 : n$ **do**
    $M_i = y_i$
  end for

  repeat
    for $j = 1 : m$ **do** ▷ Step 1: Check messages
      for $i = 1 : n$ **do**
        $E_{j,i} = \sum_{i' \in B_j, i' \neq i} (M_{i'} \mod 2)$
      end for
    end for

    for $i = 1 : n$ **do** ▷ Step 2: Bit messages
      if the messages $E_{j,i}$ disagree with y_i **then**
        $M_i = (r_i + 1 \mod 2)$
      end if
    end for

    for $j = 1 : m$ **do** ▷ Test: are the parity-check
      $L_j = \sum_{i' \in B_j} (M_{i'} \mod 2)$ ▷ equations satisfied
    end for
    if all $L_j = 0$ or $I = I_{\max}$ **then**
      Finished
    else
      $I = I + 1$
    end if
  until Finished
end procedure

$$\begin{aligned} E_{1,4} &= M_1 \oplus M_2 \\ &= 1 \oplus 0 \\ &= 1. \end{aligned}$$

The 2-nd check includes the 2-nd, 3-rd and 5-th bits, $B_2 = [2, 3, 5]$, and so the message for the 2-nd check is

$$\begin{aligned} E_{2,2} &= M_3 \oplus M_5 \\ &= 1 \oplus 1 \\ &= 0, \end{aligned}$$

$$\begin{aligned} E_{2,3} &= M_2 \oplus M_5 \\ &= 0 \oplus 1 \\ &= 1, \end{aligned}$$

$$\begin{aligned} E_{2,5} &= M_2 \oplus M_3 \\ &= 0 \oplus 1 \\ &= 1. \end{aligned}$$

Repeating for the remaining check nodes gives:

$$E_{3,1} = 0, \quad E_{3,5} = 0, \quad E_{3,6} = 0,$$
$$E_{4,3} = 1, \quad E_{4,4} = 0, \quad E_{4,6} = 1.$$

In Step 2 the 1-st bit has messages from the 1-st and 3-rd checks, $A_1 = [1, 3]$ both zero. Thus the majority of the messages into the 1-st bit node indicate a value different from the received value and so the 1-st bit node flips its value. The 2-nd bit has messages from the 1-st and 2-nd checks, $A_2 = [1, 2]$ which are one and so agree with the received value. Thus the 2-nd bit does not flip its value. Similarly, none of the remaining bit nodes have enough check to bit messages differing from their received value and so they all also retain their current values. The new bit to check messages are thus

$$M = [0\ 0\ 1\ 0\ 1\ 1].$$

For the test the parity-checks are calculated. For the first check node

$$\begin{aligned} L_1 &= M_1 \oplus M_2 \oplus M_4 \\ &= 0 \oplus 0 \oplus 0 \\ &= 0. \end{aligned}$$

For the second check node

$$\begin{aligned} L_2 &= M_2 \oplus M_3 \oplus M_5 \\ &= 0 \oplus 1 \oplus 1 \\ &= 0, \end{aligned}$$

and similarly for the 3-rd and 4-th check nodes:

$$\begin{aligned} L_3 &= 0, \\ L_4 &= 0. \end{aligned}$$

There are thus no unsatisfied checks and so the algorithm halts and returns

$$M = [0\ 0\ 1\ 0\ 1\ 1]$$

as the decoded codeword. The received string has therefore been correctly decoded without requiring an explicit search over all possible codewords. The decoding steps are shown graphically in Fig. 2.2.

The existence of cycles in the Tanner graph of a code reduces the effectiveness of the iterative decoding process. To illustrate the detrimental effect of a 4-cycle we use a new LDPC code with Tanner graph shown in Fig. 2.3. For this Tanner graph there is a 4-cycle between the first two bit nodes and the first two check nodes.

Example 2.4. ———————————————————————

A valid codeword for the code with Tanner graph in Fig. 2.3 is

$$\mathbf{c} = [0\ 0\ 1\ 0\ 0\ 1].$$

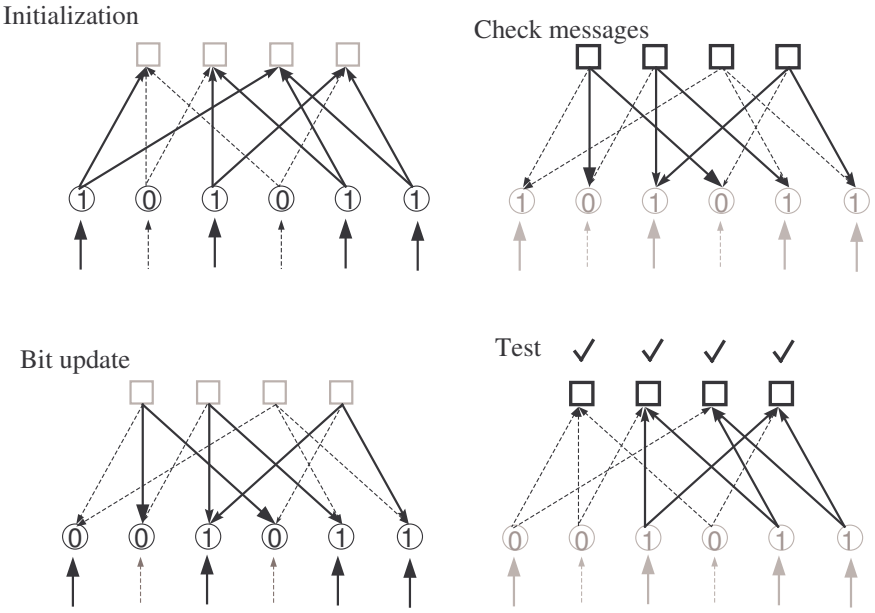

Figure 2.2: Bit-flipping decoding of the received string $y = [1\ 0\ 1\ 0\ 1\ 1]$. Each sub-figure indicates the decision made at each step of the decoding algorithm based on the messages from the previous step. A cross ($\times$) represents that the parity check is not satisfied while a tick ($\checkmark$) indicates that it is satisfied. For the messages, a dashed arrow corresponds to the messages "bit $= 0$" while a solid arrow corresponds to "bit $= 1$".

This codeword is sent through a binary input additive white Gaussian noise channel with binary phase shift keying (BPSK) signaling and

$$[-1.1\ 1.5\ -0.5\ 1\ +1.8\ -2]$$

is received. The detector makes a hard decision on each codeword bit and returns

$$\mathbf{y} = [1\ 0\ 1\ 0\ 0\ 1]$$

As in Example 2.4 the effect of the channel has been that the first bit is incorrect.

The steps of the bit-flipping algorithm for this received string are shown in Fig. 2.3. The initial bit values are $1, 0, 1, 0, 0$, and 1, respectively, and messages are sent to the check nodes indicating these values. Step 1 reveals that the 1-st and 2-nd parity-check equations are not satisfied and so at the test the algorithm continues. In Step 2 both the 1-st and 2-nd bits have the majority of their messages indicating that the received value is incorrect and so both flip their bit values. When Step 1 is repeated we see that the 1-st and 2-nd parity-check equations are again not satisfied. In further iterations the first two bits continue to flip their values together such that one of them is always incorrect and the algorithm fails to converge. As a result of the 4-cycle, each of the first two codeword bits are involved in the same two parity-check equations, and so when both of the parity-check equations are unsatisfied, it is not possible to determine which bit is in error.

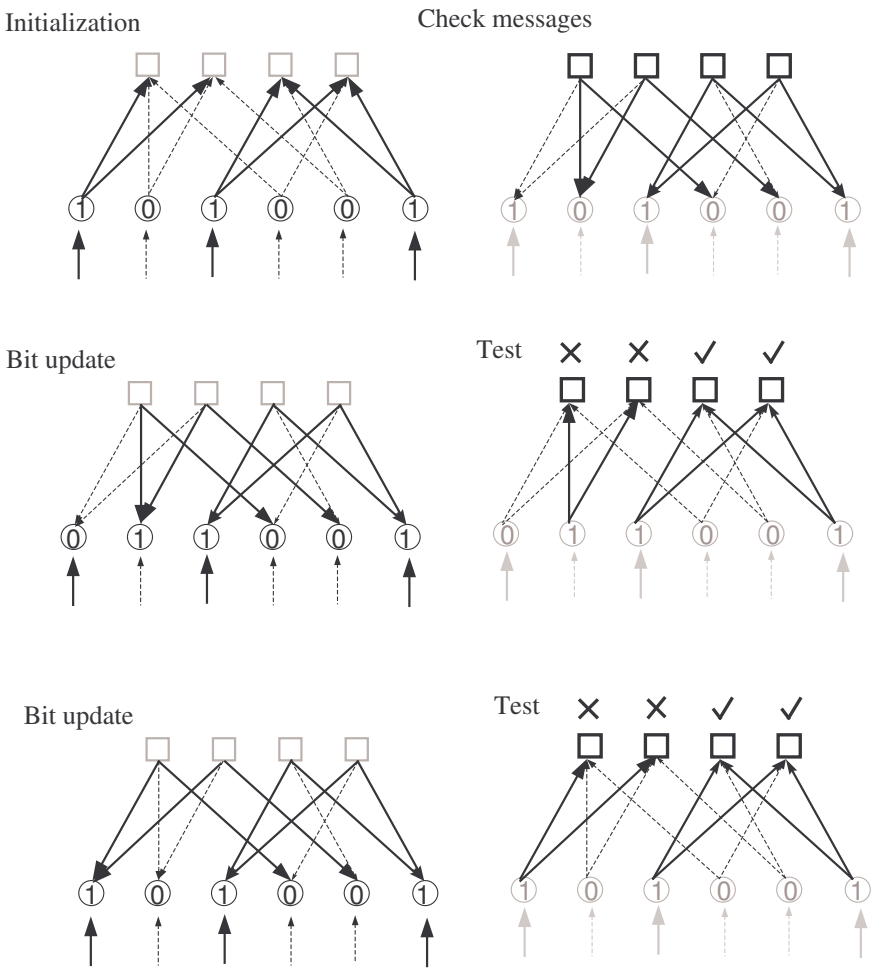

Figure 2.3: Bit-flipping decoding of $y = [1\ 0\ 1\ 0\ 0\ 1]$. Each sub-figure indicates the decision made at each step of the decoding algorithm based on the messages from the previous step. A cross ($\times$) represents that the parity check is not satisfied while a tick ($\checkmark$) indicates that it is satisfied. For the messages, a dashed arrow corresponds to the messages "bit $= 0$" while a solid arrow corresponds to "bit $= 1$".

2.3 Sum-product decoding

The sum-product algorithm is a soft decision message-passing algorithm. It is similar to the bit-flipping algorithm described in the previous section, but with the messages representing each decision (check met, or bit value equal to 1) now probabilities. Whereas bit-flipping decoding accepts an initial hard decision on the received bits as input, the sum-product algorithm is a soft decision algorithm which accepts the probability of each received bit as input.

The input bit probabilities are called the *a priori* probabilities for the received bits because they were known in advance before running the LDPC decoder. The bit probabilities returned by the decoder are called the *a posteriori* probabilities. In the case of sum-product decoding these probabilities are expressed as *log-likelihood ratios*.

For a binary variable x it is easy to find $p(x = 1)$ given $p(x = 0)$, since $p(x = 1) = 1 - p(x = 0)$ and so we only need to store one probability value for x. Log likelihood ratios are used to represent the metrics for a binary variable by a single value:

$$L(x) = \log\left(\frac{p(x=0)}{p(x=1)}\right), \tag{2.1}$$

where we use $\log$ to mean $\log_e$. If $p(x = 0) > p(x = 1)$ then $L(x)$ is positive and the greater the difference between $p(x = 0)$ and $p(x = 1)$, i.e. the more sure we are that $p(x) = 0$, the larger the positive value for $L(x)$. Conversely, if $p(x = 1) > p(x = 0)$ then $L(x)$ is negative and the greater the difference between $p(x = 0)$ and $p(x = 1)$ the larger the negative value for $L(x)$. Thus the sign of $L(x)$ provides the hard decision on x and the magnitude $|L(x)|$ is the reliability of this decision. To translate from log likelihood ratios back to probabilities we note that

$$p(x=1) = \frac{p(x=1)/p(x=0)}{1 + p(x=1)/p(x=0)} = \frac{e^{-L(x)}}{1 + e^{-L(x)}} \tag{2.2}$$

and

$$p(x=0) = \frac{p(x=0)/p(x=1)}{1 + p(x=0)/p(x=1)} = \frac{e^{L(x)}}{1 + e^{L(x)}}. \tag{2.3}$$

The benefit of the logarithmic representation of probabilities is that when probabilities need to be multiplied log-likelihood ratios need only be added, reducing implementation complexity.

The aim of sum-product decoding is to compute the *maximum a posteriori probability* (MAP) for each codeword bit, $P_i = P\{c_i = 1|N\}$, which is the probability that the i-th codeword bit is a 1 conditional on the event N that all parity-check constraints are satisfied. The extra information about bit i received from the parity-checks is called *extrinsic* information for bit i.

The sum-product algorithm iteratively computes an approximation of the MAP value for each code bit. However, the a posteriori probabilities returned by the sum-product decoder are only exact MAP probabilities if the Tanner graph is cycle free. Briefly, the extrinsic information obtained from a parity-check constraint in the first iteration is independent of the a priori probability information for that bit (it does of course depend on the a priori probabilities of

the other codeword bits). The extrinsic information provided to bit i in subsequent iterations remains independent of the original a priori probability for bit i until the original a priori probability is returned back to bit i via a cycle in the Tanner graph. The correlation of the extrinsic information with the original a priori bit probability is what prevents the resulting posteriori probabilities from being exact.

In sum-product decoding the extrinsic message from check node j to bit node i, $E_{j,i}$, is the LLR of the probability that bit i causes parity-check j to be satisfied. The probability that the parity-check equation is satisfied if bit i is a 1 is

$$P_{j,i}^{\text{ext}} = \frac{1}{2} - \frac{1}{2} \prod_{i' \in B_j, i' \neq i} (1 - 2P_{i'}^{\text{int}}), \tag{2.4}$$

where $P_{j,i'}^{\text{int}}$ is the current estimate, available to check j, of the probability that bit i' is a one. The probability that the parity-check equation is satisfied if bit i is a zero is thus $1 - P_{j,i}^{\text{ext}}$. Expressed as a log-likelihood ratio,

$$E_{j,i} \quad = \quad \text{LLR}(P_{j,i}^{\text{ext}}) = \log\left(\frac{1-P_{j,i}^{\text{ext}}}{P_{j,i}^{\text{ext}}}\right), \tag{2.5}$$

and substituting (2.4) gives:

$$E_{j,i} \quad = \quad \log\left(\frac{\frac{1}{2}+\frac{1}{2}\prod_{i' \in B_j, i' \neq i}(1-2P_{i'}^{\text{int}})}{\frac{1}{2}-\frac{1}{2}\prod_{i' \in B_j, i' \neq i}(1-2P_{i'}^{\text{int}})}\right). \tag{2.6}$$

Using the relationship

$$\tanh\left(\frac{1}{2}\log\left(\frac{1-p}{p}\right)\right) = 1 - 2p,$$

gives

$$E_{j,i} \quad = \quad \log\left(\frac{1+\prod_{i' \in B_j, i' \neq i}\tanh(M_{j,i'}/2)}{1-\prod_{i' \in B_j,\, i' \neq i}\tanh(M_{j,i'}/2)}\right) \tag{2.7}$$

where

$$M_{j,i'} = LLR(P_{j,i'}^{\text{int}}) = \log\left(\frac{1 - P_{j,i'}^{\text{int}}}{P_{j,i'}^{\text{int}}}\right).$$

Alternatively, using the relationship

$$2\tanh^{-1}(p) = \log\left(\frac{1+p}{1-p}\right),$$

(2.7) can be equivalently written as:

$$E_{j,i} \quad = \quad 2\tanh^{-1}\left(\prod_{i' \in B_j, i' \neq i}\tanh(M_{j,i'}/2)\right). \tag{2.8}$$

Each bit has access to the input a priori LLR, r_i, and the LLRs from every connected check node. The total LLR of the i-th bit is the sum of these LLRs:

$$L_i = \text{LLR}(P_i^{\text{int}}) = r_i + \sum_{j \in A_i} E_{j,i}. \tag{2.9}$$

However, the messages sent from the bit nodes to the check nodes, $M_{j,i}$, are not the full LLR value for each bit. To avoid sending back to each check node information which it already has, the message from the i-th bit node to the j-th check node is the sum in (2.9) without the component $E_{j,i}$ which was just received from the j-th check node:

$$M_{j,i} = \sum_{j' \in A_i,\, j' \neq j} E_{j',i} + r_i. \tag{2.10}$$

The sum-product algorithm is shown in Algorithm 4. Input is the log likelihood ratios for the a priori message probabilities

$$r_i = \log \frac{p(\mathbf{c}_t = 0)}{p(\mathbf{c}_t = 1)},$$

the parity-check matrix H and the maximum number of allowed iterations, $I_{\max}$. The algorithm outputs the estimated a posteriori bit probabilities of the received bits as log likelihood ratios.

Example 2.5.

Here we repeat Example 2.3 using sum-product instead of bit-flipping decoding. The codeword

$$\mathbf{c} = [0\ 0\ 1\ 0\ 1\ 1],$$

is sent through a BSC with crossover probability $p = 0.2$ and

$$\mathbf{y} = [1\ 0\ 1\ 0\ 1\ 1]$$

is received. Since the channel is binary symmetric the probability that 0 was sent if 1 is received is the probability, p, that a crossover occurred while the probability that 1 was sent if 1 is received is the probability, $1 - p$, that no crossover occurred. Similarly, the probability that 1 was sent if 0 is received is the probability that a crossover occurred while the probability that 0 was sent if 0 is received is the probability that no crossover occurred. Thus the a priori probabilities for the BSC are

$$r_i = \begin{cases} \log \frac{p}{1-p}, & \text{if } y_i = 1, \\ \log \frac{1-p}{p}, & \text{if } y_i = 0. \end{cases}$$

For this channel we have

$$\log \frac{p}{1-p} = \log \frac{0.2}{0.8} = -1.3863,$$

$$\log \frac{1-p}{p} = \log \frac{0.8}{0.2} = 1.3863,$$

and so the a priori log likelihood ratios are

$$\mathbf{r} = [-1.3863, 1.3863, -1.3863, 1.3863, -1.3863, -1.3863].$$

To begin decoding we set the maximum number of iterations to three and pass in H and $\mathbf{r}$. Initialization is

$$M_{j,i} = r_i.$$

Algorithm 4 Sum-Product Decoding

procedure DECODE(**r**)

  $I = 0$ ▷ Initialization
  for $i = 1 : n$ **do**
    for $j = 1 : m$ **do**
      $M_{j,i} = r_i$
    end for
  end for

  repeat
    for $j = 1 : m$ **do** ▷ Step 1: Check messages
      for $i \in B_j$ **do**
        $E_{j,i} = \log\left(\frac{1+\prod_{i' \in B_j, i' \neq i} \tanh(M_{j,i'}/2)}{1-\prod_{i' \in B_j,\, i' \neq i} \tanh(M_{j,i'}/2)}\right)$
      end for
    end for

    for $i = 1 : n$ **do** ▷ Test
      $L_i = \sum_{j \in A_i} E_{j,i} + r_i$
      $z_i = \begin{cases} 1, & L_i \leq 0 \\ 0, & L_i > 0. \end{cases}$
    end for
    if $I = I_{\max}$ or $H\mathbf{z}^T = 0$ **then**
      Finished
    else
      for $i = 1 : n$ **do** ▷ Step 2: Bit messages
        for $j \in A_i$ **do**
          $M_{j,i} = \sum_{j' \in A_i,\, j' \neq j} E_{j',i} + r_i$
        end for
      end for
      $I = I + 1$
    end if
  until Finished

end procedure

The 1-st bit is included in the 1-st and 3-rd checks and so $M_{1,1}$ and $M_{3,1}$ are initialized to r_1:

$$M_{1,1} = r_1 = -1.3863 \quad \text{and} \quad M_{3,1} = r_1 = -1.3863.$$

Repeating for the remaining bits gives:

$$\begin{array}{lll} i = 2: & M_{1,2} = r_2 = 1.3863 & M_{2,2} = r_2 = 1.3863 \\ i = 3: & M_{2,3} = r_3 = -1.3863 & M_{4,3} = r_3 = -1.3863 \\ i = 4: & M_{1,4} = r_4 = 1.3863 & M_{4,4} = r_4 = 1.3863 \\ i = 5: & M_{2,5} = r_5 = -1.3863 & M_{3,5} = r_5 = -1.3863 \\ i = 6: & M_{3,6} = r_6 = -1.3863 & M_{4,6} = r_6 = -1.3863 \end{array}$$

For Step 1 the extrinsic probabilities are calculated. Check one includes the 1-st, 2-nd and 4-th bits and so the extrinsic probability from the 1-st check to the 1-st bit depends on the probabilities of the 2-nd and 4-th bits.

$$\begin{aligned} E_{1,1} &= \log\left(\frac{1+\tanh(M_{1,2}/2)\tanh(M_{1,4}/2)}{1-\tanh(M_{1,2}/2)\tanh(M_{1,4}/2)}\right) \\ &= \log\left(\frac{1+\tanh(1.3863/2)\tanh(1.3863/2)}{1-\tanh(1.3863/2)\tanh(1.3863/2)}\right) \\ &= \log\left(\frac{1+0.6*0.6}{1-0.6*0.6}\right) = 0.7538. \end{aligned}$$

Similarly, the extrinsic probability from the 1-st check to the 2-nd bit depends on the probabilities of the 1-st and 4-th bits

$$\begin{aligned} E_{1,2} &= \log\left(\frac{1+\tanh(M_{1,1}/2)\tanh(M_{1,4}/2)}{1-\tanh(M_{1,1}/2)\tanh(M_{1,4}/2)}\right) \\ &= \log\left(\frac{1+\tanh(-1.3863/2)\tanh(1.3863/2)}{1-\tanh(-1.3863/2)\tanh(1.3863/2)}\right) \\ &= \log\left(\frac{1+-0.6*0.6}{1--0.6*0.6}\right) = -0.7538, \end{aligned}$$

and the extrinsic probability from the 1-st check to the 4-th bit depends on the LLRs sent from the 1-st and 2-nd bits to the 1-st check.

$$\begin{aligned} E_{1,4} &= \log\left(\frac{1+\tanh(M_{1,1}/2)\tanh(M_{1,2}/2)}{1-\tanh(M_{1,1}/2)\tanh(M_{1,2}/2)}\right) \\ &= \log\left(\frac{1+\tanh(-1.3863/2)\tanh(1.3863/2)}{1-\tanh(-1.3863/2)\tanh(1.3863/2)}\right) \\ &= \log\left(\frac{1+-0.6*0.6}{1--0.6*0.6}\right) = -0.7538. \end{aligned}$$

Next, the 2-nd check includes the 2-nd, 3-rd and 5-th bits and so the extrinsic LLRs are:

$$\begin{aligned} E_{2,2} &= \log\left(\frac{1+\tanh(M_{2,3}/2)\tanh(M_{2,5}/2)}{1-\tanh(M_{2,3}/2)\tanh(M_{2,5}/2)}\right) \\ &= \log\left(\frac{1+\tanh(-1.3863/2)\tanh(-1.3863/2)}{1-\tanh(-1.3863/2)\tanh(-1.3863/2)}\right) \\ &= \log\left(\frac{1+-0.6*-0.6}{1--0.6*-0.6}\right) = 0.7538, \\ E_{2,3} &= \log\left(\frac{1+\tanh(M_{2,2}/2)\tanh(M_{2,5}/2)}{1-\tanh(M_{2,2}/2)\tanh(M_{2,5}/2)}\right) \\ &= \log\left(\frac{1+\tanh(+1.3863/2)\tanh(-1.3863/2)}{1-\tanh(+1.3863/2)\tanh(-1.3863/2)}\right) \\ &= \log\left(\frac{1+0.6*-0.6}{1-0.6*-0.6}\right) = -0.7538, \\ E_{2,5} &= \log\left(\frac{1+\tanh(M_{2,2}/2)\tanh(M_{2,3}/2)}{1-\tanh(M_{2,2}/2)\tanh(M_{2,3}/2)}\right) \\ &= \log\left(\frac{1+\tanh(+1.3863/2)\tanh(-1.3863/2)}{1-\tanh(+1.3863/2)\tanh(-1.3863/2)}\right) \\ &= \log\left(\frac{1+0.6*-0.6}{1-0.6*-0.6}\right) = -0.7538. \end{aligned}$$

Repeating for all checks gives the extrinsic LLRs:

$$E = \begin{bmatrix} 0.7538 & -0.7538 & . & -0.7538 & . & . \\ . & 0.7538 & -0.7538 & . & -0.7538 & . \\ 0.7538 & . & . & . & 0.7538 & 0.7538 \\ . & . & -0.7538 & 0.7538 & . & -0.7538 \end{bmatrix}.$$

To save space the extrinsic LLRs are given in matrix form where the (j,i)-th entry of E holds $E_{j,i}$. A '.' entry in E indicates that an LLR does not exist for that i and j.

To test the intrinsic and extrinsic probabilities for each bit are combined. The 1-st bit has extrinsic LLRs from the 1-st and 3-rd checks and an intrinsic LLR from the channel. The total LLR for bit one is their sum:

$$L_1 = r_1 + E_{1,1} + E_{3,1} = -1.3863 + 0.7538 + 0.7538 = 0.1213.$$

Thus even though the LLR from the channel is negative, indicating that the bit is a one, both of the extrinsic LLRs are positive indicating that the bit is zero. The extrinsic LLRs are strong enough that the total LLR is positive and so the decision on bit one has effectively been changed. Repeating for bits two to six gives:

$$\begin{aligned} L_2 &= r_2 + E_{1,2} + E_{2,2} = 1.3863 \\ L_3 &= r_3 + E_{2,3} + E_{4,3} = -2.8938 \\ L_4 &= r_4 + E_{1,4} + E_{4,4} = 1.3863 \\ L_5 &= r_5 + E_{2,5} + E_{3,5} = -1.3863 \\ L_6 &= r_6 + E_{3,6} + E_{4,6} = -1.3863 \end{aligned}$$

The hard decision on the received bits is given by the sign of the LLRs,

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

To check if $\mathbf{z}$ is a valid codeword

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

Since $\mathbf{s}$ is zero $\mathbf{z}$ is a valid codeword, and the decoding stops, returning $\mathbf{z}$ as the decoded word.

Example 2.6.

Using the LDPC code from Example 1.12, the codeword

$$\mathbf{c} = [0\ 0\ 1\ 0\ 1\ 1]$$

is sent though a BPSK AWGN channel with $E_S/N_0 = 1.25$ (or 0.9691dB) and the received signal is

$$\mathbf{y} = [-0.1\ \ 0.5\ \ -0.8\ \ 1.0\ \ -0.7\ \ 0.5].$$

(If a hard decision is made without decoding, there are two bits in error in this received vector, the 1-st and 6-th bits.) For an AWGN channel the a priori LLRs are given by

$$r_i = 4y_i \frac{E_S}{N_0}$$

and so we have

$$\mathbf{r} = [-0.5 \;\; 2.5 \;\; -4.0 \;\; 5.0 \;\; -3.5 \;\; 2.5]$$

as input to the sum-product decoder.

Iteration 1

$$\mathbf{r} = \begin{bmatrix} -0.5 & 2.5 & -4.0 & 5.0 & -3.5 & 2.5 \end{bmatrix}$$

$$E = \begin{bmatrix} 2.4217 & -0.4930 & . & -0.4217 & . & . \\ \cdot & 3.0265 & -2.1892 & . & -2.3001 & \cdot \\ -2.1892 & . & . & . & -0.4217 & 0.4696 \\ \cdot & . & 2.4217 & -2.3001 & . & -3.6869 \end{bmatrix}$$

$$L = \begin{bmatrix} -0.2676 & 5.0334 & -3.7676 & 2.2783 & -6.2217 & -0.7173 \end{bmatrix}$$

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

$$H\mathbf{z}^T = \begin{bmatrix} 1 & 0 & 1 & 0 \end{bmatrix}^T \quad \Rightarrow \quad \text{Continue}$$

$$M = \begin{bmatrix} -2.6892 & 5.5265 & \cdot & 2.6999 & . & \cdot \\ \cdot & 2.0070 & -1.5783 & . & -3.9217 & \cdot \\ 1.9217 & . & \cdot & \cdot & -5.8001 & -1.1869 \\ \cdot & . & -6.1892 & 4.5783 & . & 2.9696 \end{bmatrix}$$

Iteration 2

$$E = \begin{bmatrix} 2.6426 & -2.0060 & . & -2.6326 & . & . \\ \cdot & 1.4907 & -1.8721 & . & -1.1041 & . \\ 1.1779 & \cdot & \cdot & . & -0.8388 & -1.9016 \\ \cdot & . & 2.7877 & -2.9305 & \cdot & -4.3963 \end{bmatrix}$$

$$L = \begin{bmatrix} 3.3206 & 1.9848 & -3.0845 & -0.5630 & -5.4429 & -3.7979 \end{bmatrix}$$

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

$$H\mathbf{z}^T = \begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix}^T \quad \Rightarrow \quad \text{Continue}$$

$$M = \begin{bmatrix} 0.6779 & 3.9907 & \cdot & 2.0695 & . & \cdot \\ \cdot & 0.4940 & -1.2123 & . & -4.3388 & \cdot \\ 2.1426 & \cdot & \cdot & \cdot & -4.6041 & -1.8963 \\ \cdot & \cdot & -5.8721 & 2.3674 & \cdot & 0.5984 \end{bmatrix}$$

Iteration 3

$$E = \begin{bmatrix} 1.9352 & 0.5180 & . & 0.6515 & . & \cdot \\ \cdot & 1.1733 & -0.4808 & . & -0.2637 & \cdot \\ 1.8332 & \cdot & \cdot & \cdot & -1.3362 & -2.0620 \\ \cdot & \cdot & 0.4912 & -0.5948 & \cdot & -2.3381 \end{bmatrix}$$

$$L = \begin{bmatrix} 3.2684 & 4.1912 & -3.9896 & 5.0567 & -5.0999 & -1.9001 \end{bmatrix}$$

$$\mathbf{z} = [\ 0\ 0\ 1\ 0\ 1\ 1\ ]$$

$$H\mathbf{z}^T = [\ 0\ 0\ 0\ 0\ ]^T \quad \Rightarrow \quad \text{Terminate}$$

The sum-product decoder converges to the correct codeword after three iterations.

The sum-product algorithm can be modified to reduce the implementation complexity of the decoder. This can be done by altering equation 2.8

$$E_{j,i} = 2\tanh^{-1}\left(\prod_{i' \in B_j, i' \neq i} \tanh(M_{j,i'}/2)\right),$$

so as to replace the product term by a sum. For simplicity we will write

$$\prod_{i'} \equiv \prod_{i' \in B_j, i' \neq i}$$

in the remainder of this section.

Firstly $M_{j,i'}$ can be factored as

$$M_{j,i'} = \alpha_{j,i'}\beta_{j,i'}$$

where

$$\begin{aligned} \alpha_{j,i'} &= \text{sign}(M_{j,i'}), \\ \beta_{j,i'} &= |M_{j,i'}|. \end{aligned} \tag{2.11}$$

Using this notation we have that

$$\prod_{i'} \tanh(M_{j,i'}/2) = \prod_{i'} \alpha_{j,i'} \prod_{i'} \tanh(\beta_{j,i'}/2).$$

Then equation 2.8 becomes

$$\begin{aligned} E_{j,i} &= 2\tanh^{-1}\left(\prod_{i'} \alpha_{j,i'} \prod_{i'} \tanh(\beta_{j,i'}/2)\right) \\ &= \left(\prod_{i'} \alpha_{j,i'}\right) 2\tanh^{-1}\left(\prod_{i'} \tanh(\beta_{j,i'}/2)\right) \end{aligned} \tag{2.12}$$

Equation 2.12 can now be re-arranged to replace the product by a sum:

$$\begin{aligned} E_{j,i} &= \left(\prod_{i'} \alpha_{j,i'}\right) 2\tanh^{-1} \log^{-1} \log\left(\prod_{i'} \tanh(\beta_{j,i'}/2)\right) \\ &= \left(\prod_{i'} \alpha_{j,i'}\right) 2\tanh^{-1} \log^{-1}\left(\sum_{i'} \log\tanh(\beta_{j,i'}/2)\right). \end{aligned} \tag{2.13}$$

Next, we define

$$\phi(x) = -\log\tanh(\frac{x}{2}) = \log\frac{e^x + 1}{e^x - 1}$$

and note that since

$$\phi(\phi(x)) = \log \frac{e^{\phi(x)}+1}{e^{\phi(x)}-1} = x.$$

we have $\phi^{-1} = \phi$. Finally equation 2.13 becomes

$$E_{j,i} = \left(\prod_{i'} \alpha_{j,i'} \right) \phi \left(\sum_{i'} \phi(\beta_{j,i'}) \right). \tag{2.14}$$

The product of the signs can be calculated by using modulo 2 addition of the hard decisions on each $M_{j,i'}$ while the function ϕ can be easily implemented using a lookup table.

Alternatively, the *min-sum* algorithm, simplifies the calculation of (2.7) by recognizing that the term corresponding to the smallest $M_{j,i'}$ dominates the product term and so the product can be approximated by a minimum:

$$E_{j,i} \approx \left(\prod_{i'} \operatorname{sign}(M_{j,i'}) \right) \underbrace{\operatorname{Min}}_{i'} \left| M_{j,i'} \right|.$$

Again, the product of the signs can be calculated by using modulo 2 addition of the hard decisions on each $M_{j,i'}$ and so the resulting min-sum algorithm thus requires calculation of only minimums and additions.

2.4 Bibliographic notes

Message-passing decoding algorithms for LDPC codes were first introduced by Gallager in his 1962 thesis [1]. In the early 1960s, however, limited computing resources prevented Gallager from demonstrating the capabilities of message-passing decoders for blocklengths longer than around 500 bits, and for over 30 years his work was ignored by all but a handful of researchers. It was only re-discovered by several researchers [7] in the wake of turbo decoding [12], which itself has subsequently been recognized as an instance of the sum-product algorithm.

Topic 3: Density Evolution

The subject of this topic is to analyze the performance of message-passing decoders and understand how the choice of LDPC code will impact on this performance. Ideally, for a given Tanner graph $\mathcal{T}$ we would like to know for which channel noise levels the message-passing decoder will be able to correct the errors and for which it won't. Unfortunately, this is still an open problem, but what is possible to determine is how an *ensemble* of Tanner graphs is likely to behave, if the channel is memoryless and under the assumption that the Tanner graphs are all cycle free. To do this the evolution of probability density functions are tracked through the message-passing algorithm, a process called *density-evolution*.

Density evolution can be used to find the maximum level of channel noise which is likely to be corrected by a particular ensemble using the message-passing algorithm, called the *threshold* for that ensemble. This then enables the code designer to search for the ensemble with the best threshold from which to choose a specific LDPC code.

3.1 Density evolution on the BEC

Recall from Algorithm 2 that for message passing decoding on the BEC a parity-check equation can correct an erased bit if that bit was the only erased bit in the parity-check equation. Here we make the assumption that the decoding algorithm is passing messages down through the layers of a Tanner graph which is a tree. In this case the bit-to-check message to check node in a lower level of the graph is determined by the check-to-bit messages from all the incoming edges in the level above.

3.1.1 Regular LDPC codes

Given an ensemble $\mathcal{T}(w_c, w_r)$, which consists of all regular LDPC Tanner graphs with bit nodes of degree w_c and check nodes of degree w_r, we want to know how the message-passing decoder will perform on the binary erasure channel using codes from this ensemble.

For message-passing decoding on the BEC, the messages hold either the current value of the bit, which can be '1'or '0' if known or 'x' if the bit value is not known. We define q_l to be the probability that at iteration l a check to bit message is an 'x' and p_l to be the probability that at iteration l a bit to check message is an 'x' (i.e. p_l is the probability that a codeword bit is still erased at iteration l).

The check to bit message on an edge is 'x' if one or more of the incoming messages on the other $(w_r - 1)$ edges into that check node is an 'x'. To

calculate the probability, q_l, that a check to bit message is 'x' at iteration l we make the assumption that all of the incoming messages are independent of one another. That is, we are assuming firstly that the channel is memoryless, so that none of the original bit probabilities were correlated, and secondly that there are no cycles in the Tanner graphs of length $2l$ or less, as a cycle will cause the messages to become correlated. With this assumption, the probability that none of the other $w_r - 1$ incoming message to the check node is 'x' is simply the product of the probabilities, $(1-p_l)$, that each individual message is not 'x'. So the probability that one or more of the other incoming messages are 'x' is one minus this:

$$q_l = 1 - (1 - p_l)^{(w_r - 1)} \tag{3.1}$$

At iteration l the bit to check message will be 'x' if the original message from the channel was an erasure, which occurs with probability ε, and all of the incoming messages at iteration $l-1$ are erasures, which each have probability q_l. Again we make the assumption that all of the incoming messages are independent of one another, and so the probability that the bit to check message is an 'x' is the product of the probabilities that the other $w_c - 1$ incoming messages to the bit node, and the original message from the channel, were erased.

$$p_l = \varepsilon \left(q_{l-1}\right)^{(w_c - 1)}. \tag{3.2}$$

Substituting for q_{l-1} from (3.1) gives

$$p_l = \varepsilon \left(1 - (1 - p_{l-1})^{(w_r - 1)}\right)^{(w_c - 1)}. \tag{3.3}$$

Prior to decoding the value of p_0 is the probability that the channel erased a codeword bit:

$$p_0 = \varepsilon.$$

Thus for a (w_c, w_r)-regular ensemble

$$p_0 = \varepsilon, \qquad p_l = \varepsilon \left(1 - (1 - p_{l-1})^{(w_r - 1)}\right)^{(w_c - 1)} \tag{3.4}$$

The recursion in (3.4) describes how the erasure probability of message-passing decoding evolves as a function of the iteration number l for (w_c, w_r)-regular LDPC codes. Applying this recursion we can determine for which erasure probabilities the message-passing decoder is likely to correct the erasures.

Example 3.1.

A code from the (3,6)-regular ensemble is to be transmitted on a binary erasure channel with erasure probability $\varepsilon = 0.3$ and decoded with the message-passing algorithm. The probability that a codeword bit will remain erased after l iterations of message-passing decoding (if the code Tanner graph is cycle free) is given by the recursion:

$$p_0 = 0.3, \qquad p_l = p_0 \left(1 - (1 - p_{l-1})^5\right)^2.$$

Applying this recursion for 7 iterations gives the sequence of bit erasure probabilities,

$$p_0 = 0.3000, \quad p_1 = 0.2076, \quad p_2 = 0.1419, \quad p_3 = 0.0858,$$

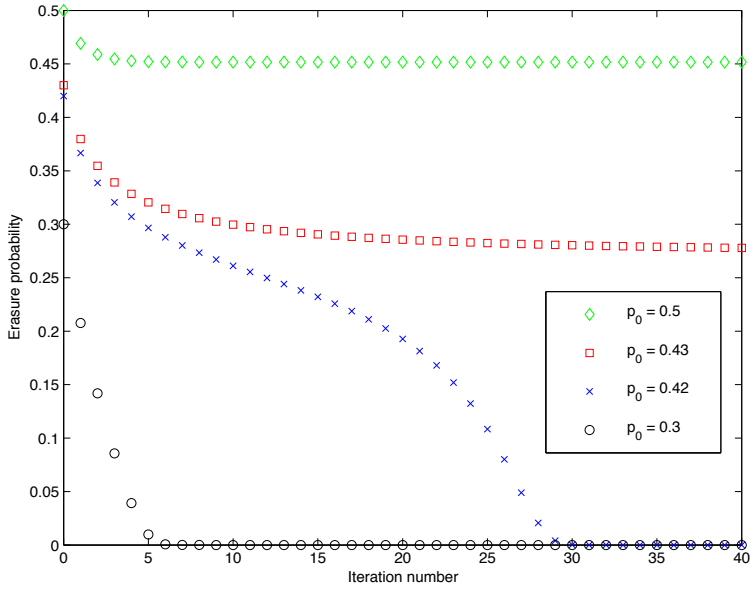

Figure 3.1: The erasure probabilities calculated in Example 3.2.

$$p_4 = 0.0392, \quad p_5 = 0.0098, \quad p_6 = 0.0007, \quad p_7 = 0.0000.$$

Thus the erasure probability in a codeword from a 4-cycle free (3,6)-regular LDPC code transmitted on a BEC with erasure probability 0.3 will approach zero after seven iterations of message-passing decoding.

Example 3.2.

Extending Example 3.1 we would like to know which erasure probabilities the codes from a (3,6)-regular ensemble are likely to be able to correct. Applying the recursion (3.3) to a range of channel erasure probabilities we see, in Fig. 3.1, that for values of $\varepsilon \geq 0.43$ the probability of remaining erasures does not decrease to zero even as l gets very large, whereas, for values of $\varepsilon \leq 0.42$ the probability of error does approach zero as $l \to \infty$. The transition value of ε, between these two outcomes is the called the *threshold* of the (3,6)-regular ensemble, a term we make more precise in the following. Again applying the recursion (3.3), Fig. 3.2 demonstrates that the threshold for a (3,6)-regular ensemble on the binary erasure channel is between 0.4293 and 0.4294.

3.1.2 Irregular LDPC codes

Recall that an irregular parity-check matrix has columns and rows with varying weights (respectively bit nodes and check nodes with varying degrees). We designated the fraction of columns of weight i by v_i and the fraction of rows of weight i by h_i.

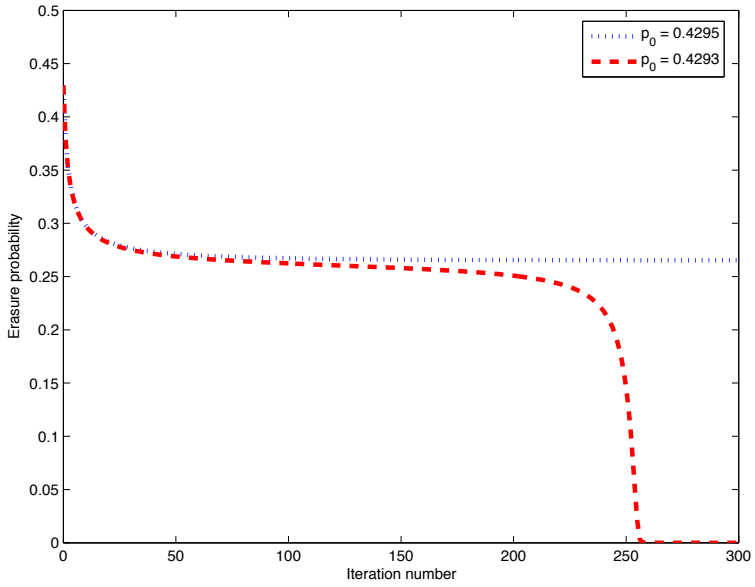

Figure 3.2: The erasure probabilities calculated in Example 3.2.

To derive density evolution for irregular LDPC codes an alternative characterization of the degree distribution, from the perspective of Tanner graph edges, is used. The fraction of edges which are connected to degree-i bit nodes is denoted λ_i, and the fraction of edges which are connected to degree-i check nodes, is denoted ρ_i. By definition:

$$\sum_i \lambda_i = 1 \tag{3.5}$$

and

$$\sum_i \rho_i = 1. \tag{3.6}$$

The functions

$$\lambda(x) = \lambda_2 x + \lambda_3 x^2 + \cdots + \lambda_i x^{i-1} + \cdots \tag{3.7}$$

$$\rho(x) = \rho_2 x + \rho_3 x^2 + \cdots + \rho_i x^{i-1} + \cdots \tag{3.8}$$

are defined to describe the degree distributions. Translating between node degrees and edge degrees:

$$v_i = \frac{\lambda_i / i}{\sum_j \lambda_j / j},$$

$$h_i = \frac{\rho_i / i}{\sum_j \rho_j / j}.$$

From (3.1) we know that, at the l-th iteration of message-passing decoding, the probability that a check to bit message is 'x', if all the incoming messages are independent, is

$$q_l = 1 - (1 - p_l)^{(w_r - 1)},$$

for an edge connected to a degree w_r check node. For an irregular Tanner graph the probability that an edge is connected to a degree w_r check node is ρ_{w_r}. Thus averaging over all the edges in an irregular Tanner graph gives the average probability that a check to bit message is in error:

$$q_l = \sum_i \rho_i \left(1 - (1-p_l)^{(i-1)}\right) = 1 - \sum_i \rho_i \left(1-p_l\right)^{(i-1)}.$$

Using the definition of $\rho(x)$ in (3.8), this becomes

$$q_l = 1 - \rho\left(1-p_l\right).$$

From (3.2) we know that the probability that a bit to check message is 'x', at the l-th iteration of message-passing decoding if all incoming messages are independent, is

$$p_l = \varepsilon \left(q_{l-1}\right)^{(w_c-1)}$$

for an edge is connected to a degree w_c bit node. For an irregular Tanner graph the probability that an edge is connected to a degree w_c bit node is λ_{w_c}. Thus averaging over all the edges in the Tanner graph gives the average probability that a bit to check message is in error:

$$p_l = \varepsilon \sum_i \lambda_i \left(q_{l-1}\right)^{(i-1)}.$$

Using the definition of $\lambda(x)$ in (3.7), this is equivalent to

$$p_l = \varepsilon\lambda\left(q_{l-1}\right).$$

Finally, substituting for q_{l-1} we have

$$p_l = \varepsilon\lambda\left(1 - \rho\left(1-p_{l-1}\right)\right).$$

Prior to decoding the value of p_0 is the probability that the channel erased a codeword bit:

$$p_0 = \varepsilon,$$

and so for irregular LDPC codes we have the recursion:

$$p_0 = \varepsilon, \qquad p_l = p_0\lambda\left(1 - \rho\left(1-p_{l-1}\right)\right). \tag{3.9}$$

3.1.3 Threshold

The aim of density evolution is to determine for which channel erasure probabilities, ε, the message-passing decoder is likely to correct all of the erased bits. Using (3.9) we have a means of approximating this as an average over all LDPC Tanner graphs with a given degree distribution λ, ρ, by assuming that the graphs are cycle free.

To examine the influence of ε on p_l we define the function

$$f(p,\varepsilon) = \varepsilon\lambda\left(1 - \rho\left(1-p\right)\right).$$

The erasure probability at iteration l is then

$$p_l(\varepsilon) = f(p_{l-1},\varepsilon)$$

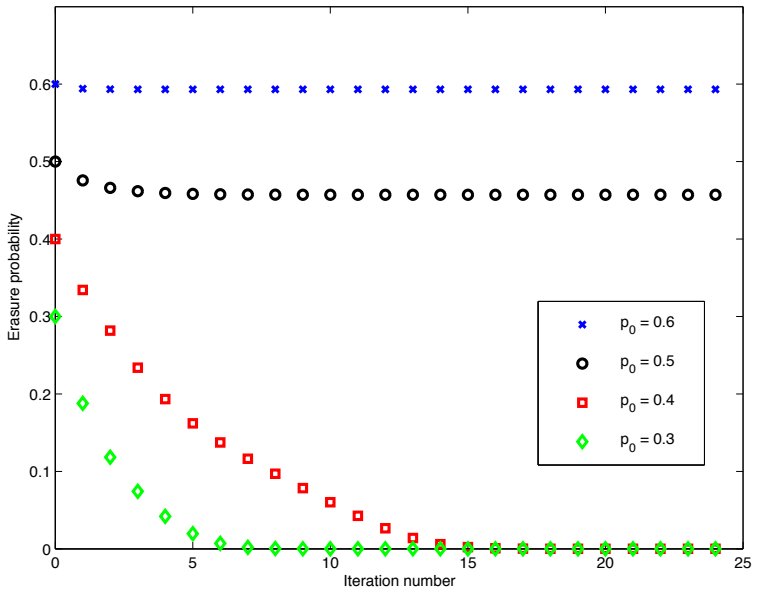

Figure 3.3: The erasure probabilities calculated in Example 3.3.

where p and ε are probabilities, and so can take values between 0 and 1. Here, $f(p, \varepsilon)$ is a strictly increasing function in p for $\varepsilon > 0$. Thus if $p_l > p_{l-1}$ then

$$p_{l+1} = f(p_l, \varepsilon) \geq f(p_{l-1}, \varepsilon) = p_l,$$

for $\varepsilon \in [0, 1]$, so $p_l(\varepsilon)$ is a monotone sequence which is lower bounded at $p = 0$ by

$$f(0, \varepsilon) = \varepsilon\lambda\left(1 - \rho\left(1\right)\right) = \varepsilon\lambda\left(1 - 1\right) = 0$$

and upper bounded at $p = 1$ by

$$f(1, \varepsilon) = \varepsilon\lambda\left(1 - \rho\left(1 - 1\right)\right) = \varepsilon\lambda\left(1 - 0\right) = \varepsilon.$$

Since $f(p, \varepsilon)$ is a strictly increasing function in p

$$0 \leq f(p, \varepsilon) \leq \varepsilon,$$

for all $p \in [0, 1]$ and $\varepsilon \in [0, 1]$. Thus p_l converges to an element $p_\infty \in [0, \varepsilon]$. Further, for a degree distribution pair (λ, ρ), and an $\varepsilon \in [0, 1]$, it can be proven that if $p_l(\varepsilon) \to 0$ then $p_l(\varepsilon') \to 0$ for all $\varepsilon < \varepsilon'$. Indeed, there is a value ε^* called the *threshold* such that for values of ε below ε^*, p_l approaches zero as the number of iterations goes to infinity while for values of of ε above ε^* it does not. The threshold, ε^*, for (λ, ρ) is defined as the supremum of ε for which $p_l(\varepsilon) \to 0$:

$$\varepsilon^*(\lambda, \rho) = \sup\{\varepsilon \in [0, 1] : p_l(\varepsilon)_{l \to \infty} \to 0\}.$$

Example 3.3.

We wish to find the threshold of an irregular LDPC ensemble with degree distributions

$$\lambda(x) = 0.1x + 0.4x^2 + 0.5x^{19}$$

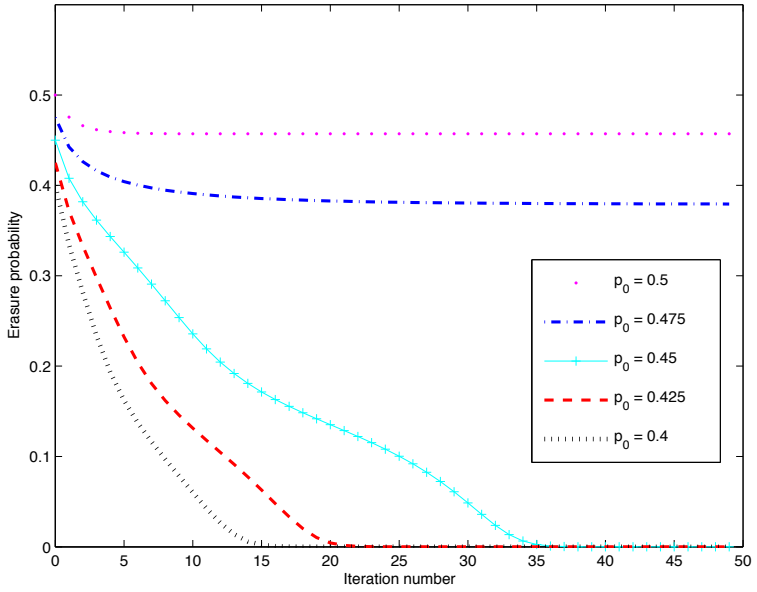

Figure 3.4: The erasure probabilities calculated in Example 3.3.

and

$$\rho(x) = 0.5x^7 + 0.5x^8.$$

This code has rate

$$1 - \frac{\sum_i \lambda_i/i}{\sum_i \rho_i/i} \approx 0.5.$$

To find the threshold we apply the recursion from (3.9) over a wide range of different channel erasure probabilities. We see in Fig. 3.3, that for values of ε 0.5 and above the probability of remaining erasures does not decrease to zero even as l gets very large, whereas, for values of ε of 0.4 and below the probability of error does go to zero. To close in on the threshold further we thus apply the recursions to channel erasure probabilities between these values. Fig. 3.4, shows that for values of $\varepsilon \geq 0.475$ the probability of remaining erasures does not decrease to zero even as l gets very large, whereas, for values of $\varepsilon \leq 0.45$ the probability of error does go to zero. To close in on the threshold even further we now apply the recursions to channel erasure probabilities between these values. Fig. 3.5, shows that for values of $\varepsilon \geq 0.475$ the probability of remaining erasures does not decrease to zero even as l gets very large, whereas, for values of $\varepsilon \leq 0.465$ the probability of error does go to zero. For $\varepsilon = 0.47$ we would need to consider a larger number of decoder iterations to determine if the decoder would converge to zero. We can conclude, however, that the threshold for this ensemble is an erasure probability between 0.465 and 0.475.

3.1.4 Stability

The recursion in (3.9) quickly results in very high order polynomials as the iteration number is increased. However, to understand its behavior when p_l is

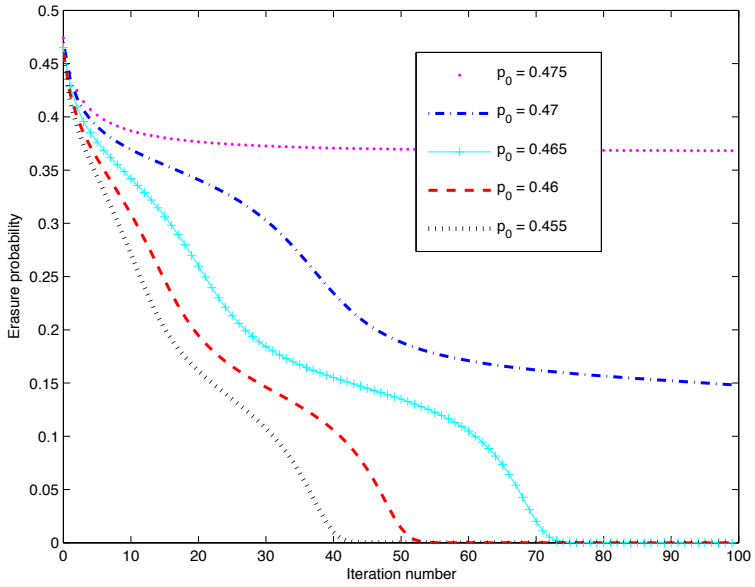

Figure 3.5: The erasure probabilities calculated in Example 3.3.

small we can approximate it by a Taylor series expansion of the right hand side around 0. i.e.

$$p_l = f(p_{l-1}, \varepsilon) \approx f'(p, \varepsilon) p_{l-1}. \tag{3.10}$$

A function $f(x) = g(h(x))$ has a derivative with respect to x given by

$$\frac{df}{dx} = \frac{dg}{dh}\frac{dh}{dx}.$$

Thus for

$$f(p, \varepsilon) = \varepsilon\lambda(h(p)) \quad \text{where} \quad h(p) = 1 - \rho(1-p)$$

the derivative with respect to p is

$$\frac{df(p,\varepsilon)}{dp} = \frac{d\lambda}{dh}\frac{dh}{dp}.$$

Evaluating this derivative at $p = 0$ we have that

$$h(p=0) = 1 - \rho(1) = 0$$

and so

$$\left.\frac{d\lambda}{dh}\right|_{p=0} = \left.\frac{d\lambda}{dh}\right|_{h=0} = \lambda_2 + 2\lambda_3 h + \cdots + (i-1)\lambda_i h^{(i-2)} + \cdots \Big|_{h=0} = \lambda_2,$$

and

$$\left.\frac{dh}{dp}\right|_{p=0} = \left.\frac{d(1-\rho(1-p))}{dp}\right|_{(1-p)=1} = \rho'(1).$$

Substituting back into (3.10),

$$p_l \approx \varepsilon\lambda_2\rho'(1)p_{l-1}, \tag{3.11}$$

as $p_l \to 0$.

For p_l to converge to zero as $l \to \infty$, requires $p_l < p_{l-1}$, and so, from (3.11), requires:

$$\varepsilon \lambda_2 \rho'(1) < 1. \tag{3.12}$$

Thus for a degree distribution pair (λ,ρ) to converge to zero on a binary erasure channel with erasure probability ε, λ_2 is upper bounded by

$$\lambda_2 < \frac{1}{\varepsilon \rho'(1)}. \tag{3.13}$$

Equation 3.13 is often called the *stability constraint* of density evolution.

3.2 Density evolution on general memoryless channels

For message-passing decoding on general memoryless channels, the bit to check messages are the log likelihood ratios (LLRs) of the probabilities that a given bit is '1' or '0'. As these LLR values are continuous, the probability that a message is a particular LLR value is described by a probability density function (pdf).

Recall that the LLR of a random variable x is

$$L(x) = \log\left(\frac{p(x=0)}{p(x=1)}\right),$$

and so $L(x)$ will be positive if $p(x=0) > p(x=1)$ and negative otherwise. Consequently the probability that the corresponding codeword bit is a '1' is the probability that the LLR is negative.

Example 3.4.

Fig. 3.6 shows a gaussian pdf for $\mathfrak{p}(r)$. The probability that the bit is a '1' is given by the shaded area under the curve.

To analyze the evolution of these pdfs in the message-passing decoder we define $\mathfrak{p}(M_l)$ to be the probability density function for a bit to check message at iteration l and $\mathfrak{p}(E_l)$ to be the probability density function for a check to bit messages at iteration l and $\mathfrak{p}(r)$ to be the probability density function for the LLR of the received signal.

Again we make the assumption that all of the incoming messages are independent of one another. That is, we are assuming firstly that the channel is memoryless, so that none of the original bit probabilities were correlated, and secondly that there are no cycles in the Tanner graphs of length $2l$ or less, as a cycle will cause the messages to become correlated.

The outgoing message at a bit node is the sum of the incoming LLRs on the other edges into that node (2.10):

$$M_{j,i} = \sum_{j' \in A_i,\, j' \neq j} E_{j',i} + r_i.$$

Since the incoming messages are independent, the pdf of the random variable formed from this summation can be obtained by the convolution [13, eqn 6.39]

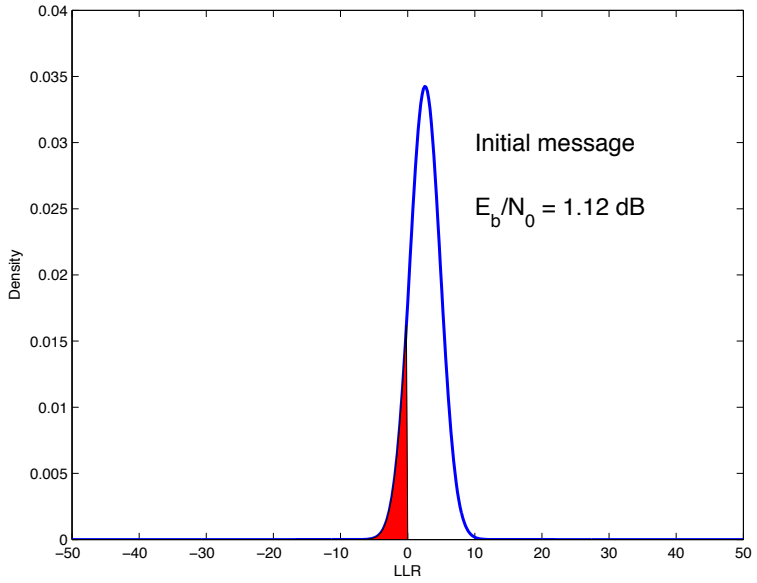

Figure 3.6: The probability density function for additive white Gaussian noise. See Example 3.4.

of the pdfs of the $w_c - 1$ incoming messages from the check nodes and the pdf of the incoming message from the channel:

$$\mathfrak{p}_M = \mathfrak{p}(r) \otimes \mathfrak{p}(E_l)^{\otimes(w_c-1)}.$$

Averaging over the bit degree distribution, $\lambda(x)$:

$$\mathfrak{p}(M_l) = \mathfrak{p}(r) \otimes \sum_i \lambda_i \mathfrak{p}(E_l)^{\otimes(i-1)} = \mathfrak{p}(r) \otimes \lambda^{\otimes}(\mathfrak{p}(E_l)).$$

The convolution operation can be evaluated numerically using FFTs.

The function to be evaluated at each check node is (2.7):

$$E_{j,i} = \log\left(\frac{1+\prod_{i'\in B_j, i'\neq i}\tanh(M_{j,i'}/2)}{1-\prod_{i'\in B_j,\, i'\neq i}\tanh(M_{j,i'}/2)}\right)$$

where

$$\tanh(x/2) = \log\frac{e^x-1}{e^x+1}.$$

Thus to sum over two messages x and y requires the calculation of the probability density function of

$$\begin{aligned} f(x,y) &= \log\tfrac{1+\tanh(x/2)\tanh(y/2)}{1-\tanh(x/2)\tanh(y/2)} = \log\tfrac{(e^x+1)(e^y+1)+(e^x-1)(e^y-1)}{(e^x+1)(e^y+1)-(e^x-1)(e^y-1)} \\ &= -\log\tfrac{e^x+e^y}{1+e^{x+y}}. \end{aligned} \tag{3.14}$$

A method to find the pdf of a function of two random variables is given in [13, eqn 6.36]. Briefly, given two random variables x and y and the function

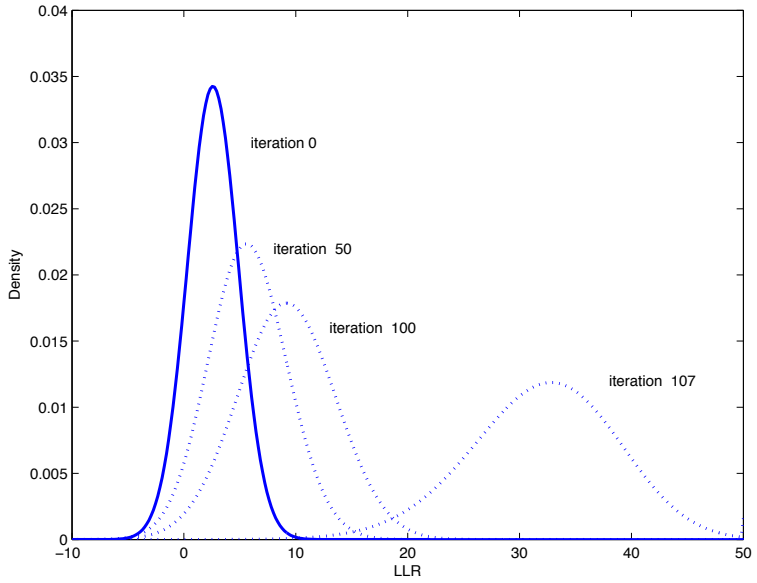

Figure 3.7: The evolution of probability density functions with iteration number in density evolution. See Example 3.5.

$z = f(x, y)$ the density of z can be found as follows:

$$f(z)dz = \int\int_{\Delta D_z} f(x, y)dxdy$$

where D_z is the region of the xy plane such that $z < g(x, y) < z + dz$.

To apply density evolution on general channels it is assumed that the original codeword was all zeros. Consequently the probability that the bit is in error is the probability that the LLR is negative.

Example 3.5.

Fig. 3.7 shows the evolution of $\mathsf{q}(E_l)$ for a (3,6)-regular ensemble on an AWGN channel with signal-to-noise ratio (E_b/N_0) of 1.12. On an AWGN channel the pdf of the original received signal will be Gaussian with variance σ, reflecting the pdf of the noise. As the iteration number is increased the area under the curve for negative LLRs decreases and so the probability of error is decreasing.

Although the pdfs start as gaussian, the result of the convolution of gaussian pdfs is not gaussian except in the limit. However an approximation for density evolution on the AWGN channel assumes that the pdfs do in fact remain gaussian. Since a Gaussian pdf is completely described by its mean and variance this approximation greatly simplifies the application of density evolution as only the mean and variance are tracked through the message-passing decoding and not the entire pdf.

It can be shown (we don't do it here see reference [14]) that the sum-product decoding algorithm preserves the order implied by degradation of the channel. i.e. as the channel improves the performance of the sum-product decoder will

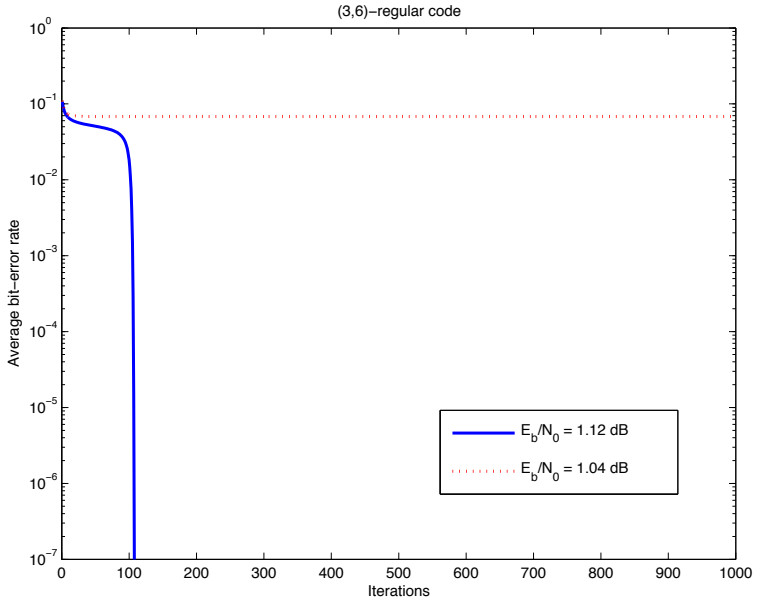

Figure 3.8: Calculating the threshold of a (3,6)-regular ensemble on an AWGN channel. Example 3.6.

also improve. For example, on a AWGN channel with variance σ for all $\sigma' < \sigma$ the expected bit error probability of sum-product decoding, P_{SP}, satisfies $P_{SP}(\sigma') < P_{SP}(\sigma)$.

The threshold of a given degree distribution for sum-product decoding is again the supremum of the channel noise values for which the probability of decoding error goes to zero as the iteration number is allowed to go to infinity. For an AWGN channel with variance σ the threshold is denoted σ^*:

$$\sigma^* = \sup\{\sigma : P_{SP}(\sigma)_{l\to\infty} \to 0\}$$

Example 3.6.

We would like to know at which AWGN signal-to-noise ratios the codes from a (3,6)-regular ensemble are likely to be able to correct the noise. Applying density evolution for different channel erasure probabilities we see, in Fig. 3.8, that the threshold is an E_b/N_0 between 1.04 and 1.12.

3.3 Choosing the degree distribution

We have seen that the threshold of an ensemble of codes with a given degree distribution can be found using density evolution. The question for code designers is then which degree distributions will produce the best threshold.

Generally, the more irregular the bit degree distribution the better. The capacity approaching LDPC codes are both very long and very irregular. The famous LDPC ensemble with threshold 0.0045 dB from the Shannon limit has

a codeword length of 10^7 bits with node degrees varying from 2 to 8000. The subset of the bits nodes with very high degree will very quickly converge to their solution, and once correct will pass high LLR values to their many connected nodes. Since the overall density of H needs to be low, a large proportion of degree-2 bit nodes are also required to reduce the average node degree. Thus a degree distribution with a good threshold will contain a few very high degree bit nodes, many degree two nodes, but no more than allowed for by stability, and some nodes with degrees in between these. Irregularity in the check node degrees is not as essential and generally one or two check degrees, chosen to achieve the required average row weight is sufficient.

Trying every possible distribution that fits this general pattern is of course not practical and so optimization techniques are used to find the best degree distribution subject to the desired constraints. Optimizing over the density evolution algorithm is not straightforward, in particular because a gradient for the cost function is not defined. Nevertheless, two general optimization algorithms have been applied to finding the degree distributions of LDPC codes, *iterative linear programming* and the confusingly named (for us) *differential evolution*.

3.4 Bibliographic notes

The type of analysis of message-passing decoding which we now call density evolution first appeared for regular codes in Gallager's work [1]. For irregular codes density evolution was first proposed in [15] when considering the binary erasure channel, applied to hard decision message-passing decoding in [4] and generalized to sum-product decoding on memoryless channels in [14, 16]. Online implementations of density evolution can be found at [17] and [18],

Topic 4: LDPC Code Properties

In this topic low-density parity-check (LDPC) codes are discussed in detail with a focus on code design, encoding, and performance. Firstly the properties of LDPC codes are considered and the difference between classical decoders and message-passing decoders is made apparent when considering how and why message-passing decoding algorithms fail.

Secondly, LDPC codes with linear-time encoders, in particular quasi-cyclic codes and repeat-accumulate codes, are considered. Finally, a number of specific methods for code design are presented. Pseudo-random constructions are considered with many of the more recent strategies for cycle and pseudo-codeword removal presented and structured LDPC codes are considered with a focus on codes from combinatorial designs.

4.1 LDPC properties

While there is no one recipe for a "good" LDPC code, there are a number of principles which inform the code designer.

Firstly, a good LDPC code is also a good classical block code. The power of sum-product decoding is that it can decode very long codes, but it is nevertheless a sub-optimal decoding algorithm which can only do as well as the optimal decoder (were it possible to implement the optimal decoder). If the LDPC code has a poor minimum distance the sum-product decoder will produce an error floor in exactly the same manner as the ML or MAP decoder. That LDPC codes often do not show an error floor is because, for very long and very sparse codes, it is relatively easy to pseudo-randomly construct a code with a good minimum distance.

A good classical code is however not necessarily a good LDPC code. Most critically, the sparsity of the parity-check matrix, H, is essential to keep the decoding complexity low. A sparse H also guarantees the linear growth in minimum distance with length proven for LDPC ensembles. A good Tanner graph also has a large girth and good expansion. This increases the number of correlation-free iterations and improves the convergence of the decoder.

Other desirable properties of LDPC codes depend on how they are to be applied. For a capacity-approaching performance on very low noise channels, long code lengths and random or pseudo-randomly constructed irregular parity-check matrices produce the performance closest to capacity. However, capacity-approaching performance (in the bit error rate) equate to poor word error rates and low error floors, making capacity-approaching codes completely unsuitable for some applications.

For long codes a randomly chosen parity-check matrix is almost always good and structured matrices are often much worse. However, for short and

medium-length LDPC codes irregular constructions are generally not better than regular ones and graph-based or algebraic constructions can outperform random ones. In addition, using structured parity-check matrices can lead to much simpler implementations, particularly for encoding, and can guarantee girth and minimum distance properties difficult to achieve randomly for shorter codes. In particular, for very low error floors a reasonably short algebraic construction with large column weight will produce the required performances, with the trade off of a larger gap to capacity.

4.1.1 Girth and expansion

Cycles in the Tanner graph lead to correlations in the marginal probabilities passed by the sum-product decoder; the smaller the cycles the fewer the number of iterations that are correlation free. Thus cycles in the Tanner graph affect decoding convergence, and the smaller the code girth, the larger the effect. Definite performance improvements can be obtained by avoiding 4-cycles[1] and 6-cycles from LDPC Tanner graphs but the returns tend to diminish as the girth is increased further.

It is important to keep in mind that when considering the properties of an LDPC code we are often actually considering the properties of a particular choice of parity-check matrix for that code, and that a different choice of parity-check matrix for the same code may behave differently.

Example 4.1.

The parity-check matrices

$$H_1 = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

and

$$H_2 = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

both describe the same code, but have Tanner graphs with different girth. Figure 4.1 shows the performance of sum-product decoding using each of these parity-check matrices on the same channel. Removing a single 4-cycle from the parity-check matrix has noticeably improved the performance of the sum-product decoder even though exactly the same code is being decoded. In Topic 5 we will outline some of the methods used to construct Tanner graphs without short cycles.

A related concept to the graph girth is the graph *expansion*. In a good expander every subset of vertices has a large number of neighbors that are not

[1] A small subset of LDPC codes which include 4-cycles have been shown to perform well with sum-product decoding, however this effect is due to the large number of extra linearly dependent rows in these parity-check matrices which helps to overcome the negative impact of the cycles.

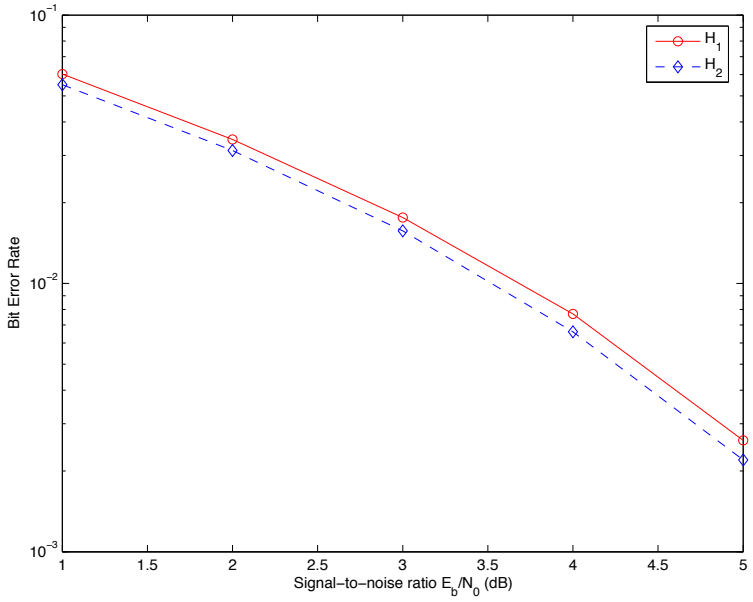

Figure 4.1: The bit error rate performance of sum-product decoding on an AWGN channel using the parity-check matrices from Example 4.1.

in the subset. More precisely, any subset S of bit vertices of size m or less is connected to at least $\epsilon|S|$ constraint vertices, for some defined m and ϵ.

If a Tanner graph is a good expander then the bit nodes of a small set of erroneous codeword bits will be connected to a large number of check nodes, all of which will be receiving correct information from an even larger number of the correct codeword bits. Sipser and Spielman [19] showed that the expansion of the graph is a significant factor in the application of iterative decoding. Using only a simple hard decision decoding algorithm they proved that a fixed fraction of errors can be corrected in linear time provided that the Tanner graph is a good enough expander.

4.1.2 Stopping sets and pseudo-codewords

As we saw in Topic 1, the message-passing decoding of LDPC codes on erasure channels is particularly straightforward since a transmitted bit is either received correctly or completely erased. Decoding is a process of finding parity-check equations which check on only one erased bit. In a decode iteration all such parity-check equations are found and the erased bits corrected. After these bits have been corrected any new parity-check equations checking on only one erased bit are then corrected in the subsequent iteration. The process is repeated until all the erasures are corrected or all the remaining uncorrected parity-check equations check on two or more erased bits. The question for coding theorists is when will this occur and why.

For the binary erasure channel at least, the answer is known. The message-passing decoder will fail to converge if the erased bits include a set of code bits, $\mathcal{S}$, which are a *stopping set*. A stopping set, $\mathcal{S}$, is a set of code bits with the property that every parity-check equation which checks on a bit in $\mathcal{S}$ checks on

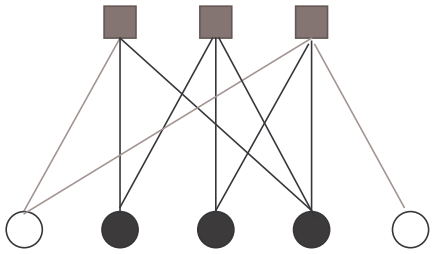

Figure 4.2: The Tanner graph of a length 5 code with a stopping set of size three shown in bold.

at least two bits in $\mathcal{S}$. The size of a stopping set is the number of bits it includes, and the minimum stopping set size of a parity-check matrix is denoted $S_{\min}$.

Example 4.2.

Fig. 4.2 shows the Tanner graph of a length 5 parity-check matrix with 3 parity-check equations and filled bit nodes representing a stopping set of size three.

The message-passing decoder cannot correct a set of erased bits $\mathcal{S}$ which are a stopping set. Since every parity-check node connected to $\mathcal{S}$ includes at least two erased bits there will never be a parity-check equation available to correct a bit in $\mathcal{S}$, regardless of the number of iterations employed. In a sense we can say that the decoder has converged to the stopping set. The stopping set distribution of an LDPC parity-check matrix determines the erasure patterns for which the message-passing decoding algorithm will fail in the same way that the codeword distribution of a code determines the error patterns for which the ML decoder will fail. The minimum stopping set size determines the minimum number of erased bits which can cause a decoding failure.

Example 4.3.

The same LDPC code used in Example 1.12:

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

is used to encode the codeword

$$\mathbf{c} = [0\ 0\ 1\ 0\ 1\ 1].$$

$\mathbf{c}$ is sent though an erasure channel but this time the 3-rd, 5-th and 6-th bits are erased and so the vector

$$\mathbf{y} = [0\ 0\ x\ 0\ x\ x]$$

is received. Message-passing decoding is used to recover the erased bits.

Initialization is $M_i = y_i$ which gives

$$M = [0\ 0\ x\ 0\ x\ x].$$

For Step 1 the check node messages are calculated. The 1-st check node is joined to the 1-st, 2-nd and 4-th bit nodes, and so has no incoming 'x' messages. The 2-nd check includes the 2-nd, 3-rd and 5-th bits, and so receives two 'x' messages, (M_3 and M_5) and thus cannot be used to correct any codeword bits. The 3-rd check includes the 1-st, 5-th and 6-th bits and so also receives two x messages, (M_5 and M_6) and thus cannot be used to correct any codeword bits. Finally, the 4-th check includes the 3-rd, 4-th and 6-th bits and so receives two 'x' messages, (M_3 and M_6) and thus also cannot used to correct any codeword bits.

In Step 2 there are no new messages coming into any of the erased bits and no corrections can be made. Regardless of how many iterations are run there will never be a corrected erased bit. By examination of H, it is clear why this is the case: the set of erased codeword bits is a stopping set in H.

Knowing exactly the places where the decoding algorithm will fail allows us to predict its performance. Indeed, if the stopping set distribution of an LDPC parity-check matrix were known, the performance of the message-passing decoder on the BEC could be determined exactly by counting the stopping sets. The probability of bit erasure for a given parity-check matrix, H, of length n on a binary erasure channel with erasure probability ϵ is

$$P(H,\epsilon) = \sum_{v=0}^{n} \binom{n}{v} \epsilon^v (1-\epsilon)^{n-v} \left(\frac{N(v)}{T(v)} \right), \tag{4.1}$$

where $T(v)$ is the total of number bit sets of size v and $N(v)$ is the number of those bit sets which are stopping sets.

Finding the stopping set distribution of an individual parity-check matrix is as prohibitively complex as finding its codeword distance distribution, however, the *average* stopping set distribution of a regular LDPC ensemble can be found combinatorially. This technique, called *finite-length analysis*, gives the exact average bit and block error probabilities for any regular ensemble of LDPC codes over the binary erasure channel (BEC) when decoded iteratively. From the ensemble perspective $T(v)$ is the total of number of ways a bit set of size v can be constructed over all possible codes in the ensemble and $N(v)$ is the number of those ways which result in the v points being a stopping set. Thus $N(v)/T(v)$ can be considered the probability that a given set of v points is a stopping set.

Every codeword is a stopping set. This is easy to see since any check on a non-zero bit in the codeword must include an even number of non-zero codeword bits to ensure even parity. Thus the set of stopping sets includes the set of codewords. However, not all of the stopping sets correspond to codewords. The stopping set in Fig. 4.2 for example is not a codeword.

Not all parity-check matrices for the same code will have the same stopping set distribution or same minimum stopping set size. Fig. 4.3 for example shows the Tanner graphs of two parity-check matrices for the same code, one with three size-3 stopping sets and one with two size-3 stopping sets. The two common stopping sets include bits $\{1,2,4\}$ and $\{1,3,5\}$. Every possible

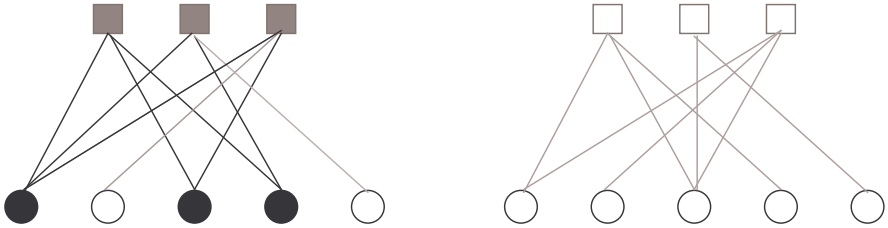

Figure 4.3: The Tanner graph of two different parity-check matrices for the same code.

parity-check matrix for this code will also contain stopping sets in these bit locations.

The role of stopping sets in predicting the performance of message-passing decoding on the BEC tells us that for message-passing decoding, unlike for ML decoding, properties other than the codeword set influence the decoder performance. The same is true of message-passing decoding on more general channels, however defining the configurations which lead to decoder failure in the general case, called *pseudo-codewords* is less straightforward.

Stepping back briefly we note that the low complexity of iterative message-passing decoding is because the algorithm operates locally on the Tanner graph representing the matrix H (i.e. each node in the decoding algorithm acts only on the messages it can see, not on the other messages in the graph). This same operation also leads to a fundamental weakness of the algorithm: because it acts locally, each node does not know the whole structure of the Tanner graph.

An alternative graph which would produce, locally, the same LLRs as the actual Tanner graph is called a finite lift or *cover* of the Tanner graph. Pseudo-codewords can be thought of as the set of valid codewords in all finite lifts of the Tanner graph. Since the message-passing decoder cannot locally distinguish between the actual Tanner graph and the lifted graphs, any codeword in any of the lifted graphs is as equally valid as a codeword in the Tanner graph. Thus when we say that the decoder has failed to converge it has actually converged to one of these pseudo-codewords.

4.1.3 Threshold vs. erasure floor using stopping sets

Unfortunately, the LDPC codes which are capacity approaching have poor error floor performances, while the codes with extremely low error floors have thresholds far from capacity. The large number of weight-2 nodes returned by optimizing the degree distribution result in a reduced minimum distance for the ensemble. This tradeoff is made more precise on the BEC using stopping sets.

On the binary erasure channel the message-passing decoder will fail if the erased bits contain a stopping set and so, for a given erasure rate, the expected number of stopping sets will determine the expected error rate of the decoder. To determine the impact of the weight two nodes only the stopping sets in the subgraph, $\mathcal{T}_2$, which includes the degree-2 bit nodes, all of the edges connected to degree-2 bit nodes and the check nodes connected to the degree-2 bit nodes by an edge, are considered.

Since the degree of all the bit nodes in $\mathcal{T}_2$ is 2, a stopping set of size k in $\mathcal{T}_2$ is also a cycle of size $2k$. The expected number of cycles of size k in a length n ensemble is:

$$\mathbb{E}_{k-\text{cycles}}(C_{\lambda(x),\rho(x)}, n) = \frac{(\lambda_2\rho'(1))^k}{2k} + O(n^{-1/3}), \tag{4.2}$$

and thus the average probability that a randomly chosen size-v subset of the $\psi_2 n$ degree-2 bit nodes in the ensemble $C_{\lambda(x),\rho(x)}$ is a stopping set is:

$$P_{\text{SS}}(C_{\lambda(x),\rho(x)}, v) = \frac{(\lambda_2\rho'(1))^v/2v + O(n^{-1/3})}{\binom{\psi_2 n}{v}}.$$

The word error rate on the BEC with erasure probability ϵ is lower bounded by summing over the contribution of stopping sets of size $s = 2, \ldots, \psi_2 n$ in the $\psi_2 n$ degree-2 bit nodes:

$$\begin{aligned}\mathbb{E}_{\text{WER}}(C_{\lambda(x),\rho(x)}, n, \epsilon) &\geq \sum_{s=2}^{\psi_2 n} \binom{\psi_2 n}{s} \epsilon^s P_{\text{SS}}(C_{\lambda(x),\rho(x)}, s) \\ &= \sum_{s=2}^{\psi_2 n} \epsilon^s \left(\frac{(\lambda_2\rho'(1))^s}{2s} + O(n^{-1/3}) \right), \end{aligned} \tag{4.3}$$

where ϵ^s is the probability of an erasure of size at least s occurring. For asymptotically long codes,

$$\begin{aligned}\lim_{n\to\infty} \mathbb{E}_{\text{WER}}(C_{\lambda(x),\rho(x)}, n, \epsilon) &\geq \lim_{\psi_2 n\to\infty} \sum_{s=2}^{\psi_2 n} \frac{(\lambda_2\rho'(1)\epsilon)^s}{2s} \\ &= \ln\left(\frac{1}{\sqrt{1+\lambda_2\rho'(1)\epsilon}} \right) - \frac{(\lambda_2\rho'(1)\epsilon)}{2}. \end{aligned} \tag{4.4}$$

For randomly constructed LDPC codes from the ensemble with degree distribution pair ($\lambda(x)$,$\rho(x)$) and girth g, the expected error floor performance of the ensemble will be dominated by stopping sets of size $g/2$ and the word error rate is approximated by

$$W_g \triangleq \frac{(\lambda_2\rho'(1)\epsilon)^{\frac{g}{2}}}{g}. \tag{4.5}$$

To bound the ensemble degree distribution to obtain a word error rate below W_g the degree distribution is constrained to satisfy:

$$\lambda_2 \leq \frac{E}{\rho'(1)\epsilon^*}, \qquad \text{where} \quad E \triangleq (gW_g)^{2/g}, \tag{4.6}$$

and ϵ^* is the threshold value returned by density evolution. Thus $\epsilon < \epsilon^*$ corresponds to the error floor region of the WER curve, making ϵ^* an ideal erasure probability at which to evaluate (4.5). Note that setting $E = 1$ returns the stability constraint for the BEC and the traditional optimized degree distribution is returned.

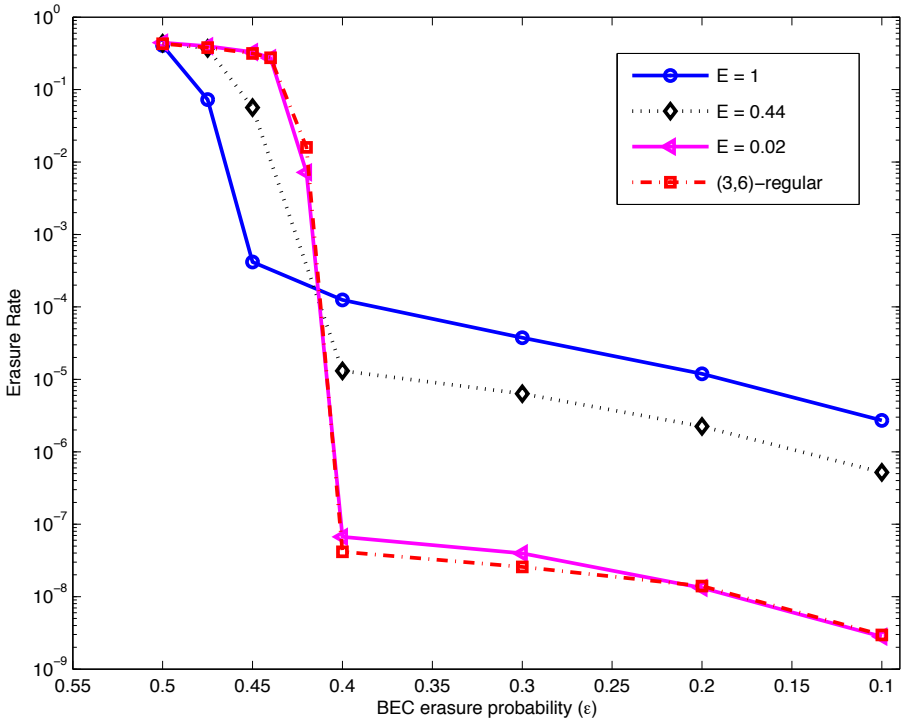

Figure 4.4: Average decoding performances of LDPC ensembles with constrained degree distributions from Example 4.4.

Example 4.4.

Fig. 4.4 shows the simulated ensemble average performance, using message-passing decoding, of codes with varying degree distributions. The irregular ensemble gives the best threshold performance, but a very poor error floor, while the regular LDPC ensemble has a better error floor and worse threshold. The constrained degree distributions allow a tradeoff between the threshold and error floor performance to be made.

Similar arguments apply to constraining the degree distributions of LDPC codes designed for more general memoryless channels. For these channels, the number of low weight codewords within $\mathcal{T}_2$ are controlled by constraining λ_2.

4.2 Easily encodable LDPC codes

Rather than trying to convert a parity-check matrix into an encodable form after it has been produced, encodability can be instead be incorporated into the design of the parity-check matrix. For classical codes this has been successfully achieved using cyclic and quasi-cyclic codes and the same ideas can be applied to LDPC codes.

4.2.1 Quasi-cyclic codes

A code is *quasi-cyclic* if for any cyclic shift of a codeword by c places the resulting word is also a codeword, and so a cyclic code is a quasi-cyclic code with $c = 1$. The simplest quasi-cyclic codes are row circulant codes which are described by a parity-check matrix

$$H = [A_1, A_2, \ldots, A_l], \tag{4.7}$$

where $A_1, \ldots, A_l$ are binary $v \times v$ circulant matrices.

Provided that one of the circulant matrices is invertible (say A_l) the generator matrix for the code can be constructed in systematic form

$$G = \begin{bmatrix} & & (A_l^{-1}A_1)^T \\ & I_{v(l-1)} & (A_l^{-1}A_2)^T \\ & & \vdots \\ & & (A_l^{-1}A_{l-1})^T \end{bmatrix}, \tag{4.8}$$

resulting in a quasi-cyclic code of length vl and dimension $v(l-1)$. As one of the circulant matrices is invertible, the construction of the generator matrix in this way necessarily leads to a full rank H.

The algebra of $(v \times v)$ binary circulant matrices is isomorphic to the algebra of polynomials modulo $x^v - 1$ over GF(2) [20]. A circulant matrix A is completely characterized by the polynomial $a(x) = a_0 + a_1 x + \cdots + a_{v-1}x^{v-1}$ with coefficients from its first row, and a code C of the form (4.7) is completely characterized by the polynomials $a_1(x), \ldots, a_l(x)$. Polynomial transpose is defined as

$$a(x)^T = \sum_{i=0}^{n-1} a_i x^{n-i} \qquad (x^n = 1).$$

For a binary code, length $n = vl$ and dimension $k = v(l-1)$, the k bit message $[i_0, i_1, \ldots, i_{k-1}]$ is described by the polynomial $i(x) = i_0 + i_1 x + \cdots + i_{k-1}x^{k-1}$ and the codeword for this message is $c(x) = [i(x), p(x)]$, where $p(x)$ is given by

$$p(x) = \sum_{j=1}^{l-1} i_j(x) * (a_l^{-1}(x) * a_j(x))^T, \tag{4.9}$$

$i_j(x)$ is the polynomial representation of the information bits $i_{v(j-1)}$ to i_{vj-1},

$$i_j(x) = i_{v(j-1)} + i_{v(j-1)+1}x + \cdots + i_{vj-1}x^{v-1}$$

and polynomial multiplication $(*)$ is modulo $x^v - 1$.

Example 4.5.

A rate-$\frac{1}{2}$ quasi-cyclic code with $v = 5$, is made up of a first circulant described by $a_1(x) = 1 + x$, and a second circulant described by $a_2(x) = 1 + x^2 + x^4$.

$$H = \left[\begin{array}{ccccc|ccccc} 1 & 1 & & & & 1 & & 1 & & 1 \\ & 1 & 1 & & & 1 & 1 & & 1 & \\ & & 1 & 1 & & & 1 & 1 & & 1 \\ & & & 1 & 1 & 1 & & 1 & 1 & \\ 1 & & & & 1 & & 1 & & 1 & 1 \end{array}\right]$$

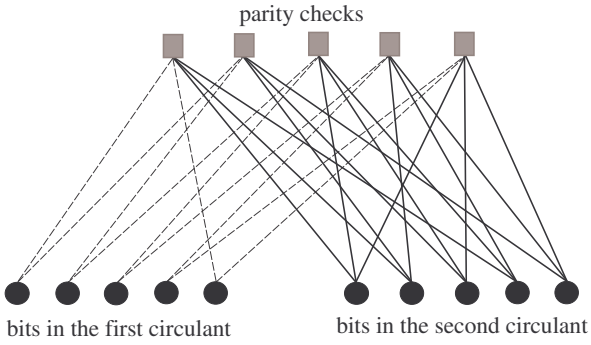

Figure 4.5: A Tanner graph for the quasi-cyclic LDPC code in Example 4.5

The second circulant is invertible

$$a_2^{-1}(x) = x^2 + x^3 + x^4,$$

and so the generator matrix contains a 5×5 identity matrix and the 5×5 matrix described by the polynomial

$$(a_2^{-1}(x) * a_1(x))^T = (1 + x^2)^T = 1 + x^3.$$

$$G = \left[\begin{array}{ccccc|ccccc} 1 & & & & & 1 & & & 1 & \\ & 1 & & & & & 1 & & & 1 \\ & & 1 & & & 1 & & 1 & & \\ & & & 1 & & & 1 & & 1 & \\ & & & & 1 & & & 1 & & 1 \end{array}\right]$$

Fig. 4.5 shows the Tanner graph for H.

Linear-time encoding can be achieved using $(l-1)$ v-stage shift registers with separate length v shift registers for each circulant in G.

Example 4.6.

A quasi-cyclic, length-108, rate-3/4 LDPC code has the parity-check matrix:

$$H = [A_1, A_2, A_3, A_4].$$

H consists of four circulants defined by

$$a_1(x) = 1 + x^3 + x^{16},$$

$$a_2(x) = x^2 + x^6 + x^8,$$

$$a_3(x) = x + x^8 + x^9,$$

$$a_4(x) = x + x^{13} + x^{23}.$$

The polynomial $a_4(x)$ is invertible with inverse given by

$$a_4^{-1}(x) = x+x^4+x^5+x^6+x^7+x^9+x^{12}+x^{13}+x^{15}+x^{17}+x^{20}+x^{21}+x^{23}+x^{24}+x^{25},$$

and so the parity-check matrix can be put into systematic form

$$H_{\rm S} = [A_4^{-1}A_1, A_4^{-1}A_2, A_4^{-1}A_3, I_{27}].$$

We thus have,

$$\begin{aligned}
a_4^{-1}(x)a_1(x) = \;& 1 + x + x^4 + x^5 + x^7 + x^9 + x^{11} + x^{13} + x^{15} + x^{18} + x^{19} \\
& + x^{22} + x^{23} + x^{25} + x^{26}, \\
a_4^{-1}(x)a_2(x) = \;& x^4 + x^5 + x^7 + x^9 + x^{12} + x^{17} + x^{19} + x^{22} + x^{26}, \\
a_4^{-1}(x)a_3(x) = \;& x^4 + x^6 + x^7 + x^{10} + x^{13} + x^{14} + x^{15} + x^{18} + x^{19} + x^{21} \\
& + x^{22} + x^{23} + x^{24} + x^{25} + x^{26},
\end{aligned}$$

and the generator matrix for this code is:

$$G = \begin{bmatrix} & & (A_4^{-1}A_1)^T \\ & I_{81} & (A_4^{-1}A_2)^T \\ & & (A_4^{-1}A_3)^T \end{bmatrix}.$$

Using G in this form the code can be encoded using shift registers. Figure 4.6 shows an encoding circuit for this code.

Note that although we use $H_{\rm S}$ to construct G we will use the original matrix, H to do our decoding. Both H and $H_{\rm S}$ are valid parity-check matrices for the code, however H has the properties required for sum-product decoding.

Block circulant quasi-cyclic codes

More general quasi-cyclic codes are the *block circulant* codes. The parity-check matrix of a block circulant quasi-cyclic LDPC code is:

$$\begin{bmatrix} I_p & I_p & I_p & \dots & I_p \\ I_p & I_p(p_{1,1}) & I_p(p_{1,2}) & \dots & I_p(p_{1,w_r}) \\ \vdots & & & \ddots & \vdots \\ I_p & I_p(p_{w_c-1,1}) & I_p(p_{w_c-1,2}) & \dots & I_p(p_{w_c-1,w_r-1}) \end{bmatrix},$$

where I_p represents the $p \times p$ identity matrix and $I_p(p_{i,j})$ represents the circulant shift of the identity matrix by $r + p_{i,j} (\mod p)$ columns to the right which gives the matrix with the r-th row having a one in the $(r + p_{i,j} \mod p)$-th column. Block circulant LDPC codes can have better minimum distances and girths than row-circulant codes.

4.2.2 Repeat-accumulate codes

Earlier we saw that an LDPC code can be put into an approximate upper triangular form so as to facilitate almost linear-time encoding. A repeat-accumulate (RA) code is an LDPC code with an upper triangular form already built into the parity-check matrix during the code design.

An $m \times n$ RA code parity-check matrix H has two parts:

$$H = [H_1, H_2], \tag{4.10}$$

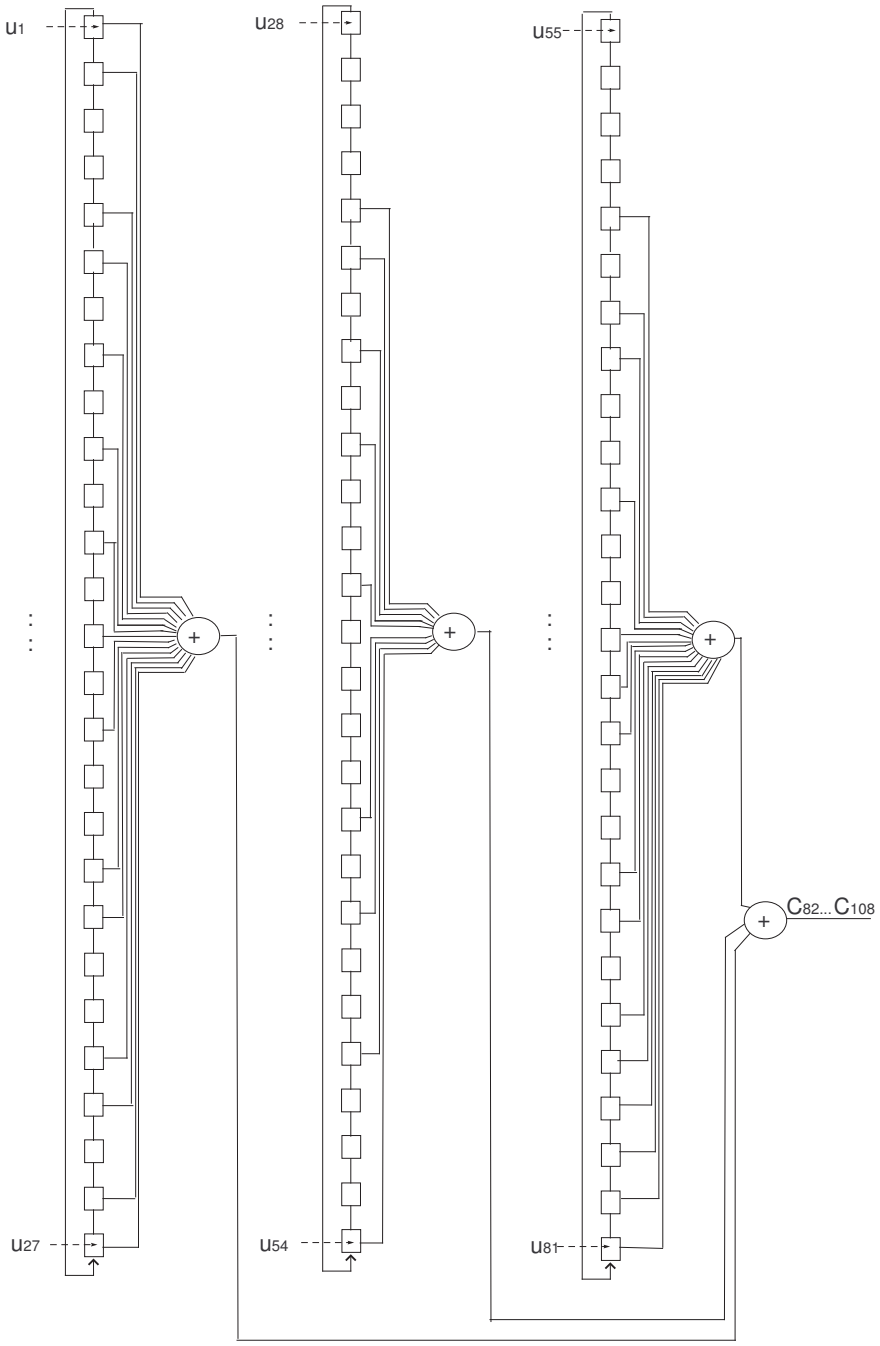

Figure 4.6: Encoding circuit for the $n = 108, k = 81$ LDPC code from Example 4.6.

where H_2 is an $m \times m$ matrix with the form:

$$H_2 = \begin{bmatrix} 1 & 0 & 0 & & 0 & 0 & 0 \\ 1 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & 1 & & 0 & 0 & 0 \\ & \vdots & & \ddots & & \vdots & \\ 0 & 0 & 0 & & 1 & 0 & 0 \\ 0 & 0 & 0 & \cdots & 1 & 1 & 0 \\ 0 & 0 & 0 & & 0 & 1 & 1 \end{bmatrix}. \tag{4.11}$$

The parity-check matrix of an RA code is called (q, a)-*regular* if the weight of all the rows of H_1 are the same, a, and the weight of all the columns of H_1 are the same, q. Note that a regular RA parity-check matrix has columns of weight 2, and one column of weight 1, in H_2 and so is not regular in the sense of (j, r)-regular LDPC codes. An irregular RA code will have an irregular column weight distribution in H_1, with H_2 the same as for a regular code.

Example 4.7.

A (3,2)-regular RA parity-check matrix for a length-10 rate-2/5 code is:

$$H = \begin{bmatrix} 1 & . & 1 & . & 1 & . & . & . & . & . \\ . & 1 & . & 1 & 1 & 1 & . & . & . & . \\ 1 & 1 & . & . & . & 1 & 1 & . & . & . \\ . & . & 1 & 1 & . & . & 1 & 1 & . & . \\ 1 & . & 1 & . & . & . & . & 1 & 1 & . \\ . & 1 & . & 1 & . & . & . & . & 1 & 1 \end{bmatrix} \tag{4.12}$$

As for LDPC codes, the Tanner graph of an RA code is defined by H, where there is a parity-check equation vertex for every parity-check equation in H and a bit vertex for every codeword bit. The Tanner graph of an RA code consists of $m = kq/a$ check vertices and $k + m = k(q + a)/a$ bit vertices.

Unlike for a general LDPC code, the message bits in the codeword of an RA code are easily distinguished from the parity bits. Fig. 4.7 shows the Tanner graph for the RA code from Example 4.7. We distinguish between *message bit vertices* corresponding to the K message bits in the codeword, shown at the top of the graph, and *parity bit vertices* corresponding to the M parity-check bits in the codeword, which are shown at the bottom of the graph.

Encoding RA codes

The encoding of an RA code is split into four main operations. Firstly, since H is systematic we know that the columns of H_1 correspond to the message bits. From the column regularity of H_1 we see that each message bit is repeated q times, which can be implemented with a rate $1/q$ *repetition code*. Next, the row regularity of H_1 shows that for every parity-bit a of these repeated message bits are summed modulo 2, a process that we implement using a *combiner*. The final step is to define the mapping of the repeated message bits to the combiner inputs. This is done by defining a permutation pattern Π called an *interleaver*.

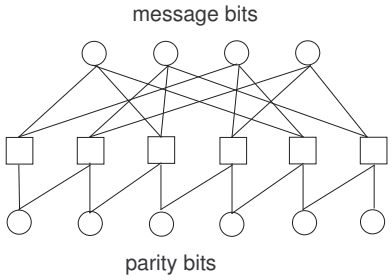

Figure 4.7: The Tanner graph for the RA parity-check matrix in Example (4.7).

Example 4.8.

The code in Example 4.7 is a length-10 RA code with $q = 3$ repetition code, $a = 2$ combiner and the interleaver $\Pi = [1, 7, 4, 10, 2, 5, 8, 11, 3, 9, 6, 12]$.

The encoding process is as follows: The qK bits at the output of the repetition code are q copies of the K message bits $\mathbf{m} = [m_1, \ldots, m_K]$, in the form

$$\begin{aligned} \mathbf{b} &= [b_1, b_2, \ldots, b_{qK}] \\ &= [\underbrace{m_1, m_1, \ldots, m_1}_{q}, \underbrace{m_2, m_2, \ldots, m_2}_{q}, \ \ldots, \ \underbrace{m_K, m_K, \ldots, m_K}_{q}], \end{aligned}$$

and so we have

$$b_i = m_{f(i)}, \qquad f(i) = \lceil i/q \rceil, \tag{4.13}$$

where $\lceil x \rceil$ denotes the smallest integer greater than or equal to x.

The interleaver pattern, $\Pi = [\pi_1, \pi_1, \ldots, \pi_n]$, defines a permutation of the input bits, $\mathbf{b} = [b_1, b_2, \ldots, b_n]$, to the output bits

$$\mathbf{d} = [d_1, d_2, \ldots, d_n] = [b_{\pi_1}, b_{\pi_2}, \ldots, b_{\pi_n}]. \tag{4.14}$$

Thus two different interleaver patterns will describe the same RA code if the difference in the permutation pattern results in a difference in which copy of the same message bit is used.

The bits at the output of the interleaver are combined, modulo-2, in sets of a bits, before being passed to the accumulator. Thus the $M = Kq/a$ bits, $\mathbf{r} = [r_1, r_2, \ldots, r_M]$, at the output of the combiner, are given by

$$r_i = d_{(i-1)a+1} \oplus d_{(i-1)a+2} \oplus \cdots \oplus d_{ia}, \qquad i = 1, 2, \ldots, M, \tag{4.15}$$

where $\oplus$ denotes modulo-2 addition.

Finally, the M parity bits, $\mathbf{p} = [p_1, p_2, \ldots, p_M]$, at the output of the accumulator are defined by

$$p_i = p_{i-1} \oplus r_i, \qquad i = 1, 2, \ldots, M. \tag{4.16}$$

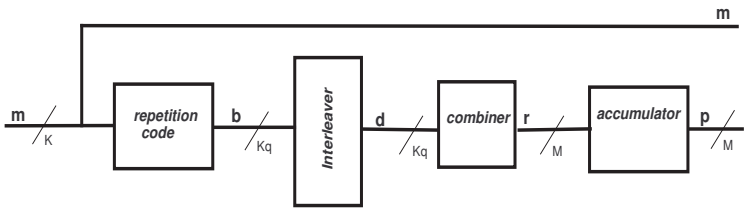

Figure 4.8: The encoding circuit for RA codes.

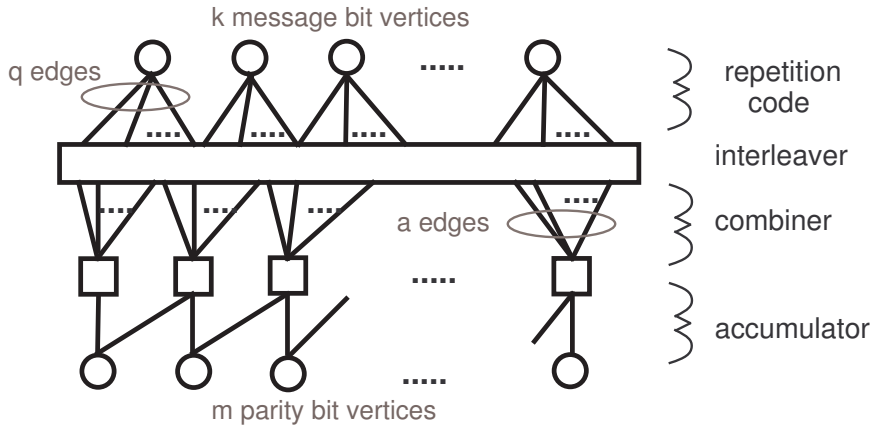

Figure 4.9: An RA code Tanner graph.

We consider *systematic* RA codes, that is codes for which both the original message bits and the parity bits are explicitly sent to the receiver, and so the final codeword is $\mathbf{c} = [m_1, m_2, \ldots, m_K, p_1, p_2, \ldots, p_M]$, and thus we have a code with length, $N = K(1 + q/a)$, and rate $R = \frac{a}{a+q}$.

The rows of H_1 describe the equations in (4.13)-(4.15), e.g. if we have $r_i = m_{c_1} + m_{c_2}$ then the i-th row of H_1 is '1' in the columns c_1 and c_2 and '0' elsewhere. In equation (4.10), H_2 is an $M \times M$ matrix which describes (4.16):

Fig. 4.8 shows the encoding circuit for RA codes.

Those familiar with turbo codes will note that the encoder of an RA code looks a lot like that of a serially concatenated turbo code. Indeed RA codes were first presented as a simple class of turbo codes for which coding theories could be developed. The two component codes are the repetition code and

$$\frac{1}{1+D}$$

convolutional code which is the accumulator. Fig. 4.9 shows the relationship between the LDPC and turbo code representations of an RA code.

4.3 Bibliographic notes

The effect of cycles on the practical performance of LDPC codes was demonstrated by simulation experiments when LDPC codes were rediscovered by MacKay and Neal [21] in the mid-1990s, and the beneficial effects of using graphs free of short cycles were shown [7]. By proving the convergence of the sum-product algorithm for codes whose graphs are free of cycles, Tanner was the first to formally recognize the importance of cycle-free graphs in the context of iterative decoding [2].

Stopping sets were introduced in [22] and used to develop analysis tools for finite length LDPC ensembles. For more on stopping sets and finite-length analysis see [23–25] while a good source for more information on pseudo-codewords is [26].

Quasi-cyclic codes, were first presented in [27] and [20], for a good introduction to quasi-cyclic codes see [28] or [8]. Block circulant quasi-cyclic LDPC codes are well presented in [29].

Topic 5: LDPC Code Construction

In the previous topic a number of the properties that make a good LDPC code have been discussed. In this section some of the methods used to construct LDPC codes which achieve these properties are outlined.

5.1 Graph based constructions

For long codes, randomly choosing a parity-check matrix almost always produces a good code. In fact for very long codes this is guaranteed by the concentration theorem which says that behavior of randomly chosen codes from an ensemble concentrates around the ensemble average. Nevertheless, for practical applications the codes may not be long enough and a user is not going to accept a code that "will probably" work. Most codes are constructed at least pseudo-randomly, where the construction is random but certain bad configurations such as 4-cycles, are either avoided during construction or removed afterwards. Some of these techniques are considered in the following:

Column or row splitting

In this technique cycles, or indeed any unwanted configurations, in the parity-check matrix are removed by splitting a column or row in half. In column splitting a single column in H is replaced by two columns which share the entries of the original column between them. Since an extra column has been added, a new code is produced with length one greater than the previous code, and with a parity-check matrix made slightly more sparse.

Example 5.1.
Figure 5.1 shows column splitting applied to remove a 4-cycle.

Figure 5.1: Column splitting to remove a 4-cycle.

Figure 5.2: Row splitting to remove a 4-cycle.

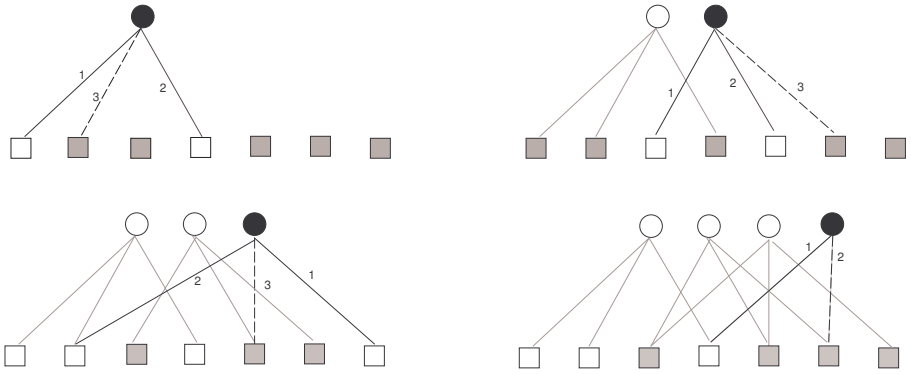

Figure 5.3: Bit filling to avoid cycles of size 4.

Alternatively, a single row in H can be replaced by two rows which share the entries of the original row between them. Since an extra row has been added, a new code is produced with 1 more parity-check equation in H than the previous H and with a parity-check matrix which is slightly more sparse.

Example 5.2.

Figure 5.2 shows row splitting applied to remove a 4-cycle.

Bit filling or progressive edge growth (PEG) Tanner graphs

In bit filling bit nodes are added to the Tanner graph one at a time and edges connecting the new bit nodes to the graph are chosen to avoid cycles of size g. For each new bit node b_i, w_c check nodes are selected to join by an edge to b_i. The set of feasible check nodes are the nodes that are distance $g/2$ or more edges away from all of the check nodes already connected to $b + i$. From each set of feasible check nodes the check node chosen is the one which least used so far (i.e. with the lowest degree). Fig 5.3 shows this process.

In progressive edge growth Tanner graphs edges are similarly added to the graph one at a time but instead of meeting some fixed girth requirement g the edge is added so as to maximize the local girth at the current bit node.

These techniques can also be applied to semi-structured codes, i.e. an RA code can be constructed by fixing the accumulator portion of the Tanner graph and applying bit filling to the remaining bit nodes.

5.2 Codes from designs

A combinatorial design is an assignment of a set of objects into subsets subject to some defined condition on the size, structure or incidence of the subsets.

Example 5.3.

A simple combinatorial problem is to arrange a set of seven academics into seven committees with three academics in each committee, every academic serving on the same number of committees, and each pair of academics serving together in exactly one committee. The set of academics (points)

$$\mathcal{P} = \{1, 2, 3, 4, 5, 6, 7\}$$

can be formed into a design with committees (blocks),

$$\mathcal{B} = \{[1, 3, 5], [2, 3, 7], [4, 5, 7], [1, 6, 7], [1, 2, 4], [3, 4, 6], [2, 5, 6]\}. \quad (5.1)$$

Formally, an *incidence structure* $(\mathcal{P}, \mathcal{B}, \mathcal{I})$ consists of a finite non-empty set $\mathcal{P}$ of points (academics) and a finite non-empty set $\mathcal{B}$ of subsets of those points called blocks (committees), together with an incidence relation $\mathcal{I} \subseteq \mathcal{P} \times \mathcal{B}$. A point P and block B are *incident*, denoted $P \in B$, if and only if $(P, B) \in \mathcal{I}$. A *design* $\mathcal{D}$ is an incidence structure with a constant number of points per block and no repeated blocks. A design is *regular* if the number of points in each block, and the number of blocks which contain each point, designated γ and r respectively, are the same for every point and block in the design. In the field of combinatorial designs the block size is usually denoted by the symbol k, however we use γ in this thesis to avoid confusion with the use of k for the number of message symbols in the code.

Every design can be represented by a $v \times b$ binary matrix N, $v = |\mathcal{P}|$, $b = |\mathcal{B}|$ called an *incidence matrix*, where each column in N represents a block B_j of the design and each row a point P_i. The (i, j)th entry of N is a one if the i-th point is contained in the j-th block, otherwise it is 0:

$$N_{i,j} = \begin{cases} 1 & \text{if } P_i \in B_j, \\ 0 & \text{otherwise.} \end{cases} \quad (5.2)$$

The *incidence graph* of $\mathcal{D}$ has vertex set $\mathcal{P} \bigcup \mathcal{B}$, with two vertices x and y connected if and only if $x \in \mathcal{P}$, $y \in \mathcal{B}$ and $P_x \in B_y$, or $x \in \mathcal{B}$, $y \in \mathcal{P}$ and $P_y \in B_x$, and is thus a bipartite graph.

Example 5.4.

The design in Example 5.3 can easily be seen to satisfy the regularity constraint with $\gamma = 3$ points in every block (3 academics in every committee) and each point in exactly $r = 3$ blocks (each academic on exactly three committees). An incidence matrix and incidence graph for this design are shown in Fig 5.4 using

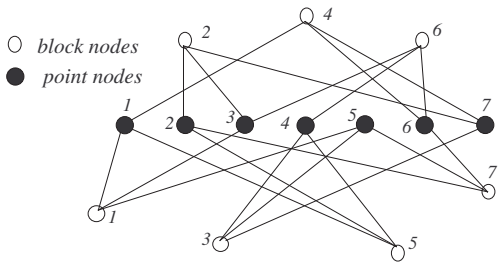

Figure 5.4: An incidence graph of the 2-$(7, 3, 1)$ design in Example 5.3.

an ordering of blocks 1–7 respectively from left to right.

$$N = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

The design in Example 5.3 is from a class of designs called 2-designs. These designs have the property that every pair of points appear together in a fixed number λ of blocks together and are denoted 2-$(v, b, r, \gamma, \lambda)$. For the design in Example 5.3, each pair of points (academics) occurs in one block (committee) together thus the blocks in $\mathcal{B}$ form a 2-design with $v = b = 7$, $\gamma = r = 3$, and $\lambda = 1$. The 2-designs with $\lambda = 2$, called Steiner 2-designs, are particularly useful for LDPC codes. An LDPC code is defined by setting the incidence matrix of the design as the parity-check matrix of the code. Designs which are regular give regular LDPC codes and sparse codes are defined by choosing designs with γ and r small relative to v and b. In particular 4-cycle free LDPC codes are guaranteed by choosing Steiner 2-designs since each pair of points (rows of H) cannot occur in more than one block (column of H) together and so a 4-cycle cannot be formed.

Combinatorial designs and error correction codes have a long history together. The blocks of a design can be associated with the codewords of a code, as for the *geometric codes*, which have minimum weight codewords the incidence vectors of the blocks of a projective or Euclidean geometry designs. The minimum weight codewords of Reed-Muller and punctured Reed Muller codes are the blocks of the PG$(m, 2)$ designs while the generalized Reed-Muller codes have as minimum weight codewords the blocks of the geometries PG(m, q).

Designs have also played a role in defining new codes such as in the case of difference set cyclic codes. In this case the codes were defined using the transpose of the incidence matrix of the projective geometry design, PG$(2, q)$, as the code parity-check matrix. The properties of these projective geometry designs are well suited to the majority logic decoding algorithm. More recently these codes have had an important impact on the field of iterative decoding

when it was shown that the properties that make them majority logic decodable also make them excellent LDPC codes.

From a combinatorial perspective, a design is generally associated with a code of length v defined as the column space of the design incidence matrix N, called its *block code* or a code defined as the column space of the design incidence matrix transpose N^T, called its *point code* . The block code of a design can be thought of as the code with generator matrix given by N^T. Most of the general results about designs in codes including the celebrated Assmus–Mattson theorem [30] consider these block codes.

For LDPC codes the dual codes of the block and point codes are of interest. The dual of the block (respectively point) codes have as their dual space the column space of N (respectively N^T). Thus the incidence matrix of the design, or its transpose, is the parity-check matrix of the code. The dual of the point code, using the incidence matrix of a design as the parity-check matrix of the code, in particular, are used to define LDPC codes.

Finite geometries

An area closely related to designs is that of finite geometries. The finite projective geometry of a vector space V of dimension $m + 1$, PG(V), has as elements the subspaces of V. The points of PG(V) are the 1-dimensional subspaces of V, the lines are 2-dimensional subspaces of V, the planes are 3-dimensional subspaces of V and so on to hyperplanes the m-dimensional subspaces of V. The incidence between elements of PG(V) corresponds to containment between subspaces of V. Thus a point P is incident with a line L in PG(V) if the 1-dimensional subspace corresponding to P is contained in the 2-dimensional subspace corresponding to L. For V a vector space of dimension $m + 1$ over the field $F =$ GF(q), the projective geometry is often written PG(m, q). A Euclidean geometry EG(V) has as elements the cosets $x + U$ of the subspaces U of V where x is any vector in V and incidence is again given by containment.

Designs can be formed by taking as points of the design the points of the geometries and as blocks the lines, planes or hyperplanes of the geometry with the incidence of the geometry carried into the design. The designs consisting of the points and lines of PG$(2, q)$ are *finite projective planes* of order q. The PG designs, which give us PG-LDPC codes, are the set of $q^2 + q + 1$ lines and $q^2 + q + 1$ points such that every line passes through exactly $q + 1$ points and every point is incident on exactly $q + 1$ lines. Since, any pair of points in the plane must be incident together in exactly one line. The points and lines of a projective plane are the points and blocks of a 2-$(q^2 + q + 1, q + 1, 1)$ design with the incidence of the design given by the incidence of the plane. Fig. 5.5 shows the typical representation of the finite projective plane of order 3. The designs of points and lines of PG$(m, 2)$ are the classical Steiner triple systems or 2-$(v, 3, 1)$ designs which lead to STS-LDPC codes.

Example 5.5.

Figure 5.5 shows the finite projective plane of order 3 which consists of 13 points on 13 lines.

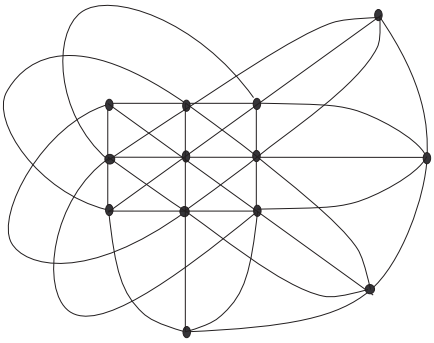

Figure 5.5: The finite projective plane of order 3 consists of 13 points on 13 lines

An important outcome of the work with algebraic codes was the demonstration that highly redundant parity-check matrices can lead to very good iterative decoding performances without the need for very long block lengths. While the probability of a random graph having a highly redundant parity-check matrix is vanishingly small, the field of combinatorial designs offers a rich source of algebraic constructions for matrices which are both sparse and redundant.

Example 5.6.

Starting with the Euclidean geometry EG$(2, 2^4)$ the EG design is the $2^{2s} - 1$ points of the geometry not including the origin and blocks of the design are the $2^{2s} - 1$ lines of the geometry which do not pass through the origin. The incidence matrix of this EG design is thus a square 255×255 matrix with column and row weights both 16. Although this incidence matrix is square it has a large number of linearly dependent rows, and rank 80.

The dual of the block code of this EG design, i.e. the code with parity-check matrix N, produces a length 255 rate-$175/255$ LDPC code with a $16, 16$-regular parity-check matrix.

Figure 5.6 shows the bit error rate performance on an AWGN channel of a short EG LDPC code from a Euclidean geometry compared to an LDPC constructed pseudo-randomly using Neal's construction. Although both codes have the same length and rate the EG code has significantly more rows in its parity-check matrix, and a much greater minimum distance, of 17, which gives it its improved performance.

Partial geometries

LDPC codes have also been defined from a more general class of designs called *partial geometries*. A partial geometry is a set of points, and subsets of those points, called blocks or lines, completely specified by three parameters, s, t, and α. A partial geometry, denoted pg(s, t, α), satisfies the following properties:

P1. Each point P is incident with $t + 1$ blocks and each block B is incident with $s + 1$ points.

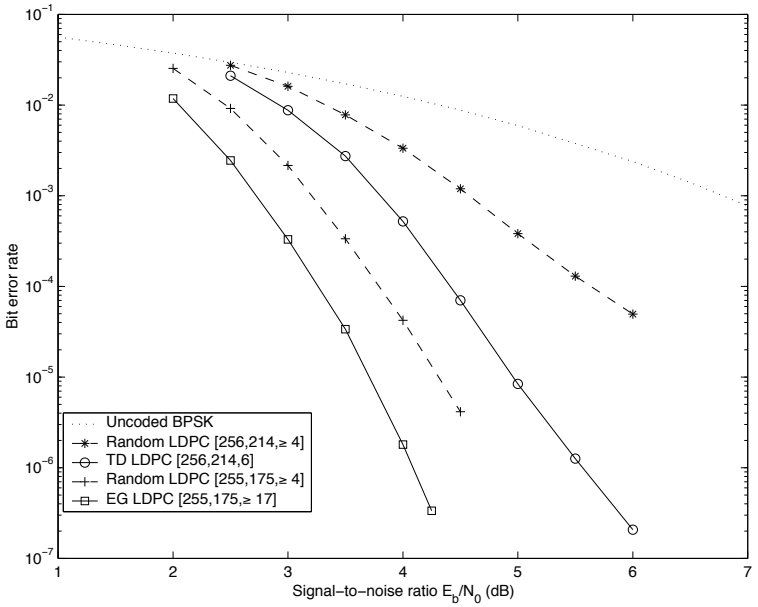

Figure 5.6: The decoding performance of length-256 LDPC codes on an AWGN channel using sum-product decoding with a maximum of 200 iterations.

P2. Any two blocks have at most one point in common.

P3. For any non-incident point-block pair (P, B) the number of blocks incident with P and intersecting B equals some constant α.

Example 5.7.

The incidence matrix of the partial geometry $pg(1, 2, 1)$ is:

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

Fig 5.7 gives its incidence graph.

The subset of the partial geometries with $\alpha = s + 1$ are exactly Steiner 2-designs since if a point P is not incident in a block B, every block incident with P must intersect B and thus every pair of points must appear in a block together. The four main classes of partial geometries are:

- a partial geometry with $\alpha = s + 1$ is a Steiner 2-design or 2-$(v, s + 1, 1)$ design,
- a partial geometry with $\alpha = t$ is called a *net* or, dually with $\alpha = s$, a *transversal design* (TD),

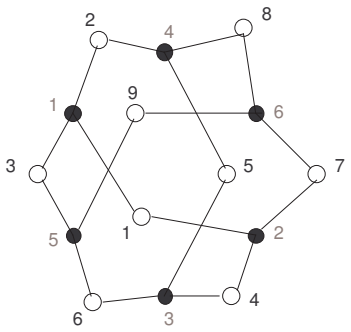

Figure 5.7: The incidence graph for the partial geometry pg(1,2,1) in Example 5.7.

- a partial geometry with $\alpha = 1$ is called a *generalized quadrangle* (GQ),
- if $1 < \alpha < \min\{s, t\}$ the partial geometry is *proper*,

The transversal designs, generalized quadrangles, and partial geometries also make good LDPC codes. The generalized quadrangles in particular can define LDPC codes with girth 8.

Example 5.8.

The incidence matrix of the transversal design, with $\alpha = s = 2$ and $t = 15$, produces the parity-check matrix of a length-256 rate-214/256, $(3, 16)$-regular LDPC code. Figure 5.6 also shows the bit error rate performance on an AWGN channel of a short LDPC code from a transversal design compared to an LDPC constructed pseudo-randomly using Neal's Algorithm.

In addition to a deterministic construction and guaranteed lower bounds on girth and minimum distance the LDPC codes from combinatorial designs can also produce codes which offer straightforward encoders. Many of the STS, Euclidean and projective geometry designs produce cyclic and quasi-cyclic codes. For example the quasi-cyclic code in Example 4.6 was derived from a cyclicly resolvable STS design. However, even for designs which are not cyclic, straightforward encoding can be achieved using the structure of the design.

Further quasi-cyclic LDPC codes can be constructed explicitly using combinatorial structures called difference families.

Difference families

Row-circulant quasi-cyclic LDPC codes can be constructed using combinatorial deigns called *difference families*. A difference family is an arrangement of a group of v elements into not necessarily disjoint subsets of equal size which meet certain difference requirements. More precisely: The t γ-element subsets, called base blocks, of an Abelian group $\mathcal{G}$, $D_1, \ldots, D_t$ with $D_i = \{d_{i,1}, d_{i,2}, \ldots, d_{i,\gamma}\}$ form a (v, γ, λ) difference family if the differences $d_{i,x} - d_{i,y}$, $(i = 1, \ldots t;\ x, y = 1, \ldots, \gamma, x \neq y)$ give each non-zero element

of $\mathcal{G}$ exactly λ times. If the Abelian group is Z_v each translate is a cyclic shift and the difference family is a cyclic difference family.

Example 5.9. The subsets $D_1 = \{1, 2, 5\}$, $D_2 = \{1, 3, 9\}$ of Z_{13} form a $(13, 3, 1)$ difference family with differences:

$$\begin{array}{lll} \text{From } D_1: & 2-1=1, & 1-2=12, \quad 5-1=4, \\ & 1-5=9, & 5-2=3, \quad 2-5=10, \\ \text{From } D_2: & 3-1=2, & 1-3=11, \quad 9-1=8, \\ & 1-9=5, & 9-3=6, \quad 3-9=7. \end{array}$$

Difference families with $\lambda = 1$ allow the design of quasi-cyclic codes free of 4-cycles. To construct a length vl rate $\frac{l-1}{l}$ regular quasi-cyclic code $H = [a_1(x), a_2(x), \ldots, a_l(x)]$ with column weight γ, take l of the base blocks of a $(v, \gamma, 1)$ difference family, and define the jth circulant of H as the transpose of the circulant formed from the jth base block in the difference family as follows:

$$a_j(x) = x^{d_{j,1}} + x^{d_{j,2}} + \cdots + x^{d_{j,\gamma}}.$$

5.3 Bibliographic notes

Tanner founded the topic of algebraic methods for constructing graphs suitable for sum-product decoding in [2]. The length 73 finite geometry code was first implemented on an integrated circuit using iterative decoding by Karplus and Krit [31] and many subsequent authors have considered the construction of LDPC codes using designs [32–36], partial geometries [37] and generalized quadrangles [38]. Graph-based constructions for codes with good girth have been presented by Margulis [39], and extended by Rosenthal and Vontobel [40] and Lafferty and Rockmore [41]. While other constructions for LDPC codes have been presented which have a mixture of algebraic and randomly constructed portions [42]. The monograph by Assmus and Key [43] gives an excellent treatment of the connection between codes and designs. For more on designs see [44] and a good source of constructions is [45].

BIBLIOGRAPHY

[1] R. G. Gallager, *Low-Density Parity-Check Codes*. Cambridge, MA: MIT Press, 1963.

[2] R. M. Tanner, "A recursive approach to low complexity codes," *IEEE Trans. Inform. Theory*, vol. IT-27, no. 5, pp. 533–547, September 1981.

[3] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, D. A. Spielman, and V. Stemann, "Practical loss-resilient codes," in *Proc. 30th ACM Symp. on the Theory of Computing*, 1998, pp. 249–258.

[4] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. A. Spielman, "Improved low-density parity-check codes using irregular graphs," *IEEE Trans. Inform. Theory*, vol. 47, no. 2, pp. 585–598, February 2001.

[5] S.-Y. Chung, G. D. Forney, Jr., T. J. Richardson, and R. L. Urbanke, "On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit," *IEEE Commun. Letters*, vol. 5, no. 2, pp. 58–60, February 2001.

[6] T. J. Richardson and R. L. Urbanke, "Efficient encoding of low-density parity-check codes," *IEEE Trans. Inform. Theory*, vol. 47, no. 2, pp. 638–656, February 2001.

[7] D. J. C. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE Trans. Inform. Theory*, vol. 45, no. 2, pp. 399–431, March 1999.

[8] S. B. Wicker, *Error Control Systems for Digital Communication and Storage*. Upper Saddle River, NJ 07458: Prentice Hall, 1995.

[9] S. Lin and D. J. Costello Jr., *Error Control Coding*, 2nd ed. New Jersey: Prentice Hall, 2004.

[10] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*. Amsterdam: North-Holland, 1977.

[11] W. C. Huffman and V. Pless, *Fundamentals of Error Correcting Codes*. Cambridge University Press, Cambridge UK, 2003.

[12] C. Berrou and A. Glavieux, "Near optimum error correcting coding and decoding: Turbo codes," *IEEE Trans. Commun.*, vol. 44, no. 10, pp. 1261–1271, October 1996.

[13] A. Papoulis, *Probability, random variables and stochastic processes. 2nd Ed*. Singapore: McGraw-Hill, 1984.

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

[15] M. G. Luby, M. Mitzenmacher, and Shokrollahi, "Analysis of random processes via and-or tree evaluation," *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 364–373, 1998.

[16] T. J. Richardson, M. A. Shokrollahi, and R. L. Urbanke, "Design of capacity-approaching irregular low-density parity-check codes," *IEEE Trans. Inform. Theory*, vol. 47, no. 2, pp. 619–637, February 2001.

[17] A. Amraoui and R. L. Urbanke, *LdpcOpt:* ⟨http://lthcwww.epfl.ch/research/ldpcopt/⟩.

[18] D. Hayes, S. Weller, and S. Johnson, *LODE:* ⟨http://sigpromu.org/ldpc/DE/⟩.

[19] M. Sipser and D. A. Spielman, "Expander codes," *IEEE Trans. Inform. Theory*, vol. 42, no. 6, pp. 1710–1722, November 1996.

[20] M. Karlin, "New binary coding results by circulants," *IEEE Trans. Inform. Theory*, vol. IT-15, no. 1, pp. 81–92, 1969.

[21] D. J. C. MacKay and R. M. Neal, "Near Shannon limit performance of low density parity check codes," *Electron. Lett.*, vol. 32, no. 18, pp. 1645–1646, March 1996, reprinted *Electron. Lett*, vol. 33(6), pp. 457–458, March 1997.

[22] C. Di, D. Proietti, I. E. Telatar, T. J. Richardson, and R. L. Urbanke, "Finite-length analysis of low-density parity-check codes on the binary erasure channel," *IEEE Trans. Inform. Theory*, vol. 48, no. 6, pp. 1570–1579, June 2002.

[23] T. J. Richardson and R. L. Urbanke, "Finite-length density evolution and the distribution of the number of iterations on the binary erasure channel," unpublished manuscript, available at ⟨http://lthcwww.epfl.ch/papers/RiU02.ps⟩.

[24] T. J. Richardson, M. A. Shokrollahi, and R. L. Urbanke, "Finite-length analysis of various low-density parity-check ensembles for the binary erasure channel," in *Proc. International Symposium on Information Theory (ISIT'2002)*, Lausanne, Switzerland, June 30 – July 5 2002, p. 1.

[25] S. J. Johnson and S. R. Weller, "Constraining LDPC degree distributions for improved error floor performance," *IEEE Commun. Letters*, 2006.

[26] P. Votobel, ⟨http://www.hpl.hp.com/personal/Pascal Vontobel/pseudocodewords/⟩.

[27] R. L. Townsend and E. J. Weldon, "Self-orthogonal quasi-cyclic codes," *IEEE Trans. Inform. Theory*, vol. IT-13, no. 2, pp. 183–195, April 1967.

[28] S. Lin and D. Costello, Jr., *Error Control Coding: Fundamentals and Applications*, ser. Prentice-Hall Series in Computer Applications in Electrical Engineering. Englewood Cliffs, N. J. 07632f: Prentice-Hall, Inc., 1983.

[29] M. P. C. Fossorier, "Quasi-cyclic low-density parity-check codes from circulant permutation matrices," *IEEE Trans. Inform. Theory*, vol. 50, no. 8, pp. 1788–1793, Aug 2004.

[30] E. F. Assmus, Jr. and H. F. Mattson, Jr., "New 5-designs," *J. Combin. Theory*, vol. 6, pp. 122–151, 1969.

[31] K. Karplus and H. Krit, "A semi-systolic decoder for the PDSC-73 error-correcting code," *Discrete Applied Math*, vol. 33, no. 1–3, pp. 109–128, November 1991.

[32] J. L. Fan, *Constrained Coding and Soft Iterative Decoding*, ser. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 2001.

[33] R. Lucas, M. P. C. Fossorier, Y. Kou, and S. Lin, "Iterative decoding of one-step majority logic decodable codes based on belief propagation," *IEEE Trans. Commun.*, vol. 48, no. 6, pp. 931–937, June 2000.

[34] 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. Inform. Theory*, vol. 47, no. 7, pp. 2711–2736, November 2001.

[35] S. J. Johnson and S. R. Weller, "Resolvable 2-designs for regular low-density parity-check codes," *IEEE Trans. Commun.*, vol. 51, no. 9, pp. 1413–1419, September 2003.

[36] B. Vasic, "Structured iteratively decodable codes based on Steiner systems and their application in magnetic recording," in *Proc. IEEE Globecom Conf.*, San Antonio, TX, November 2001, pp. 2954–2960.

[37] S. J. Johnson and S. R. Weller, "Codes for iterative decoding from partial geometries," *IEEE Trans. Commun.*, vol. 52, no. 2, pp. 236–243, February 2004.

[38] P. O. Vontobel and R. M. Tanner, "Construction of codes based on finite generalized quadrangles for iterative decoding," in *Proc. International Symposium on Information Theory (ISIT'2001)*, Washington, DC, June 24–29 2001, p. 223.

[39] G. A. Margulis, "Explicit constructions for graphs without short cycles and low density codes," *Combinatorica*, vol. 2, no. 1, pp. 71–78, 1982.

[40] P. Vontobel, "*Algebraic Coding for Iterative Decoding*," Ph.D. dissertation, Swiss Federal Institute of Technology, Zurich, 2003.

[41] J. Lafferty and D. Rockmore, "Codes and iterative decoding on algebraic expander graphs," in *Proc. International Symposium on Information Theory and its applications (ISITA2000)*, Hawaii, USA, November 5–8 2000.

[42] J. W. Bond, S. Hui, and H. Schmidt, "Constructing low-density parity-check codes with circulant matrices," in *Proc. IEEE Information Theory Workshop (ITW1999)*, Metsovo, Greece, June 27 – July 1 1999, p. 52.

[43] E. F. Assmus, Jr. and J. D. Key, *Designs and their Codes*, ser. Cambridge Tracts in Mathematics. Cambridge, U.K.: Cambridge University Press, 1993, vol. 103.

[44] P. J. Cameron and J. H. van Lint, *Graphs, Codes and Designs*, ser. London Mathematical Society Lecture Note Series, No. 43. Cambridge: Cambridge University Press, 1980.

[45] I. Anderson, *Combinatorial Designs: Construction Methods*, ser. Mathematics and its Applications. Chichester: Ellis Horwood, 1990.