A Tutorial on Low Density Parity-Check Codes

Tuan Ta
The University of Texas at Austin

Abstract − **Low density parity-check codes are one of the hottest topics in coding theory nowadays. Equipped with very fast encoding and decoding algorithms (probabilistically,) LDPC are very attractive both theoretically and practically. In this paper, I present a throughout view of LDPC. I discuss in detail LDPC's popular decoding algorithm: belief propagation. I also present some noticeable encoding algorithms including Richardson and Urbanke's lower triangular modification that gives rise to a linear encoding time. I show that irregular LDPC perform better than regular LDPC thus are more desirable.**

Index Terms—**belief propagation, density evolution, Gallager codes, irregular codes, low density parity-check codes**

I. INTRODUCTION

LOW density parity-check code (LDPC) is an error correcting code used in noisy communication channel to reduce the probability of loss of information. With LDPC, this probability can be reduced to as small as desired, thus the data transmission rate can be as close to Shannon's limit as desired.

LDPC was developed by Robert Gallager in his doctoral dissertation at MIT in 1960 [1]. It was published 3 years latter in MIT Press. Due to the limitation in computational effort in implementing the coder and decoder for such codes and the introduction of Reed-Solomon codes, LDPC was ignored for almost 30 years. During that long period, the only notable work done on the subject was due to R. Michael Tanner in 1981 [2] where he generalized LDPC codes and introduced a graphical representation of the codes later called Tanner graph. Since 1993, with the invention of turbo codes, researchers switched their focus to finding low complexity code which can approach Shannon channel capacity. LDPC was reinvented with the work of Mackay [3], [4] and Luby [5]. Nowadays, LDPC have made its way into some modern applications such as 10GBase-T Ethernet, WiFi, WiMAX, Digital Video Broadcasting (DVB).

Before discussing LDPC, I present a brief review of error correcting code in section II. I pay the most attention to linear block code. (7, 4) Hamming code is introduced as an example to illustrate the behavior of a linear block code. I also discuss the use of Tanner graph as equivalence to a parity-check matrix for linear block codes. Section II is concluded by the definition of LDPC. In section III, I discuss how to decode an LDPC using *belief propagation* algorithm. Both the hard-decision decoder and the soft-decision decoder are presented. An example is given to illustrate the use of the algorithm. Section IV talks about how to encode an LDPC in linear time. Two approaches are presented, the *accumulate approach,* and Richardson and Urbanke's *lower triangular modification approach*. Section V discusses *irregular* LDPC and shows that irregular codes perform better than regular codes. Three designs are considered: Luby's, Richardson's and Chung's. Section V shows that it is possible for irregular LDPC to come as close as 0.0045 dB to Shannon capacity.

II. ERROR CORRECTING CODES

A. Introduction

In communication, errors can occur due to a lot of reasons: noisy channel, scratches on CD or DVD, power surge in electronic circuits, etc. It is often desirable to detect and correct those errors. If no additional information is added to the original message, errors can turn a legal message (bit pattern) into another legal bit pattern. Therefore, redundancy is used in error-correcting schemes. By adding redundancy, a lot of bit patterns will become illegal. A good coding scheme will make an illegal pattern caused by errors to be closer to one of the legal patterns than others.

A metric used to measure the "closeness" between two bit patterns is Hamming distance. The Hamming distance between two bit patterns is the number of bits that are different. For example, bit pattern 1100 and 0100 differ by one bit (the 1st bit), thus have Hamming distance of one. Two identical bit patterns have Hamming distance of zero.

A parity bit is an additional bit added to the bit pattern to make sure that the total number of 1's is even (even parity) or odd (odd parity). For example, the information message is 01001100 and even parity is used. Since the number of 1's in the original message is 3, a '1' is added at the end to give the transmitted message of 010011001. The decoder counts the number of 1's (normally done by exclusive OR the bit stream) to determine if an error has occurred. A single parity bit can detect (but not correct) any odd number of errors. In the previous example, the *code rate* (number of useful information bits/total number of bits) is 8/9. This is an efficient code rate, but the effectiveness is limited. Single parity bit is often used in scenarios where the likelihood of errors is small and the receiver is able to request retransmission. Sometimes it is used even when retransmission is not possible. Early IBM Personal Computers employ single bit parity and simple crash when an error occurs [6].

B. *Linear block code*

If a code uses *n* bits to provide error protection to *k* bits of information, it is called a (*n, k*) block code. Often times, the minimum Hamming distance *d* between any two valid codewords is also included to give a (*n, k, d*) block code. An example of block codes is Hamming code. Consider the scenario where we wish to correct single error using the fewest number of parity bits (highest code rate). Each parity bit gives the decoder a parity equation to validate the received code. With 3 parity bits, we have 3 parity equations, which can identify up to $2^3 = 8$ error conditions. One condition identifies "no error", so seven would be left to identify up to seven places of single error. Therefore, we can detect and correct any single error in a 7-bit word. With 3 parity bits, we have 4 bits left for information. Thus this is a (7, 4) block code. In a (7, 4) Hamming code, the parity equations are determined as follow:

- The first parity equation checks bit 4, 5, 6, 7
- The second parity equation checks bit 2, 3, 6, 7
- The third parity equation checks bit 1, 3, 5, 7

This rule is easy to remember. The parity equations use the binary representation of the location of the error bit. For example, location 5 has the binary representation of 101, thus appears in equation 1 and 3. By applying this rule, we can tell which bit is wrong by reading the value of the binary combination of the result of the parity equations, with 1 being incorrect and 0 being correct. For example, if equation 1 and 2 are incorrect and equation 3 is correct, we can tell that the bit at location 6 (110) is wrong.

At the encoder, if location 3, 5, 6, 7 contain the original information and location 1, 2, 4 contain the parity bits (locations which are power of 2) then using the first parity equation and bits at location 5, 6, 7, we can calculate the value of the parity bit at location 4 and so on.

(7, 4) Hamming code can be summarized in the following table [7]

TABLE 1
(7, 4) HAMMING CODE FULL TABLE

Bit number	**1**	**2**	**3**	**4**	**5**	**6**	**7**
	p_1	p_2	d_1	p_3	d_2	d_3	d_4
Equation corresponds to p_1	1	0	1	0	1	0	1
Equation corresponds to p_2	0	1	1	0	0	1	1
Equation corresponds to p_3	0	0	0	1	1	1	1

In table 1, p denotes parity bits, d denotes data bits.

Removing the parity bits, we have

TABLE 2
(7, 4) HAMMING CODE ABBREVIATED TABLE

	d_1	d_2	d_3	d_4
p_1	1	1	0	1
p_2	1	0	1	1
p_3	0	1	1	1

For larger dimension block codes (larger *n* and *k*), a matrix representation of the codes is used. This representation includes a *generator matrix*, ***G***, and a *parity-check matrix*, ***H***. Given a message ***p***, the codeword will be the product of ***G*** and ***p*** with entries modulo 2:

$$\boldsymbol{c} = \boldsymbol{Gp} \tag{1}$$

Given the received codeword y, the *syndrome vector* is

$$\boldsymbol{z} = \boldsymbol{Hy} \tag{2}$$

If $\boldsymbol{z} = \boldsymbol{0}$ then the received codeword is error-free, else the value of z is the position of the flipped bit. For the (7, 4) Hamming code, the parity-check matrix is

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{3}$$

and the generator matrix is

$$\mathbf{G} = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4}$$

H is driven straight from Table 1. ***G*** is obtained by

- For parity bit locations: use associated data bits (from Table 2)
- For data bit locations: put a 1 for the position of the data bit, the rest are 0's

For the generator matrix of (7, 4) Hamming code above, bit location 1 (1st row) is a parity bit, thus we use row 1 from table 2 (1101). Bit location 5 (5th row) is a data bit, and bit 5 is data bit number 2, thus we set bit 2 to 1 (0100).

If the message is

$$\mathbf{p} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

then the codeword will be

$$\mathbf{c} = \mathbf{Gp} = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 1 \\ 2 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

If no bit is flipped during transmission, in other words, $\boldsymbol{y} = \boldsymbol{c}$. Then the syndrome vector is

$$\mathbf{z} = \mathbf{Hy} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

If the 6th bit is flipped,

$$\mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

then

$$\mathbf{z} = \mathbf{Hy} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$$

Reading $\boldsymbol{z}$ from the bottom up (higher position first), we see the flipped bit is indeed 6 (110).

C. *Tanner graph*

A very useful way of representing linear block codes is using Tanner graph. Tanner graph is a bipartite graph, which means the graph is separated into two partitions. These partitions are called by different names: subcode nodes and digit nodes, variable nodes and check nodes, message nodes and check nodes. I will call them message nodes and check nodes from now on. Tanner graph maps directly to the parity-check matrix $\boldsymbol{H}$ of a linear block code, with check nodes represent the rows of $\boldsymbol{H}$. The Tanner's graph of (7, 4) Hamming code is shown below, with check nodes on the left and message nodes on the right.

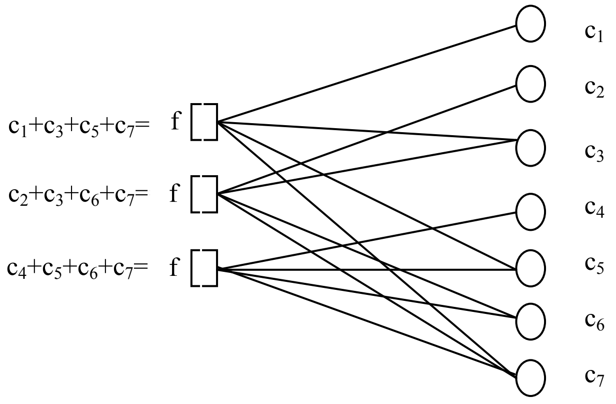


Figure 1: Tanner graph for (7, 4) Hamming code

As seen above, there is an edge connects a check node with a message node if the message node is included in the check node's equation. From a Tanner graph, we can deduce a parity-check matrix by putting a 1 at position (i, j) if there is an edge connecting f_i and c_j. The code defined by a Tanner graph (or a parity-check matrix) is the set of vectors $\boldsymbol{c} = (c_1,\dots,c_n)$ such that $\boldsymbol{Hc^T = 0}$. In other words, the code forms the null space of $\boldsymbol{H}$.

D. *Low Density Parity-check Code (LDPC)*

Any linear code has a bipartite graph and a parity-check matrix representation. But not all linear code has a sparse representation. A $n \times m$ matrix is sparse if the number of 1's in any row, the *row weight* w_r, and the number of 1's in any column, the *column weight* w_c, is much less than the dimension ($w_r << m, w_c << n$). A code represented by a sparse parity-check matrix is called low density parity-check code (LDPC). The sparse property of LDPC gives rise to its algorithmic advantages. An LDPC code is said to be *regular* if w_c is constant for every column, w_r is constant for every row and $w_r = w_c \frac{n}{m}$. An LDPC which is not regular is called *irregular*.

III. DECODING

Different authors come up independently with more or less the same iterative decoding algorithm. They call it different names: the sum-product algorithm, the belief propagation algorithm, and the message passing algorithm. There are two derivations of this algorithm: hard-decision and soft-decision schemes.

A. Hard-decision Decoder

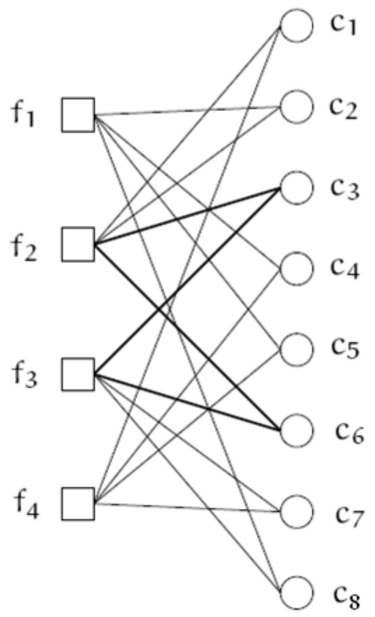


Figure 2: Belief propagation example code

In [8], Leiner uses a (4, 8) linear block code to illustrate the hard-decision decoder. The code is represented in Figure 2, its corresponding parity-check matrix is

$$\mathbf{H} = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{bmatrix} \qquad (5)$$

An error free codeword of $\boldsymbol{H}$ is $\mathbf{c} = [1\ 0\ 0\ 1\ 0\ 1\ 0\ 1]^T$. Suppose we receive $\mathbf{y} = [1\ 1\ 0\ 1\ 0\ 1\ 0\ 1]^T$. So c_2 was flipped. The algorithm is as follow:

1. In the first step, all message nodes send a message to their connected check nodes. In this case, the message is the bit they believe to be correct for them. For example, message node c_2 receives a 1 ($y_2=1$), so it sends a message containing 1 to check nodes f_1 and f_2. Table 3 illustrates this step.

2. In the second step, every check nodes calculate a response to their connected message nodes using the messages they receive from step 1. The response message in this case is the value (0 or 1) that the check node believes the message node has based on the information of other message nodes connected to that check node. This response is calculated using the parity-check equations which force all message nodes connect to a particular check node to sum to 0 (mod 2).

   In Table 3, check node f_1 receives 1 from c_4, 0 from c_5, 1 from c_8 thus it believes c_2 has 0 (1+0+1+0=0), and sends that information back to c_2. Similarly, it receives 1 from c_2, 1 from c_4, 1 from c_8 thus it believes c_5 has 1 (1+1+1+1=0), and sends 1 back to c_5.

   At this point, if all the equations at all check nodes are satisfied, meaning the values that the check nodes calculate match the values they receive, the algorithm terminates. If not, we move on to step 3.

3. In this step, the message nodes use the messages they get from the check nodes to decide if the bit at their position is a 0 or a 1 by majority rule. The message nodes then send this hard-decision to their connected check nodes. Table 4 illustrates this step. To make it clear, let us look at message node c_2. It receives 2 0's from check nodes f_1 and f_2. Together with what it already has $y_2 = 1$, it decides that its real value is 0. It then sends this information back to check nodes f_1 and f_2.

4. Repeat step 2 until either exit at step 2 or a certain number of iterations has been passed.

In this example, the algorithm terminates right after the first iteration as all parity-check equations have been satisfied. *c_2* is corrected to 0.

TABLE 3
CHECK NODES ACTIVITIES FOR HARD-DECISION DECODER FOR CODE OF FIGURE 2

check nodes	activities				
f_1	receive	$c_2 \rightarrow 1$	$c_4 \rightarrow 1$	$c_5 \rightarrow 0$	$c_8 \rightarrow 1$
	send	$0 \rightarrow c_2$	$0 \rightarrow c_4$	$1 \rightarrow c_5$	$0 \rightarrow c_8$
f_2	receive	$c_1 \rightarrow 1$	$c_2 \rightarrow 1$	$c_3 \rightarrow 0$	$c_6 \rightarrow 1$
	send	$0 \rightarrow c_1$	$0 \rightarrow c_2$	$1 \rightarrow c_3$	$0 \rightarrow c_6$
f_3	receive	$c_3 \rightarrow 0$	$c_6 \rightarrow 1$	$c_7 \rightarrow 0$	$c_8 \rightarrow 1$
	send	$0 \rightarrow c_3$	$1 \rightarrow c_6$	$0 \rightarrow c_7$	$1 \rightarrow c_8$
f_4	receive	$c_1 \rightarrow 1$	$c_4 \rightarrow 1$	$c_5 \rightarrow 0$	$c_7 \rightarrow 0$
	send	$1 \rightarrow c_1$	$1 \rightarrow c_4$	$0 \rightarrow c_5$	$0 \rightarrow c_7$

TABLE 4
MESSAGE NODES DECISIONS FOR HARD-DECISION DECODER FOR CODE OF FIGURE 2

message nodes	y_i	messages from check nodes		decision
c_1	1	$f_2 \rightarrow 0$	$f_4 \rightarrow 1$	1
c_2	1	$f_1 \rightarrow 0$	$f_2 \rightarrow 0$	0
c_3	0	$f_2 \rightarrow 1$	$f_3 \rightarrow 0$	0
c_4	1	$f_1 \rightarrow 0$	$f_4 \rightarrow 1$	1
c_5	0	$f_1 \rightarrow 1$	$f_4 \rightarrow 0$	0
c_6	1	$f_2 \rightarrow 0$	$f_3 \rightarrow 1$	1
c_7	0	$f_3 \rightarrow 0$	$f_4 \rightarrow 0$	0
c_8	1	$f_1 \rightarrow 1$	$f_3 \rightarrow 1$	1

B. Soft-decision Decoder

The soft-decision decoder operates with the same principle as the hard-decision decoder, except that the messages are the conditional probability that the received bit is a 1 or a 0 given the received vector ***y***.

Let $P_i = \Pr[c_i = 1 \mid \mathbf{y}]$ be the conditional probability that c_i is a 1 given the value of ***y***. We have $\Pr[c_i = 0 \mid \mathbf{y}] = 1 - P_i$.

Let $q_{ij}^{(l)}$ be the message sent by message node c_i to check node f_j at round *l*. Every message contains a pair $q_{ij}^{(l)}(0)$ and $q_{ij}^{(l)}(1)$ which stands for the "amount of belief" that y_i is 0 or 1, $q_{ij}^{(l)}(0) + q_{ij}^{(l)}(1) = 1$. In particular, $q_{ij}^{(0)}(1) = P_i$ and $q_{ij}^{(0)}(0) = 1 - P_i$.

Similarly, let $r_{ji}^{(l)}$ be the message sent by check node f_j to message node c_i at round *l*. Every message contains a pair $r_{ji}^{(l)}(0)$ and $r_{ji}^{(l)}(1)$ which stands for the "amount of belief" that y_i is 0 or 1. We also have $r_{ji}^{(l)}(0) + r_{ji}^{(l)}(1) = 1$. $r_{ji}^{(l)}(0)$ is also the probability that there are an even number of 1's on all other message nodes rather than c_i.

First, let consider the probability that there are an even number of 1's on 2 message nodes. Let q_1 be the probability that there is a 1 at message node c_1 and q_2 be the probability that there is a 1 at message node c_2. We have

$$\begin{aligned}\Pr[c_1 \oplus c_2 = 0] &= q_1 q_2 + (1-q_1)(1-q_2) \\ &= 1 - q_1 - q_2 + 2q_1 q_2 \\ &= \frac{1}{2}(2 - 2q_1 - 2q_2 + 4q_1 q_2) \\ &= \frac{1}{2}[1 + (1-2q_1)(1-2q_2)] = q \quad (6)\end{aligned}$$

Now consider the probability that there are an even number of 1's on 3 message nodes, c_1, c_2 and c_3. Note that $1 - q$ is the probability that there are an odd number of 1's on c_1 and c_2.

$$\Pr[(c_1 \oplus c_2) \oplus c_3 = 0]$$

$$= \frac{1}{2}[1 + (1 - 2(1-q))(1 - 2q_3)]$$

$$= \frac{1}{2}[1 + (1 - 2q_1)(1 - 2q_2)(1 - 2q_3)] \quad (7)$$

In general

$$\Pr[c_1 \oplus \ldots \oplus c_n = 0] = \frac{1}{2} + \frac{1}{2}\prod_{i=1}^{n}(1 - 2q_i) \quad (8)$$

Therefore, the message that f_j sends to c_i at round l is

$$r_{ji}^{(l)}(0) = \frac{1}{2} + \frac{1}{2}\prod_{i' \in V_j \neq i}(1 - 2q_{i'j}^{(l-1)}(1)) \quad (9)$$

$$r_{ji}^{(l)}(1) = 1 - r_{ji}^{(l)}(0) \quad (10)$$

where V_j is the set of all message nodes connected to check node f_j.

The message that c_i sends to f_j at round l is

$$q_{ij}^{(l)}(0) = k_{ij}(1 - P_i)\prod_{j' \in C_i \neq j} r_{j'i}^{(l-1)}(0) \quad (11)$$

$$q_{ij}^{(l)}(1) = k_{ij} P_i \prod_{j' \in C_i \neq j} r_{j'i}^{(l-1)}(1) \quad (12)$$

where C_i is the set of all check nodes connected to message node c_i.

The constant k_{ij} is chosen so that

$$q_{ij}(0) + q_{ij}(1) = 1$$

At each message node, the following calculations are made

$$Q_i^{(l)}(0) = k_i(1 - P_i)\prod_{j \in C_i} r_{ji}^{(l)}(0) \quad (13)$$

$$Q_i^{(l)}(1) = k_i P_i \prod_{j \in C_i} r_{ji}^{(l)}(1) \quad (14)$$

$Q_i^{(l)}$ is the effective probability of 0 and 1 at message node c_i at round l. If $Q_i^{(l)}(1) > Q_i^{(l)}(0)$ then the estimation at this point is $c_i = 1$, otherwise $c_i = 0$. If this estimation satisfies the parity-check equations then the algorithm terminates. Else, the algorithm runs through a predetermined number of iterations.

As seen above, this algorithm uses a lot of multiplications which are costly to implement. Another approach is to use logarithmic likelihood ratio. Let

$$L_i = \left(\frac{\Pr[c_i = 0 \mid \mathbf{y}]}{\Pr[c_i = 1 \mid \mathbf{y}]}\right) = \frac{1 - P_i}{P_i} \quad (15)$$

$$l_i = \ln L_i = \ln\left(\frac{\Pr[c_i = 0 \mid \mathbf{y}]}{\Pr[c_i = 1 \mid \mathbf{y}]}\right) \quad (16)$$

L_i is the likelihood ratio and l_i is the log likelihood ratio at message node c_i. Using log ratio turns multiplications into additions which are much cheaper to implement in hardware.

With log likelihood ratio, we have

$$P_i = \frac{1}{1 + L_i} \quad (17)$$

From (11) and (12), the message that c_i sends to f_j at round l is

$$m_{ij}^{(l)} = \ln\frac{q_{ij}^{(l)}(0)}{q_{ij}^{(l)}(1)} = \ln\left[\frac{1 - P_i}{P_i}\prod_{j' \in C_i \neq j}\frac{r_{j'i}^{(l-1)}(0)}{r_{j'i}^{(l-1)}(1)}\right]$$

$$= l_i + \sum_{j' \in C_i \neq j} m_{j'i}^{(l-1)} \quad (18)$$

From (9) and (10), the message that f_j sends to c_i at round l is

$$m_{ji}^{(l)} = \ln\frac{r_{ji}^{(l)}(0)}{r_{ji}^{(l)}(1)} = \ln\frac{\frac{1}{2} + \frac{1}{2}\prod_{i' \in V_j \neq i}(1 - 2q_{i'j}^{(l-1)}(1))}{\frac{1}{2} - \frac{1}{2}\prod_{i' \in V_j \neq i}(1 - 2q_{i'j}^{(l-1)}(1))}$$

$$= \ln\frac{1 + \prod_{i' \in V_j \neq i}\tanh(\frac{m_{i'j}^{(l-1)}}{2})}{1 - \prod_{i' \in V_j \neq i}\tanh(\frac{m_{i'j}^{(l-1)}}{2})} \quad (19)$$

We have (19) because from (18),

$$e^{m_{i'j}} = \frac{1 - q_{i'j}(1)}{q_{i'j}(1)} \quad (20)$$

Thus

$$q_{i'j}(1) = \frac{1}{1+e^{m_{i'j}}} \tag{21}$$

and

$$1-2q_{i'j}(1) = \frac{e^{m_{i'j}}-1}{e^{m_{i'j}}+1} = \tanh(\frac{m_{i'j}}{2}) \tag{22}$$

Equation (13), (14) turn into

$$l_i^{(l)} = \ln\frac{Q_i^{(l)}(0)}{Q_i^{(l)}(1)} = l_i^{(0)} + \sum_{j\in C_i} m_{ji}^{(l)} \tag{23}$$

If $l_i^{(l)} > 0$ then $c_i = 0$ else $c_i = 1$.

In practice, belief propagation is executed for a maximum number of iterations or until the passed likelihoods are closed to certainty, whichever comes first. A certain likelihood is $l_i = \pm\infty$, where $P_i = 0$ for $l_i = \infty$ and $P_i = 1$ for $l_i = -\infty$.

One very important aspect of belief propagation is that its running time is linear to the code length. Since the algorithm traverses between check nodes and message nodes, and the graph is sparse, the number of traversals is small. Moreover, if the algorithm runs a fixed number of iterations then each edge is traversed a fixed number of times, thus the number of operations is fixed and only depends on the number of edges. If we let the number of check nodes and message nodes increases linearly with the code length, the number of operations performed by belief propagation also increases linearly with the code length.

C. Performance of Belief Propagation

A parameter to measure the performance of belief propagation is the expected fraction of incorrect messages passed at the *l*th iteration, $P_e^n(l)$. In [9], Richardson and Urbanke show that

1. For any $\delta > 0$, the probability that the actual fraction of incorrect messages passed among any instance at round 1 that lies outside $(P_e^n(l)-\delta, P_e^n(l)+\delta)$ converges to zero exponentially fast with n.
2. $\lim_{n\to\infty} P_e^n(l) = P_e^\infty(l)$, where $P_e^\infty(l)$ is the expected fraction of incorrect messages passed at round l assuming that the graph does not contain any cycle of length $2l$ or less. The assumption is to ensure that the decoding neighborhoods become "tree-like" so that the messages are independent for l rounds [10]. The value $P_e^\infty(l)$ can be calculated by a method called *density evolution.* For a message alphabet of size q, $P_e^\infty(l)$ can be expressed by means of $q-1$ coupled recursive functions.
3. There exists a channel parameter σ^* with the following property: if $\sigma < \sigma^*$ then $\lim_{l\to\infty} P_e^\infty(l) = 0$, else if $\sigma > \sigma^*$ then there exists a constant $\gamma(\sigma) > 0$ such that $P_e^\infty(l) > \gamma(\sigma)$ for all $l \geq 1$. Here σ^2 is the noise variance in the channel. In other words, σ^* sets the limit to which belief propagation decodes successfully.

IV. ENCODING

If the generator matrix ***G*** of a linear block code is known then encoding can be done using equation (1). The cost (number of operations) of this method depends on the Hamming weights (number of 1's) of the basis vectors of ***G.*** If the vectors are dense, the cost of encoding using this method is proportional to n^2. This cost becomes linear with n if ***G*** is sparse.

However, LDPC is given by the null space of a sparse parity-check matrix ***H***. It is unlikely that the generator matrix ***G*** will also be sparse. Therefore the straightforward method of encoding LDPC would require number of operations proportional to n^2. This is too slow for most practical applications. Therefore it is desirable to have encoding algorithms that run in linear time. This section will look at two approaches to achieve that goal.

A. Accumulate approach

The first approach modifies LDPC code so it has an inherited fast encoding algorithm. In this case, we assign a value to each check node which is equal to the sum of all message nodes that are connected to it. (It would be more appropriate to talk about information nodes and redundant nodes, but for consistence of notation, I will use message nodes and check nodes.) The number of summations needed to calculate the value of a check node is bounded by the number of message nodes connected to a check node. This is a constant when the code is sparse. The message consists of the message nodes appended by the values of the check nodes. To illustrate the difference between this modified version of LDPC and the original version, consider Figure 2. If Figure 2 represents an original LDPC then c_1, c_2, c_3, c_4 are information bits and c_5, c_6, c_7, c_8 are parity bits which have to be calculated from c_1, c_2, c_3, c_4 by solving the parity-check equations in f_1, f_2, f_3, f_4. The code rate is 4/8 = 1/2. Now if Figure 2 represents a modified LDPC, then all of $c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8$ are information bits; while f_1, f_2, f_3, f_4 are redundant bits calculated from $c_1,\dots,c_8$. f_1 is connected to c_2, c_4, c_5, c_8 so $f_1 = c_2 + c_4 + c_5 + c_8$ and so on. The codeword in this case is $[c_1 \dots c_8\ f_1 \dots f_4]^T$. The code rate is 8/12 = 2/3.

Although this approach gives a linear encoder, it causes a major problem at the decoder. In case the channel is erasure, the value of the check nodes might be erased. On the contrary, the check nodes of the original LDPC are *dependencies*, not *values*, thus they are not affected by the channel. In other words, a check node defines a relationship of its connected message nodes. This relationship comes straight from the parity-check matrix. The fact that in modified LDPC, the values of check nodes can be erased creates a lower bounded for the convergence of any decoding algorithm. In [r4], Shokrollahi proves the existence of such lower bound. Suppose that the channel is erasure with the erasure probability p. Then an expected p-fraction of the message nodes and an expected p-fraction of the check nodes will be erased. Let M_d be the fraction of message nodes of degree d (connected with d check nodes.) The probability of a message node of degree d having all its connected check nodes erased is p^d. This probability is conditioned on the event that the degree of the message node is d. Since the graph is created randomly, the probability that a message node has all its connected check nodes erased is $\sum_d M_d p^d$, which is a constant independent of the length of the code. Therefore, no algorithm can recover the value of that message node.

B. Lower triangular modification approach

In [11], Richardson and Urbanke propose an encoding algorithm that has effectively linear running time for any code with a sparse parity-check matrix. The algorithm consists of two phases: preprocessing and encoding.

In the preprocessing phase, $\boldsymbol{H}$ is converted into the form shown in Figure 3 by row and column permutations.

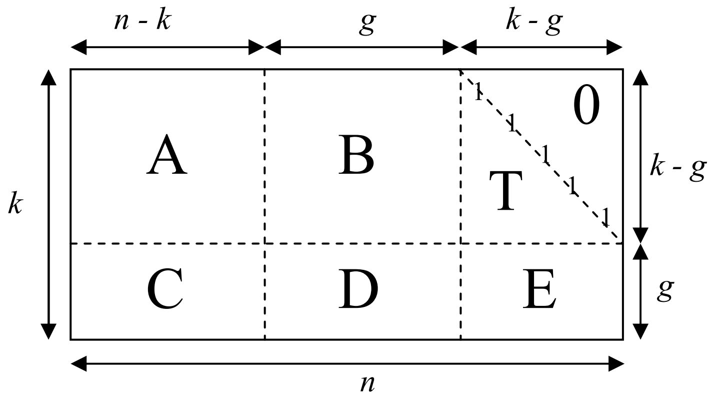


Figure 3: Parity-check matrix in approximately lower triangular form

In matrix notation,

$$\mathbf{H} = \begin{bmatrix} \mathbf{A} & \mathbf{B} & \mathbf{T} \\ \mathbf{C} & \mathbf{D} & \mathbf{E} \end{bmatrix} \qquad (24)$$

where $\boldsymbol{T}$ has a lower triangular form with all diagonal entries equal to 1. Since the operation is done by row and column permutations and $\boldsymbol{H}$ is sparse, $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D}, \boldsymbol{E}, \boldsymbol{T}$ are also sparse. g, the *gap*, measures how close $\boldsymbol{H}$ can be made, by row and column permutations, to a lower triangular matrix.

TABLE 5

COMPUTING p_1 USING RICHARDSON AND URBANKE'S ENCODING ALGORITHM

Operation	**Comment**	**Complexity**
$\mathbf{As}^{\mathbf{T}}$	Multiplication by sparse matrix	O(n)
$\mathbf{T}^{-1}[\mathbf{As}^{\mathbf{T}}]$	Back-substitution, **T** is lower triangular	O(n)
$-\mathbf{E}[\mathbf{T}^{-1}\mathbf{As}^{\mathbf{T}}]$	Multiplication by sparse matrix	O(n)
$\mathbf{Cs}^{\mathbf{T}}$	Multiplication by sparse matrix	O(n)
$[-\mathbf{ET}^{-1}\mathbf{As}^{\mathbf{T}}]+[\mathbf{Cs}^{\mathbf{T}}]$	Addition	O(n)
$-\mathbf{\Phi}^{-1}(-\mathbf{ET}^{-1}\mathbf{As}^{\mathbf{T}}+\mathbf{Cs}^{\mathbf{T}})$	Multiplication by $g\times g$ matrix	O($n+g^2$)

TABLE 6

COMPUTING p_2 USING RICHARDSON AND URBANKE'S ENCODING ALGORITHM

Operation	**Comment**	**Complexity**
$\mathbf{As}^{\mathbf{T}}$	Multiplication by sparse matrix	O(n)
$\mathbf{Bp_1}^{\mathbf{T}}$	Multiplication by sparse matrix	O(n)
$[\mathbf{As}^{\mathbf{T}}]+[\mathbf{Bp_1}^{\mathbf{T}}]$	Addition	O(n)
$-\mathbf{T}^{-1}(\mathbf{As}^{\mathbf{T}}+\mathbf{Bp_1}^{\mathbf{T}})$	Back-substitution, **T** is lower triangular	O(n)

Multiple $\boldsymbol{H}$ from the left by

$$\begin{bmatrix} \mathbf{I} & \mathbf{0} \\ -\mathbf{ET}^{-1} & \mathbf{I} \end{bmatrix} \tag{25}$$

we get

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} & \mathbf{T} \\ -\mathbf{ET}^{-1}\mathbf{A}+\mathbf{C} & -\mathbf{ET}^{-1}\mathbf{B}+\mathbf{D} & \mathbf{0} \end{bmatrix} \tag{26}$$

Let the codeword $\boldsymbol{c} = (\boldsymbol{s}, \boldsymbol{p_1}, \boldsymbol{p_2})$ where $\boldsymbol{s}$ is the information bits, $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ are the parity-check bits, $\boldsymbol{p_1}$ has length g, $\boldsymbol{p_2}$ has length $k-g$.

By $\boldsymbol{Hc^T} = \boldsymbol{0}$, we have

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

$$= \begin{bmatrix} \mathbf{A} & \mathbf{B} & \mathbf{T} \\ -\mathbf{ET}^{-1}\mathbf{A}+\mathbf{C} & -\mathbf{ET}^{-1}\mathbf{B}+\mathbf{D} & \mathbf{0} \end{bmatrix}\begin{bmatrix} \mathbf{s} \\ \mathbf{p}_1 \\ \mathbf{p}_2 \end{bmatrix} = \mathbf{0} \tag{27}$$

Therefore

$$\mathbf{As}^{\mathbf{T}} + \mathbf{Bp_1}^{\mathbf{T}} + \mathbf{Tp_2}^{\mathbf{T}} = \mathbf{0} \tag{28}$$

$$(-\mathbf{ET}^{-1}\mathbf{A} + \mathbf{C})\mathbf{s}^{\mathbf{T}} + (-\mathbf{ET}^{-1}\mathbf{B} + \mathbf{D})\mathbf{p_1}^{\mathbf{T}} = \mathbf{0} \tag{29}$$

The procedure to find $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$ is summarized in Table 5 and 6.

Define: $\mathbf{\Phi} = -\mathbf{ET}^{-1}\mathbf{B} + \mathbf{D}$ and assume for the moment that $\mathbf{\Phi}$ is nonsingular. Then

$$\mathbf{p_1}^{\mathbf{T}} = -\mathbf{\Phi}^{-1}(-\mathbf{ET}^{-1}\mathbf{A} + \mathbf{C})\mathbf{s}^{\mathbf{T}}$$

$$= -\mathbf{\Phi}^{-1}(-\mathbf{ET}^{-1}\mathbf{As}^{\mathbf{T}} + \mathbf{Cs}^{\mathbf{T}}) \tag{30}$$

First we compute $\mathbf{As}^{\mathbf{T}}$. Since **A** is sparse, this is done in linear time O(n). Then we compute $\mathbf{T}^{-1}[\mathbf{As}^{\mathbf{T}}] = \mathbf{y}^{\mathbf{T}}$. Since $[\mathbf{As}^{\mathbf{T}}] = \mathbf{Ty}^{\mathbf{T}}$ and **T** is lower triangular, by back-substitution we can compute $\mathbf{y}^{\mathbf{T}}$ in linear time. The calculations $-\mathbf{Ey}^{\mathbf{T}}$ and $\mathbf{Cs}^{\mathbf{T}}$ are also done in O(n) as **E, C** are sparse. Now we have $(-\mathbf{ET}^{-1}\mathbf{As}^{\mathbf{T}} + \mathbf{Cs}^{\mathbf{T}}) = \mathbf{z}^{\mathbf{T}}$ computed in O(n). Since $\mathbf{\Phi}$ is $g\times g$, $\boldsymbol{p_1}$ is computed from (30) in O($n+g^2$).

From (28), $\mathbf{p_2}^{\mathbf{T}} = -\mathbf{T}^{-1}(\mathbf{As}^{\mathbf{T}} + \mathbf{Bp_1}^{\mathbf{T}})$. The steps to calculate $\boldsymbol{p_2}$ are quite similar and are shown in Table 6. We see that $\boldsymbol{p_2}$ can be computed in O(n).

As seen in Table 5 and Table 6, $\boldsymbol{c}$ can be computed in O($n+g^2$). Richardson and Urbanke prove in [11] that the gap g concentrates around its expected value, αn, with high probability. α here is a small constant. For a regular LDPC with row weight $w_r = 6$, column weight $w_c = 3$, $\alpha = 0.017$. Therefore even though mathematically the encoding algorithm run in $O(n^2)$ (α^2 $O(n^2)$ to be precise), in practice the encoder still runs in reasonable time for n = 100,000. In the same paper, Richardson and Urbanke also show that for known "optimized" codes, the expected g is bounded by $O(\sqrt{n})$ thus the encoder runs in $O(n)$.

V. IRREGULAR CODES

It has been shown that irregular LDPC perform better than regular LDPC [12], [13], [14]. The idea was pioneered by Luby *et al* in [12]. He thinks of finding coefficients for an irregular code as a game, with the message nodes and check nodes as players. Each player tries to choose the right number of edges for them. A constraint of the game is that the message nodes and the check nodes must agree on the total number of edges. From the point of view of the message nodes, it is best to have high degree since the more information it has from the check nodes, the more accurately it can judge what its correct value should be. On the other hand, from the point of view of the check nodes, it is best to have low degree, since the lower the degree of a check node, the more valuable the information it can transmit back to the message nodes. These two requirements must be appropriately balanced to have a good code.

MacKay shows in [15], [16] that for regular codes, it is best to have low density. However, allowing irregular codes provides another degree of freedom. In [12], Luby shows that having a wide spread of degree is advantageous, at least for the message nodes. The reason is message nodes with high degree tend to correct their value faster. These nodes then provide good information to the check nodes, which subsequently provide better information to the lower degree message nodes. Therefore irregular graph has potential to provide a wave effect where high degree message nodes are corrected first, followed by slightly smaller degree nodes, and so on.

Before getting into the details of how to construct irregular codes, let us introduce some notations. Let d_l, d_r be the maximum degrees of message nodes and check nodes. Define the left (right) degree of an edge to be the degree of the message node (check node) that is connected to the edge. λ_i (ρ_i) is the fraction of edges with left (right) degree i. Any LDPC graph is specified by the sequences $(\lambda_1,\ldots,\lambda_{d_l})$ and $(\rho_1,\ldots,\rho_{d_r})$. Further, define

$$\lambda(x) = \sum_i \lambda_i x^{i-1} \tag{31}$$

and

$$\rho(x) = \sum_i \rho_i x^{i-1} \tag{32}$$

to be the degree distribution of message nodes and check nodes. Also, define p_i to be the probability that an incorrect message is passed in the ith iteration.

Now consider a pair of message node and check node (m, c) and let c' be another check node of m different than c. At the end of the ith iteration, c' will send m its correct value if there are an even number (including 0) of message nodes other than m sending c' the incorrect bit. By an analogous analysis to equation (8), the probability that c' receives an even number of errors is

$$\frac{1+(1-2p_i)^{d_r-1}}{2} \tag{33}$$

for the case of unvarying degrees of the nodes, and

$$\frac{1+\rho(1-2p_i)}{2} \tag{34}$$

for the case of varying degrees of the nodes, where $\rho(x)$ is defined in (32).

A. *Luby's design*

In [12], Luby proves the iterative description of p_i.

$$p_{i+1} = p_0 - \sum_{j=1}^{d_l} \lambda_j \cdot \left[p_0 \sum_{t=b_{i,j}}^{j} \binom{j-1}{t} \left[\frac{1+\rho(1-2p_i)}{2}\right]^t \left[\frac{1-\rho(1-2p_i)}{2}\right]^{j-1-t} + (1-p_0) \sum_{t=b_{i,j}}^{j} \binom{j-1}{t} \left[\frac{1+\rho(1-2p_i)}{2}\right]^t \cdot \left[\frac{1-\rho(1-2p_i)}{2}\right]^{j-1-t} \right] \quad (35)$$

Note that p_0 is the error probability of the channel. $b_{i,j}$ is given by the smallest integer the satisfies

$$\frac{1-p_0}{p_0} \le \left[\frac{1+\rho(1-2p_i)}{1-\rho(1-2p_i)}\right]^{2b_{i,j}-j+1} \quad (36)$$

The goal of this design is to find sequences λ = $(\lambda_1, \ldots, \lambda_{d_l})$ and ρ = $(\rho_1, \ldots, \rho_{d_r})$ that yield the biggest value of p_0 such that the sequence $\{p_i\}$ decreases to 0. Define

$$f(x) = p_0 - \sum_{j=1}^{d_l} \lambda_j \cdot \left[p_0 \sum_{t=b_{i,j}}^{j} \binom{j-1}{t} \left[\frac{1+\rho(1-2x)}{2}\right]^t \left[\frac{1-\rho(1-2x)}{2}\right]^{j-1-t} + (1-p_0) \sum_{t=b_{i,j}}^{j} \binom{j-1}{t} \left[\frac{1+\rho(1-2x)}{2}\right]^t \cdot \left[\frac{1-\rho(1-2x)}{2}\right]^{j-1-t} \right] \quad (37)$$

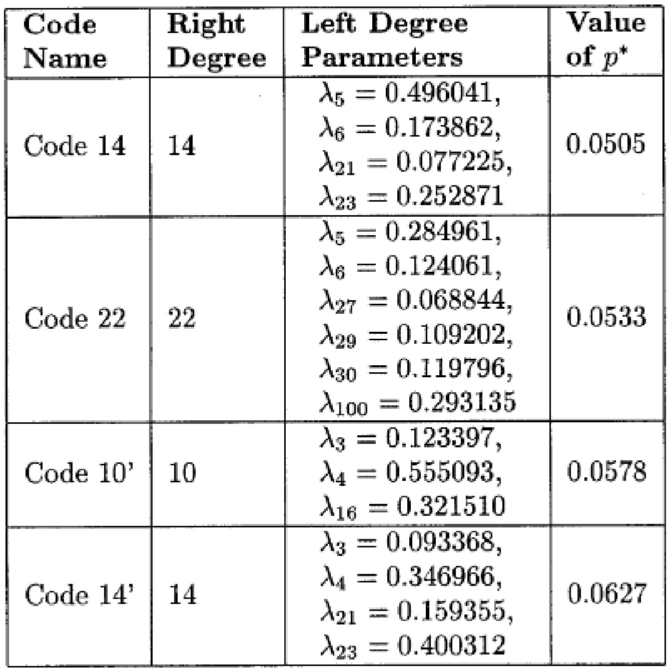
So p_{i+1} = f(p_i), therefore we want f(x) < x. Another constraint to λ and ρ is

$$\sum_l \frac{\lambda_l}{l} = (1-R)\sum_i \frac{\rho_i}{i} \quad (38)$$

Equation (38) makes sure that the total number of left and right degrees are the same. Luby's approach tries to find *any* sequence λ that satisfies (38) and f(x) < x for $x \in (0, p_0)$. It accomplishes this task by examining the conditions at $x = p_0 / N$ for some integer N. By plugging those values of x into (37), it creates a system of linear inequalities. The algorithm finds any λ that satisfies this linear system. As seen, Luby's approach cannot determine the best sequence λ and ρ. Instead, it determines a good vector λ given a vector ρ and a desired code rate R.

Luby shows through simulations that the best codes have constant ρ, in other words, the check nodes have the same degree. Some results from [r6] is reproduced in Table 7.

TABLE 7
LUBY'S IRREGULAR CODES



Code Name	Right Degree	Left Degree Parameters	Value of p^*
Code 14	14	$\lambda_5 = 0.496041$, $\lambda_6 = 0.173862$, $\lambda_{21} = 0.077225$, $\lambda_{23} = 0.252871$	0.0505
Code 22	22	$\lambda_5 = 0.284961$, $\lambda_6 = 0.124061$, $\lambda_{27} = 0.068844$, $\lambda_{29} = 0.109202$, $\lambda_{30} = 0.119796$, $\lambda_{100} = 0.293135$	0.0533
Code 10'	10	$\lambda_3 = 0.123397$, $\lambda_4 = 0.555093$, $\lambda_{16} = 0.321510$	0.0578
Code 14'	14	$\lambda_3 = 0.093368$, $\lambda_4 = 0.346966$, $\lambda_{21} = 0.159355$, $\lambda_{23} = 0.400312$	0.0627

p^* in Table 7 is the maximum value of p_0

achieved by each code. All of the code above have code rate R = 1/2. Previously, the best p^* for Gallager's regular codes with code rate 1/2 is 0.0517 [1].

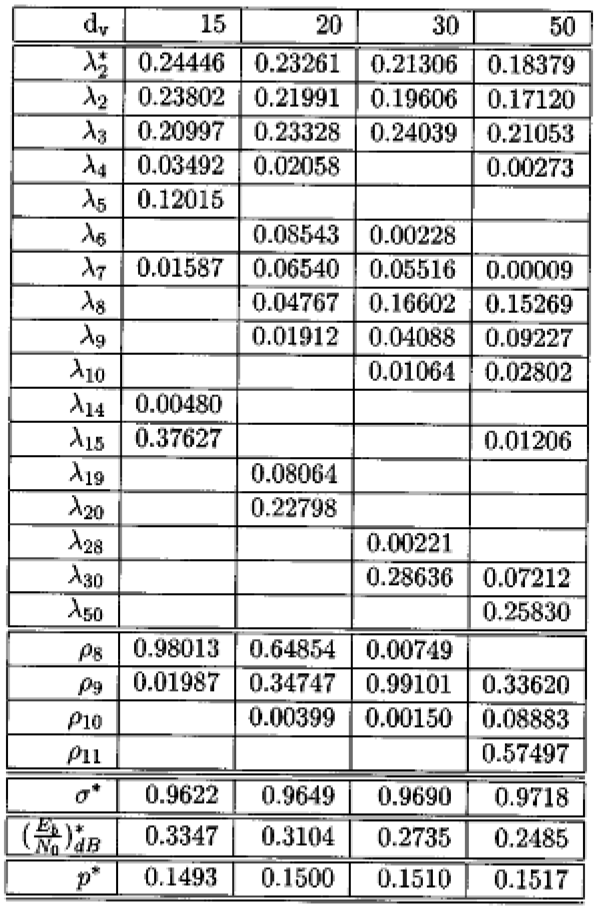
B. Richardson's design

In [13], Richardson, Shokrollahi and Urbanke propose a design of irregular LDPC that can approach Shannon channel capacity tighter than turbo code. Their algorithm employs two optimizations: tolerate the error floor for practical purpose and carefully design quantization of density evolution to match the quantization of messages passed. The idea of the former optimization is that in practice, we always allow a finite (but small) probability of error ε. If we choose ε small enough then it automatically implies convergence. The latter optimization makes sure that the performance loss due to quantization errors is minimized. Since belief propagation is optimal, the quantized version is suboptimal, therefore the simulation results can be thought of as lower bound for actual values.

Richardson's algorithm starts with an arbitrary degree distribution (λ, ρ). It sets the target error probability ε and the maximum number of iterations *m*. The algorithm searches for the maximum *admissible* channel parameter such that belief propagation returns a probability of error less than ε after *m* iterations. Now slightly change the degree distribution pair and runs the algorithm again and check if a larger admissible channel parameter or a lower probability of error is found. If yes then set the current distribution pair to the new distribution pair, else keep the original pair. This process is repeated a large number of times. The basic of this algorithm is that Richardson notices the existence of *stable regions* where the probability of error does not decrease much with the increase number of iterations. This fact helps limit the search space of the degree distribution thus shortens the running time.

Another optimization in Richardson's algorithm is the fact that he lets the degree to be a *continuous* variable, and round it to return to real integer degree. The reason why this optimization works is because it suits *Differential Evolution* well. The detail of Differential Evolution is discussed in [17]. Some results from [13] is reproduced in Table 8.

TABLE 8
RICHARDSON'S IRREGULAR CODES



d_v	15	20	30	50
λ_2^*	0.24446	0.23261	0.21306	0.18379
λ_2	0.23802	0.21991	0.19606	0.17120
λ_3	0.20997	0.23328	0.24039	0.21053
λ_4	0.03492	0.02058		0.00273
λ_5	0.12015			
λ_6		0.08543	0.00228	
λ_7	0.01587	0.06540	0.05516	0.00009
λ_8		0.04767	0.16602	0.15269
λ_9		0.01912	0.04088	0.09227
λ_{10}			0.01064	0.02802
λ_{14}	0.00480			
λ_{15}	0.37627			0.01206
λ_{19}		0.08064		
λ_{20}		0.22798		
λ_{28}			0.00221	
λ_{30}			0.28636	0.07212
λ_{50}				0.25830
ρ_8	0.98013	0.64854	0.00749	
ρ_9	0.01987	0.34747	0.99101	0.33620
ρ_{10}		0.00399	0.00150	0.08883
ρ_{11}				0.57497
σ^*	0.9622	0.9649	0.9690	0.9718
$(\frac{E_b}{N_0})^*_{dB}$	0.3347	0.3104	0.2735	0.2485
p^*	0.1493	0.1500	0.1510	0.1517

d_v is the maximum message node degree, for each d_v, the individual degree fraction is provided. σ^* is the channel parameter discussed in Section III – C. p^* is the input bit error probability of a hard-decision decoder. All codes have rate 1/2.

C. Chung's design

In his PhD dissertation, Chung introduces a derivation of density evolution called *discretized density evolution.* This derivation is claimed to model exactly the behavior of discretized belief propagation. In his letter [14], Chung introduces an irregular code which is within 0.0045 dB of Shannon capacity. This code has d_v = 8000, which is much greater than the maximum message node degree studied by Luby and Richardson. Chung's code is the closest code to Shannon capacity that has been simulated. It further confirms that LDPC indeed approaches channel capacity.

VI. CONCLUSION

This paper summaries the important concepts regarding low density parity-check code (LDPC). It goes through the motivation of LDPC and how LDPC can be encoded and decoded. Different modifications of the codes are presented, especially irregular codes. I chose to leave out derivations of regular codes such as MacKay codes [18], repeat-accumulate codes [19] because they have become less important with the advance of irregular codes.

This paper however has not mentioned how LDPC is implemented in real hardware. For this, I refer the readers to [20], where a decoder design based on IEEE 802.11n standards with very high throughput (900Mbps for FPGA, 2Gbps for ASIC design) is discussed.

REFERENCES

[1] R. Gallager, "Low density parity-check codes," *IRE Trans, Information Theory,* pp. 21-28. January 1962.

[2] R. M. Tanner, "A recursive approach to low complexity codes," *IEEE Trans. Information Theory,* pp. 533-547, September 1981.

[3] D. Mackay and R. Neal, "Good codes based on very sparse matrices," *Cryptography and Coding, 5th IMA Conf.,* C. Boyd, Ed., *Lecture Notes in Computer Science,* pp. 100-111, Berlin, Germany, 1995.

[4] D. Mackay, "Good error correcting codes based on very sparse matrices," *IEEE Trans. Information Theory,* pp 399-431, March 1999.

[5] N. Alon and M. Luby, "A linear time erasure-resilient code with nearly optimal recovery," *IEEE Trans. Information Theory,* vol. 47, pp. 6238-656, February 2001.

[6] "Information and Entropy", *MIT OpenCourseWare.* Spring 2008.

[7] Wikipedia. "Hamming(7,4)". Accessed May 01, 2009.

[8] B. M. J. Leiner, "LDPC Codes – a brief Tutorial," April 2005.

[9] T. Richardson and R. Urbanke, "The capacity of low-density parity check codes under message-passing decoding," *IEEE Trans. Inform. Theory*, vol. 47, pp. 599 – 618, 2001.

[10] A. Shokrollahi, "LDPC Codes: An Introduction," *Digital Fountain, Inc.*, April 2, 2003.

[11] T. Richardson and R. Urbanke, "Efficient encoding of low-density parity-check codes," *IEEE Trans. Inform. Theory*, vol. 47, pp. 638 – 656, 2001.

[12] M. Luby, M. Mitzenmacher, A. Shokrollahi, and D. Spielman, "Improved Low-Density Parity-Check Codes Using Irregular Graphs," *IEEE Trans. Inform. Theory,* vol. 47, pp. 585 – 598, 2001.

[13] T. Richardson, A. Shokrollahi, and R. Urbanke, "Design of capacity-approaching irregular low-density parity-check codes," *IEEE Trans. Inform. Theory*, vol. 47, pp. 619 – 637 , 2001.

[14] S.-Y. Chung, D. Forney, T. Richardson, and R. Urbanke, "On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit," *IEEE Communication Letters*, vol. 5, pp. 58 − 60, 2001.

[15] D. J. C. MacKay, "Good error correcting codes based on very sparse matrices," *IEEE Trans. Inform. Theory*, vol. 45, pp. 399–431, Mar. 1999.

[16] D. J. C. MacKay and R. M. Neal, "Near Shannon limit performance of low-density parity-check codes," *Electron. Lett.,* vol. 32, pp. 1645–1646, 1996.

[17] K. Price and R. Storn, “Differential evolution − A simple and efficient heuristic for global optimization over continuous spaces,” *J. Global Optimiz.,* vol. 11, pp. 341–359, 1997.

[18] D. Mackay, “Information Theory, Interference, and Learning Algorithms,” *Cambridge University Press* 2003.

[19] D.Divsalar, H. Jin and R. McEliece, “Coding theorems for turbo-like codes,” Proc. 36th Annual Allerton Conf. on Comm., Control and Conputingce, pp. 201-210. September 1998.

[20] Marjan Karkooti, Predrag Radosavljevic and Joseph R. Cavallaro, “Configurable, High Throughput, Irregular LDPC Decoder Architecture: Tradeoff Analysis and Implementation,” *Rice Digital Scholarship Archive,* September 01, 2006.