Scheduling Techniques of LDPC Error Correcting Codes

Student Seamus Gilroy

Supervisor Dr. Mark Flanagan

School of Electronic and Electrical Engineering

Final Report

BE Project

2014

Final Report
Seamus Gilroy

Contents

Introduction

Systems dedicated to the communication or storage of information are a common facet of everyday life. Broadly speaking, a communication system is a system which allows us to send information from one place to another. Examples include telephone networks, computer networks, audio/video broadcasting, etc. Storage systems, e.g. magnetic and optical disk drives are systems for storage and later retrieving information.

American Electronic Engineer Claude Shannon has long been regarded as ‘The Father of Information Theory’, due to the publication of a landmark paper in 1948 [1]. In this paper, Shannon introduced to us a concept now known as the ‘Shannon limit’. This limit represents the theoretical maximum information transfer rate of the channel, for a particular noise level.

Communication over noisy channels can be improved by the use of a channel code C, as demonstrated by Shannon with his famous channel coding theorem [1]. This theorem tells us that coding can reduce the bit error rate of a communication system to an arbitrarily low value provided the information rate is less than the channel capacity.

While Shannon proposed this theorem, he provided no insight in how to achieve this capacity. The evidence of the search for this coding scheme can be seen by the rapid development of capacity improving schemes. Researchers are continuously looking for ways to improve capacity. Currently, the only measure that can be used for code performance is its proximity to Shannon’s Limit. Shannon’s limit can be expressed in a number of different ways.

Shannon’s limit in bits per second for a band-limited channel is:

$$C = B * log_2(1 + \frac{S}{N})$$

We see the capacity is a function of the Bandwidth(B) and the signal to noise ratio of the link $\left(\frac{S}{N}\right)$.

In 1993, turbo codes were introduced, the first practical codes which came close to the Shannon limit. They did this using iterative decoding techniques, achieving performances to within 0.5dB [2] of the Shannon limit for a bit error rate(BER) of 10^{-5}. Before this, conventional thinking was that there existed a cutoff rate about 2dB above the Shannon Limit on error correction codes without an unbounded increase in processing complexity.

Although people may not be aware of its existence in many applications, the impact of advancements in error correction has been crucial to the development of the Internet, the popularity of compact discs (CD), the feasibility of mobile phones, the success of the deep space missions, etc. Thus we see the study of Error Correcting Codes is an important priority for the development of effective communication systems.

An error correcting code provides us with a means of representing a sequence of numbers, binary digits here, in such a way that we can reliably detect & correct a number of errors. These errors will typically arise due to noise in the communication channel.

All well-designed systems aim at reproducing as reliably as possible while sending as much information as possible per unit time. Simple channel coding schemes allow the receiver of the transmitted data signal to detect errors, while more advanced channel coding schemes provide the ability to recover a finite amount of corrupted data. This gives rise to more reliable communication, and in many cases, eliminates the need for retransmission. It is not possible to detect an error if every possible symbol or set of symbols that can be received is a legitimate message. It is only possible to catch errors if there are some restrictions on what a proper message is.

A primitive example of an error correcting code might be to simply transmit each bit multiple times – a ‘repetition’ code. This increases redundancy in the message, decreasing the rate of information transfer, but helps in terms of correcting errors. However, encoding one bit at a time is clearly a very inefficient system, we get much better results by adding redundancy to blocks of data instead.

Low Density Parity Check (LDPC) codes are a particular type of error-correcting code which were introduced in 1963 by Robert G. Gallager [3], but largely forgotten about until the last decade. LDPC codes are a class of linear block codes.

The biggest difference between LDPC codes and classical block codes is how they are decoded. Classical block codes are generally decoded with Maximum-likelihood 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.

The defining characteristic of the LDPC code is its sparse H matrix, as we will discuss below. Compared to alternative highly structured codes, we find that LDPC codes are only effective at large block lengths. Certain LDPC Codes have been shown to have a capacity extremely close [4] to the Shannon limit, this is of course our primary interest in these codes.

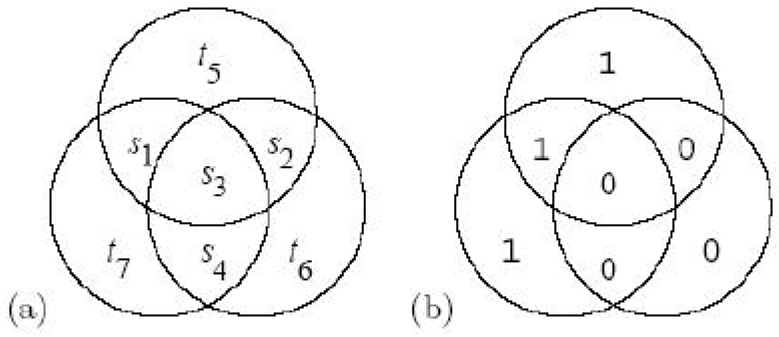
Review

In my early research for the project in order to gain an understanding of the fundamentals of error correcting codes it was necessary to start with more primitive codes. A primary focus of my analysis and early experimentation was on the well-known Hamming Code [5]. This was one of the early advances in finding a code which provides a decent error correction capability together with a respectable code rate. Hamming Code is what is known as a 'block' code. We associate two objects with these type of codes: a generator matrix and a parity-check matrix. The generator matrix works as a compact representation of the code and as a means for efficient encoding. In analyzing the code we use the parity-check matrix. This matrix forms part of the general framework we develop for the decoding of linear codes and it allows us to compute a codes minimum distance.

(7,4) Hamming Code

Hamming (7,4) code is a relatively simple starting point in order to understand channel coding. It is a block code that generates 3 parity bits for every 4 bits of data. Hamming code operates off even parity. Hamming (7,4) is a very simplistic code to understand as it can be graphically represented by a Venn diagram.

This diagram shows how the Hamming(7,4)'s parity bits are calculated. The data bits s_1, s_2, s_3 and s_4 are placed in the middle of the Venn diagram as shown, then the parity bits t_5, t_6, and t_7 are assigned in a manner such that each of the 3 circles has an even number of ones (even parity).



In the figure we see a string $[s_1\ s_2\ s_3\ s_4]$ to be encoded. Let this string be [1 0 0 0]. We see that for all three circles to have even parity, our parity bits t_5 and t_7 must both be a 1, as seen in the figure.

A block code is essentially a rule for converting a sequence of source bits of length k into an encoded sequence of length N bits, ready for transmission. The set of parity check equations show us how this is done:

$$p_1 = s_1 + s_2 + s_3$$

$$p_2 = s_1 + s_3 + s_4$$

$$p_3 = s_2 + s_3 + s_4$$

$$\mathbf{H} := \begin{array}{ccccccc} \left(\begin{matrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{matrix}\right) \\ s_1 \quad s_2 \quad s_3 \quad s_4 \quad p_1 \quad p_2 \quad p_3 \end{array}$$

Fig. Parity check equations and associated Matrix

Shown also, is the codes Parity check matrix, which we can see directly from observing the equations.

The codeword c to be transmitted is found from the original sequence s by the following:

$$c = sG$$

Where G is the generator matrix of the code.

$$\mathbf{G} = \begin{array}{c|c} \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} & \begin{matrix} s_1 \\ s_2 \\ s_3 \\ s_4 \end{matrix} \\ \begin{matrix} C_1 & C_2 & C_3 & C_4 & C_5 & C_6 & C_7 \end{matrix} & \end{array}$$

The additional redundant bits are linear functions of the original K bits, as we can see from the G matrix. Each sequence of 4 bits gain an additional 3 redundant bits in encoding. This means there are 2^4 possible combinations for the original message.

An important concept to understand is that while the resulting codeword is now 7 bits long, there is still only 2^4 possible combinations of the 7 bits. It is this fact that allows us to detect if an error has occurred – If we receive a combination which is not in the set of possible codewords.

The Hamming Distance between two sequences X and Y is defined [5] as the number of positions where X differ from Y. The greater the minimum Hamming Distance of a code, the better the expected performance of the code. If we observe the Hamming Code, we find its minimum distance is 3. This means that given an arbitrary codeword, it will take 3 bits flipping to produce any other possible codeword. Half the minimum distance of a code is the maximum number of bit flips we can reliably expect to correct. Thus with the Hamming Code we can reliably correct any 1 bit error, and detect any 2 bit errors.

It is important to note, that if 3 or more bit errors occur, then the decoder will be unable to correct the bit errors, and in fact may be unable to detect that a bit error occurred.

A large minimum distance makes for a good coding scheme, as it increases the noise immunity in the system. It is often very difficult to determine the minimum distance for a given code. This is the case because there exists 2k possible codewords in any given coding scheme, therefore computing minimum distance requires that $(s^{k-1})(s^{k}-1)$ comparisons be performed.

It is obvious that as the blocklength increases, measuring the minimum distance would require a large amount of computational power. We see this as a problem in analysing LDPC codes [6].

Syndrome Decoding

Decoding Hamming Code is conventionally carried out using 'Syndrome' decoding[3]. This involves the following operation:

$$q = Hy^{T}$$

Where q is what we call the syndrome of the received codeword, y. The syndrome indicates which parity-check constraints are not satisfied by y. In the ideal situation that our transmission picks up no errors, the syndrome will return all zeroes, as shown in the example below:

$$\mathrm{Hy}^{\mathrm{T}} = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

We can see that if we flip a single bit in the codeword, then we will no longer be left with a syndrome of zeroes. From here we can use this information to determine which bit is in error. When the redundancy is small in a block code, such as here in the Hamming Code, we can simply use a lookup table to determine this. This is best demonstrated with an example.

While this method works well for small block codes, syndrome decoding is known to be computationally expensive [7] (NP-complete) for general matrices H and vectors s. Thus this particular method does not scale well and so is not practical for LDPC decoding.

If y is a valid codeword, the syndrome will always return all zeroes. However this system fails for more than one bit error in our codeword. While any 2 bit errors will be detected, they cannot be corrected using this method. Furthermore if we have 3 or more errors the incorrect transmission may itself be a valid codeword, thus the syndrome will be all zeroes.

Hamming Code Simulation

I carried out a simulation of the (7,4)Hamming Code using syndrome decoding. The results are shown below:

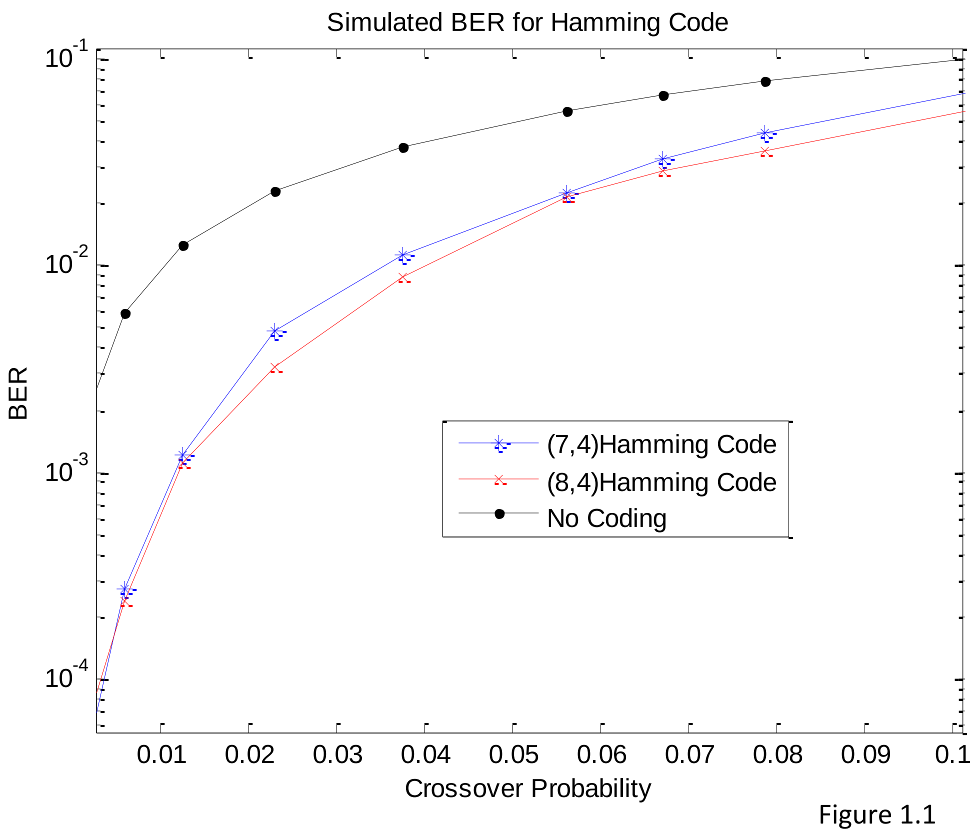


Figure 1.1

We see the (7,4)Hamming code achieves respectable results with moderate error correction capability. Included for reference is the (8,4)Hamming Code, which includes one extra parity bit. We see this extra parity bit allows us a slightly improved BER. However, there is a cost for this.

Code rate

For any encoded binary string with k message bits and length n codewords, the ratio k/n is known as the $rate$ of the code. We see the (8,4) code operates at a slightly slower code rate of ½, compared with $\frac{4}{7}$ for the (7,4) code. Thus we see this is not a fair comparison of the capabilities of the two codes.

For us to draw an effective comparison, we must consider the added requirement of transmitting additional parity bits. The standard way for us to do this is to compare the $\frac{E_b}{N0}$ for various codes, where E_b is the average energy required to transmit a message bit, and $N0$ is the noise power spectral density. For codes of rate ½ it takes twice as much energy to communicate each message bit, since two symbol bits are sent for each message bit.

This relationship can be found as follows:

$$\frac{E_b}{N0} = \frac{1}{R}\frac{E_s}{N0}$$

Representing this in dB we find:

$$\frac{E_b}{N0}(dB) = 10\log_{10}\frac{E_s}{N0} + 10\log_{10}\frac{1}{R}$$

This corresponds to an additional 3.01dB and 2.43dB for the (8,4) and (7,4) codes respectively. Now we can see a better comparison of the codes, as shown below:

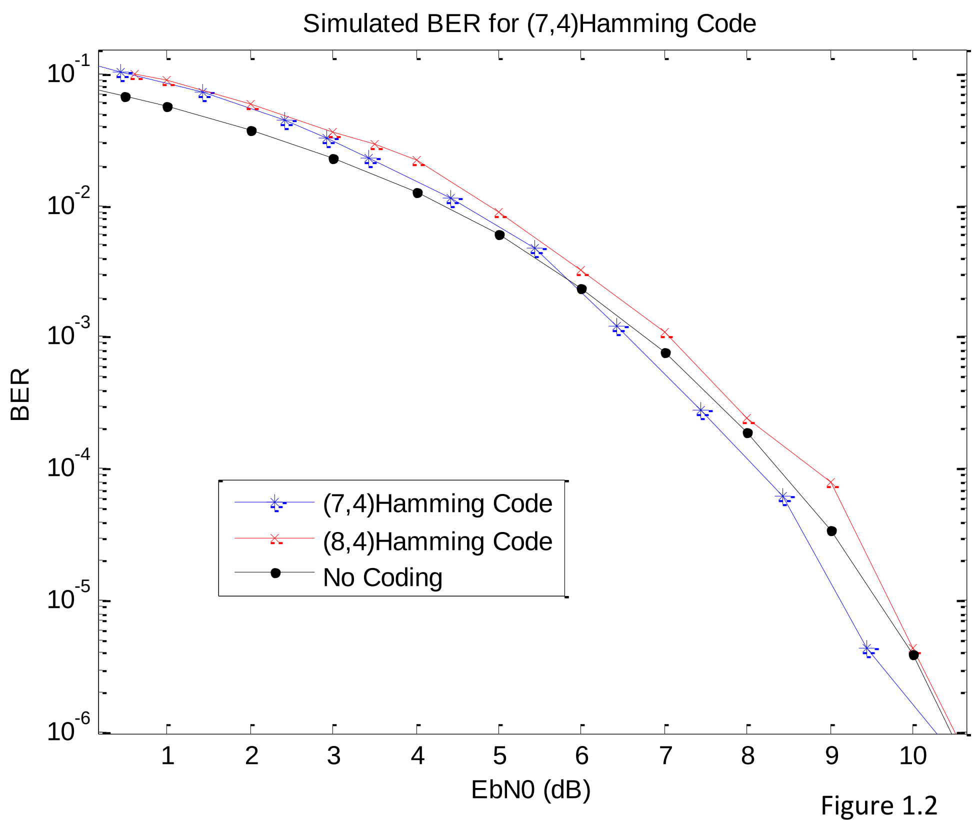


Figure 1.2

We see the faster rate code offers better performance compared to an uncoded system than the slower rate code, despite having an improved capacity for error correction.

This demonstrates an important principle in analysing the practicality of error correcting codes. It is not a difficult task to create good error correcting codes by simply making the rate very low. However it is overall performance we are concerned with.

Low Density Parity Check

Low Density Parity Check Codes are currently at the forefront in state-of-the-art error correcting schemes for Communication systems.

LDPC codes are a class of linear block codes. They are no different to any other linear block codes except for one characteristic. The parity check matrix of an LDPC code is sparse, meaning it has significantly less ones than zeroes. It is this sparseness of the H matrix that ensures a decoding complexity which only increases linearly with the code length. Such codes will typically have no more than 3 ones per column for column lengths exceeding several thousand bits. For my simulations I stick to considerably more modest block lengths of a couple hundred bits long, due to computational limits.

LDPC codes have been shown to be capable of capacity approaching performance over a range of communication channels. The principle advantage of LDPC codes is that there exists an efficient low-complexity algorithm for decoding. Added to this, the decoding process is well suited for hardware implementations that make heavy use of parallelism [8].

The communication channel I deal with in all simulations is the Binary Symmetric Channel (BSC). The BSC is used frequently as it is one of the simplest channels to analyse. In the BSC, the receiver receives a binary bit which we expect will usually be received correctly, but will be flipped with a small probability. We call this probability the ‘crossover probability’.

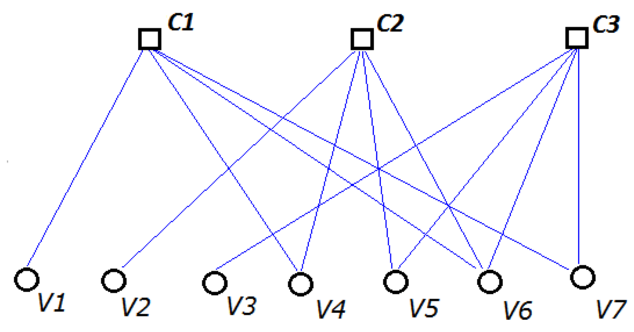
Tanner Graph

Michael Tanner in his 1981 paper [9] introduced an effective graphical representation of LDPC codes, thereafter referred to as ‘Tanner Graphs’. These graphs are simply a graphical means of representing the parity check matrix of a code, but they also help us describe and understand the decoding algorithms used and the concept of ‘Message Passing’, as we will see later.

The premise of a Tanner Graph is as follows:

- A row in the H matrix is represented by a check node
- A column in the H matrix is represented by a variable node
- A 1 in the H matrix represents a connection between a check node and a bit node

If there is a 1 in a particular position in the H matrix, we connect together the check node for the row it is on, with the variable node for the column it is on. In fact a Tanner Graph can be used to represent the parity check matrix of any block code. The Tanner Graph and equivalent H matrix for the (7,4)Hamming Code is given below for illustration purposes:



$$H = \begin{array}{c|ccccccc|c} & 1 & 0 & 0 & 1 & 0 & 1 & 1 & C1 \\ & 0 & 1 & 0 & 1 & 1 & 1 & 0 & C2 \\ & 0 & 0 & 1 & 0 & 1 & 1 & 1 & C3 \\ & V1 & V2 & V3 & V4 & V5 & V6 & V7 & \end{array}$$

Fig. 1.3

We see that the Tanner Graph is a very effective tool for understanding and analysing the decoding process of LDPC codes.

Regular and Irregular codes

Shown below is an example of a low-density parity-check matrix. We define this matrix as having dimension $n\ x\ m$, where n is the block length and m is the column length. Thus we see this is a (12,9) code. We also define k as the number of information bits per codeword. We see $k = n - m$. In the example below we see that $k = 3$. Considering $n = 12$, we see this code is rate $\frac{1}{4}$. All LDPC codes will be referred to as $(n, k)LDPC$ codes.

$$H = \left[\begin{array}{cccccccccccc} 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right] \quad Wr = 4$$

Wc = 3

Regular(3,4) LDPC(3,12) Parity Check Matrix. This code has rate 1/4

We now define two parameters of the code which refer to its sparseness. We define w_r as the number of ones in each row and w_c as the number of ones in each column. For an LDPC code to be defined as sparse it must have the following properties:

$$w_r \ll n$$

$$w_c \ll m$$

In order to meet these requirements we see the parity check matrix should be very large.

We refer to an LDPC code as *regular* if w_c is constant for every column and w_r is constant for every row. We see that for a regular code, $w_r = w_c \times (n \div m)$. We see in the example above that $(w_{c,} w_r) = (3{,}4)$.

If w_c or w_r is not constant then we say the code is *irregular.* All codes simulated in this project are regular codes. However, it is interesting to note that the best simulations, those that perform within 0.04dB of the Shannon limit, have been irregular codes.

Construction

There exist various methods of LDPC code construction [10]. Because LDPC codes are defined by their parity check matrices, this is what will generally be constructed first. The generator matrix can then be found afterwards using matrix manipulation. To save complications of finding the generator matrix for every code I create I assume transmission of the all-zero-codeword. This also saves enormously on encoding when large quantities of data are being simulated.

Gallager himself introduced a method of constructing suitable LDPC codes [3], however the method I used for generating LDPC matrices is based on one proposed by Mackay and Neal [11]. The parity check matrix is generated pseudo-randomly according to constraints.

The constraints are that each column and each row must have constant degree. All LDPC codes generated for my simulations will have columns of degree 3 and rows of degree 6. This means all rows will have 3 ones and all columns will have 6 ones.

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & ? \\ 1 & 0 & 1 & 1 & 1 & ? \\ 1 & 1 & 0 & 1 & 1 & ? \end{bmatrix}$$

Figure 2.1

The construction algorithm randomly generates columns of constant degree, one by one, and places them in the H matrix. Considering the matrix on the left, we see how the algorithm can fail to generate a matrix of constant row degree. The algorithm cannot complete the last column while keeping a constant row degree.

The algorithm attempts to generate a matrix of column degree 2 and row degree 4. Initially I choose to have the algorithm restart in situations such as figure 2.1, however for block lengths of several hundred bits the algorithm had to restart a large number of times. My solution was to simply allow a slight tolerance on the row degree distribution. In situations such as figure 2.1, the algorithm will add ones to particular rows, starting from the row with the least number of ones in it. In the example, this would cause a 1 to be added to the first row in the last column and then another 1 randomly to either of the two remaining rows. For very large block lengths having one or two rows of incorrect degree will not affect performance.

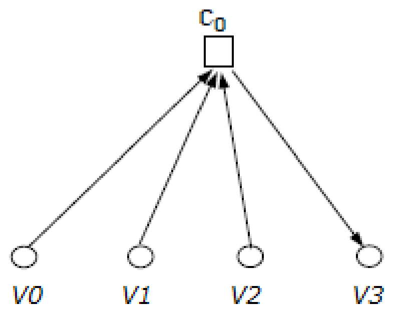
Decoding

There are two primary methods of decoding LDPC codes, Hard-decision and Soft-decision methods. The overall class of decoding algorithms used in decoding LDPC codes are collectively called message-passing algorithms. The reason for this is that the decoding process can be understood as the passing of information, or 'messages', along the edges of the codes Tanner Graph.

Each node in the Tanner Graph works in isolation, computing response messages based only on the messages received on the edges connected to it. It is this isolation of each node that allows for a parallel system of decoding. We can assign a processor to each node and compute messages for every node at once. Hard-decision decoding is a method in which binary decisions are made about each message to be passed. Soft-decision is a method which uses probabilities as messages. We first discuss hard-decision (HD) decoding.

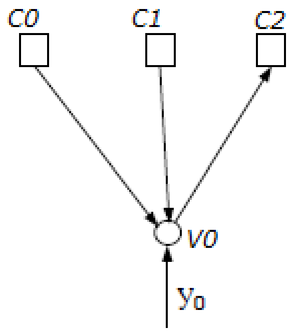
The decoding process works iteratively. This means all variable nodes and all check nodes take turns in passing messages back and forth to each other. We can look at an iteration decoding process in two steps:

Step 1: The check nodes, which represent parity check equations, determine what each node should



be for its parity check equation to be satisfied. The check node decides this using the information it receives. Considering the diagram, we see check node c_0 decides what it believes v_3 to be based on messages from the other 3 bit nodes. If v_0, v_1 and v_2 were 0, 0, 1 respectively, then to keep even parity the check equation would decide v_3 should be a 1.

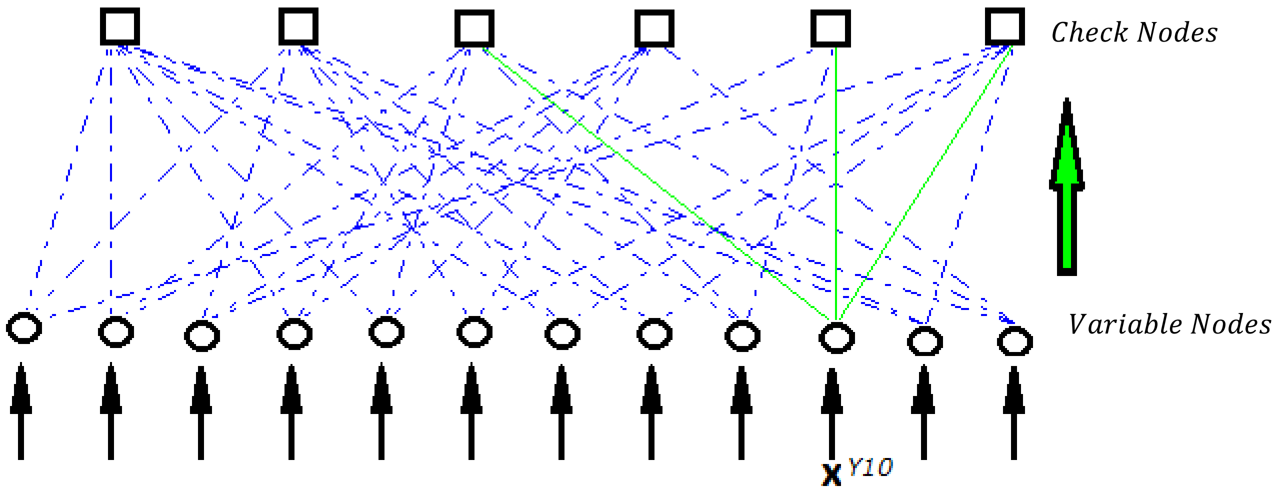
Step 2: The variable nodes receive messages from each check node indicating what they believe each



bit to be. Observing the figure, we see that the message sent to c_2 is computed as a majority vote from the other 3 sources of information – the channel value y_0 along with c_0 and c_1. If y_0 and c_0 indicated a 1 and c_1 indicated a 0, then the message to c_2 would be a 1 by majority vote.

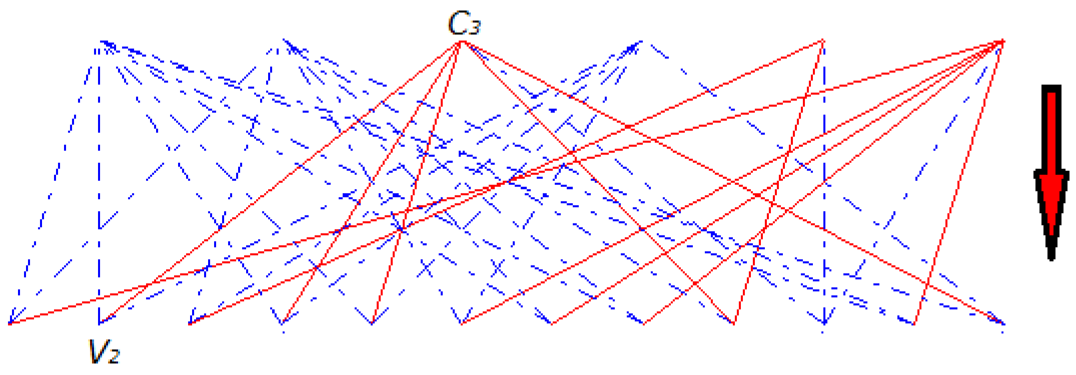
An example decoding process, utilising the bit-flipping algorithm for the Hard-Decision decoder is illustrated below. Convention is as follows: green or red edges indicate incorrect messages sent in a particular direction. The respective directions are labelled. A dashed blue edge indicates a correct message.

Iteration 1



For ease of understanding we consider a codeword which is a string of all zeroes. Thus we consider a binary one indicates an error.

In the first step of the decoding process, each variable (V) node starts off with a binary value – a bit received from the channel. In this example, we see the 10th variable node is in error as it has received an incorrect bit from the channel. On the first iteration all nodes propagate their bit belief to their respective check nodes. Thus we see three incorrect messages emanating from the tenth bit node.

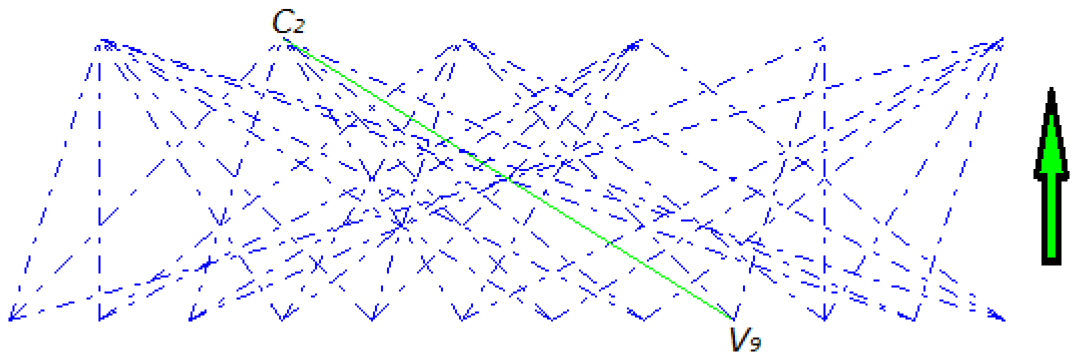


For brevity, only edges are shown in the Tanner Graph. The check nodes compute messages in response, and pass these messages 'down' the graph, as shown above. These response messages indicate what the check node believes to be the correct bit for the node it is messaging. These messages are calculated using the parity check equation associated with each node. Each individual message to a node is calculated on the basis that the other nodes are correct.

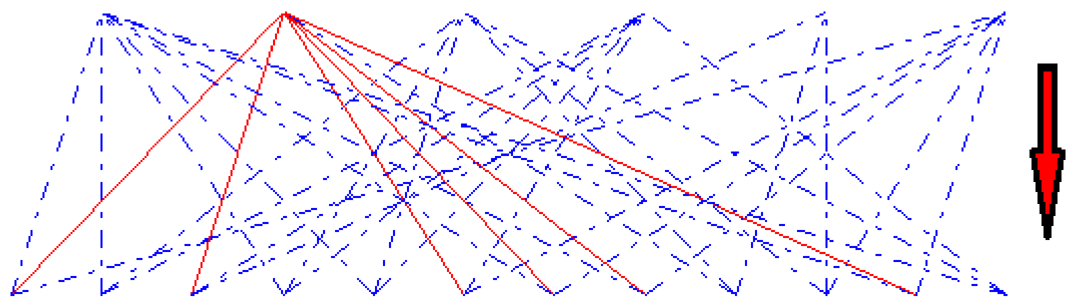
Considering node check node C_3, we see that it receives 5 messages indicating a zero and one message indicating a one. Thus when calculating its message to node V_2 it decides that if the other 5 bits are correct, that in order to have even parity then node V_2 must be a 1.

If we observe the variable nodes at the end of the first iteration, we see they received a mix of messages indicating what bit they should be. We now use a majority vote to decide on the response messages. We have four sources of information to make these decisions, the 3 messages from the check nodes and the original bit received from the channel.

Iteration 2



We see that only one node, V_9 received enough conflicting messages for a majority vote in the incorrect direction. Node V_9 indicates it believes C_2 to be a 1.



We see the incorrect message from node V_9 has 'corrupted' node C_2.

Observing the check messages at the end of the second iteration, we see now that all incorrect messages will be cleared up. No bit node is receiving more than one conflicting message, thus no incorrect messages will be computed on the next iteration. The decoder has converged on a codeword and will exit the decoding process.

The above example illustrates the correction of one bit error for a very short block length. As stated above, LDPC codes have good performance only for very long block length. This is illustrated in figure 2.2 below:

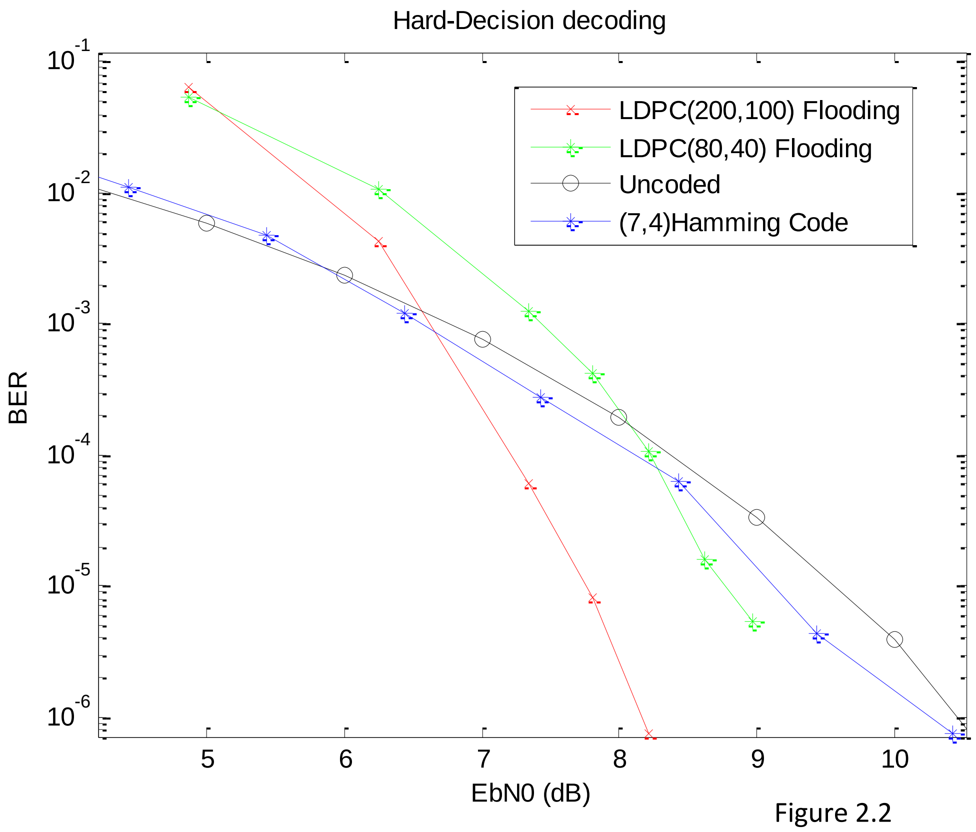


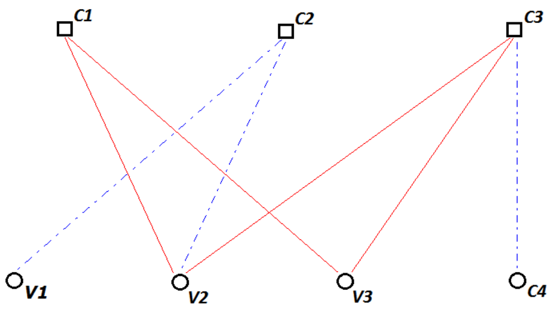
Figure 2.2

Shown in the graph is the BER for two different block length codes. While the length 80 code has mediocre performance, we see a large improvement for the longer block length 200 code. The (7,4) hamming code is included for reference. We see here a typical characteristic of good LDPC codes is a 'waterfall' effect where the BER drops quite rapidly beyond a certain dB. The above simulations are carried out for a maximum number of 12 iterations of the decoding process.

Graphical Structures.

Using the Tanner Graph as a graphical model for analysing LDPC codes allows us a very useful method of analysing the codes. We see from the graphical model that there arise various structures which affect decoding performance [12]. Primarily we aim to avoid short length cycles and stopping sets within the graphical model.

An edge on the Tanner Graph is how we pass information between nodes about the level of belief we have about each bit. A cycle is a set of edges which connect back to each other in a loop. For example we can see a 4-cycle shown below, and an example as a H matrix representation.

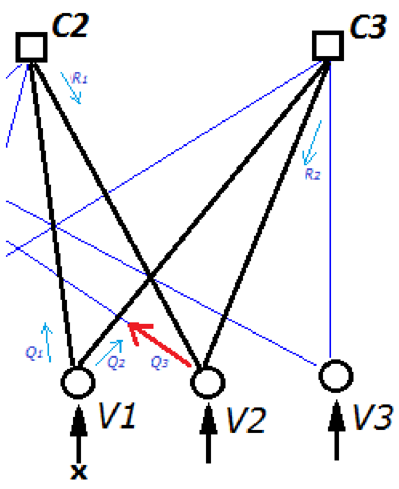


$$\begin{bmatrix} : & 1 & : & 1 & : & : \\ : & : & : & : & : & : \\ : & 1 & : & 1 & : & : \end{bmatrix}$$

Fig: Example 4 cycle in a Tanner Graph

Fig: Example 4 cycle in a H matrix.

We see that in a cycle, a node passes information in a loop back to itself, thus in general we aim to have long cycles. An example problem presented by a 4-cycle during Hard-decision decoding is demonstrated below:

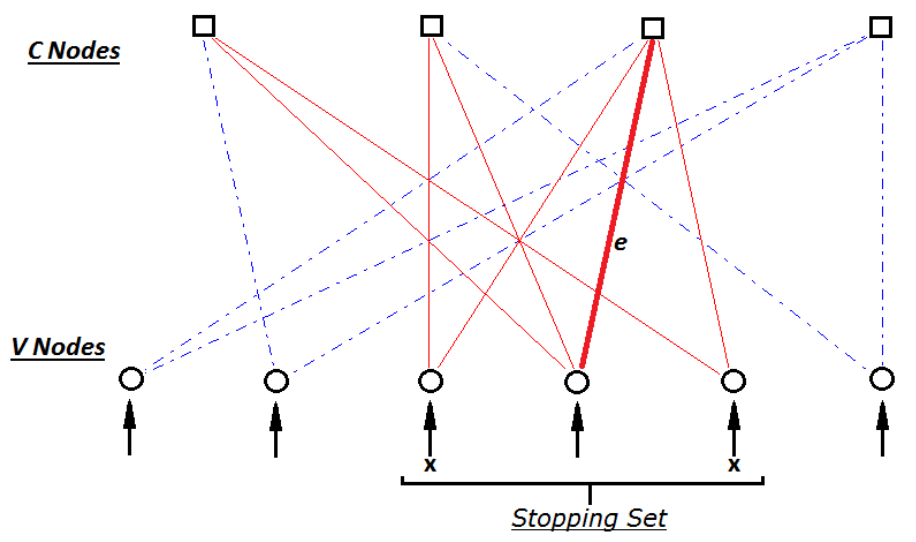


We see a 4-cycle in the structure, as highlighted in bold. Consider if node V_1receives an incorrect value from the channel. At the beginning of the first iteration all nodes propagate their belief to all check nodes. We see this incorrect node causes messages R_1 and R_2 to be incorrect (along with many others). In this case we see both R_1 and R_2 go to the same variable node V_2 because of the 4-cycle. This leads to a majority vote over-ruling the previously correct node V_2. We get the incorrect message Q_3 as a result.

A stopping set is a structure which can cause the decoder to fail. It can be defined as follows:

A stopping set S is a subset of V - the set of all variable nodes - such that all neighbours of the variable nodes in S are connected to S at least twice. [13]

Informally we can say that a stopping set is a set of variable nodes that is not well connected to the rest of the graph, in such a way that causes trouble to the decoder. We see an example of a stopping set below:



For an example of why this stopping set can cause the decoder to fail, we consider the scenario where the two variable nodes marked with an x begin with incorrect channel values. We see at the end of the first iteration this causes the one correct node in the set to receive a majority of incorrect messages, thus it propagates incorrect messages. This feeds back into the already incorrect nodes to continue the cycle. The decoder will not resolve these two errors. Consider also if all 3 nodes in the stopping set are incorrect, only one of the connected check nodes is unsatisfied. Thus we define this as a (3,1) stopping set.

We note that the neighbors of the 3 variable nodes in the stopping set shown are all connected to the stopping set at least twice. In this example we see that the stopping set comes from two 4-cycles which share a common edge. This edge is marked e in the figure. We see this is a common reason for small stopping sets occurring and is a strong reason to avoid 4-cycles.

Soft Decision

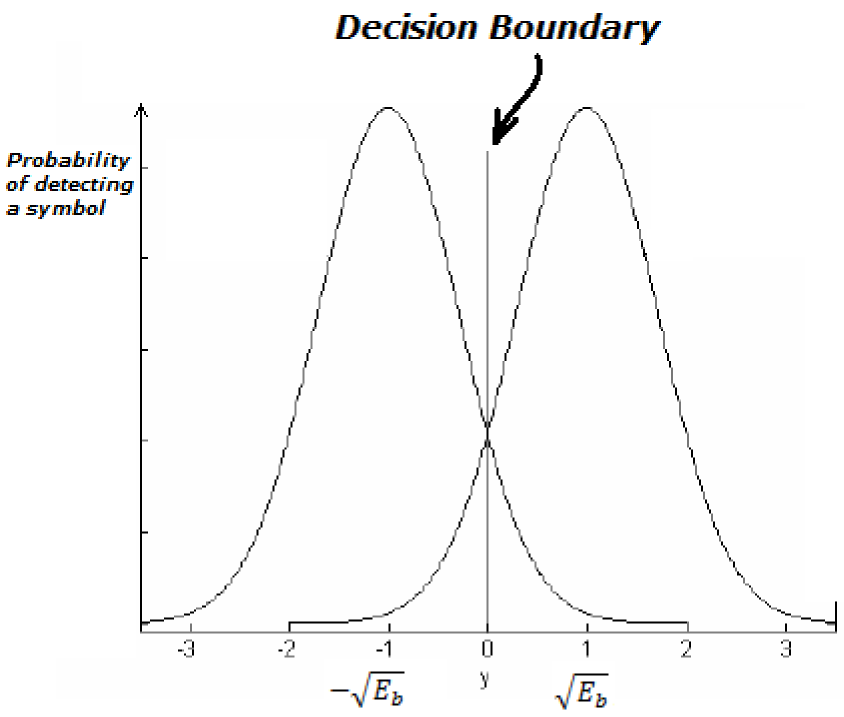
The above discussion has focused primarily on hard-decision decoding methods. Hard-decision decoding provides a relatively simple method of decoding LDPC codes however it has significantly lower performance than soft-decision methods. Soft-decision decoding deals with probabilities during the decoding process instead of making binary decisions. We see that hard-decision methods discard a lot of useful information in making these binary decisions.

BPSK

In order to transmit a signal, we must map the input binary signals to an analog signal. For this purpose we use a Binary Phase Shift Keying modulator (BPSK). We use the following bit to symbol mapping in simulation:

$$\{1,0\} \quad \longrightarrow \quad \{\sqrt{E_b}, -\sqrt{E_b}\}$$

The figure below illustrates the likelihood of a symbol being correctly interpreted, after being transmitted over an AWGN with $E_b = 1$.



White noise is added to the BPSK modulated signal in the communication channel. This corrupts the signal – if a sufficient amount of noise is added, the symbol may cross the decision boundary.

We find that any particular received symbol can lie anywhere on the distribution shown above. Any transmitted symbol which picks up enough noise that it crosses the decision boundary at zero will be incorrectly interpreted by the de-modulator.

We gain an amount of information about the likelihood of a transmitted symbol based on where it lies on the distribution when we receive it. We see that if a received symbol is, for example -3, then we can say with great certainty that the transmitted symbol was a -1. However if the received symbol was -0.01, then the decision is less clear. We can calculate the probability [14] of a transmitted symbol as follows:

$$P_i = \frac{1}{1 + e^{-\frac{2rt}{\frac{N0}{2}}}}$$

where r is our received symbol. P_i is the probability that t was transmitted. Thus we have:

$$Pr[t_i = 1|r] = \frac{1}{1 + e^{-\frac{2r(1)}{\frac{N0}{2}}}}$$

This 'channel probability' presents us a significant amount of additional information about our received vector, as opposed to our hard-decision decoder which makes binary decisions on all received symbols.

Decoding

The soft-decision decoder operates on the same principle as with the hard-decision decoder, in that it iteratively computes the distributions of variables in a graphical model. The difference is that the messages are computed as probabilities that the received bit is a 1 or a 0 given the received vector y. Gallager, in his seminal thesis [3], provided a decoding algorithm for LDPC codes which is generally near optimal. This algorithm has been independently re-discovered by several researchers, along with several related algorithms. The algorithm is referred to by a variety of names, depending on the context, such as the belief propagation algorithm (BPA) and the message passing algorithm (MPA). I will refer to it as the BPA.

It's necessary to start with introducing some notation based on the figure 3.1:

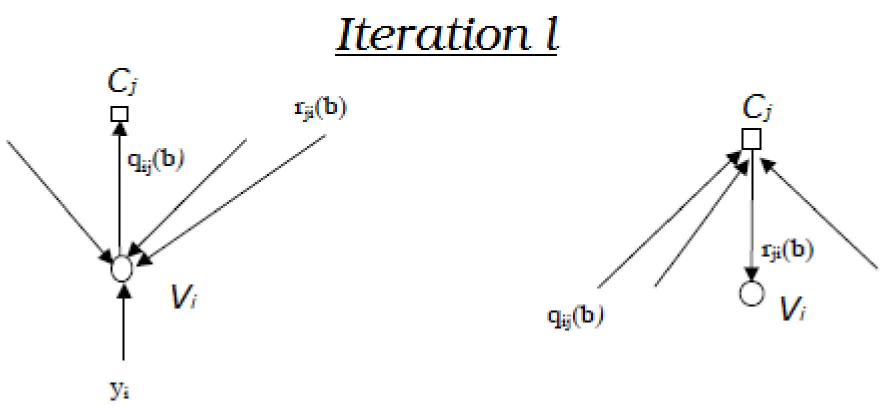


Figure 3.1: The two steps of a single decoding iteration

$P_i = \Pr[v_i = 1 \vert y]$	The conditional probability that v_i is a 1 given the value of y. We note therefore that $1 - P_i = \Pr[v_i = 0 \vert y]$
q_{ij}^l	The message sent by message node v_i to check node c_j at round l.
$q_{ij}^l(1)$	The amount of belief that y_i is 1. Note that $q_{ij}^l(1) = \mathrm{P_i}$
$q_{ij}^l(0)$	The amount of belief that y_i is 0. Note that $q_{ij}^l(1) + q_{ij}^l(0) = 1$
$q_{i'j}^l$	Set of incoming messages excluding the one received from node v_i
r_{ji}^l	The message sent by check node c_j to message node v_i at round l.
$r_{ij}^l(1),\ r_{ij}^l(0),\ r_{i'j}^l$	Notation as with q
q_i	Probability of a 1 at variable node v_i

There are two primary types of calculation which must be performed during Soft-decision decoding: the messages calculated by the check nodes and the messages calculated by the variable nodes.

We can derive an equation for calculating the former by first considering the probability that there is an even number of 1's on a set of 2 variable nodes. We see this happens if there are either two ones or two zeroes. Thus we can say:

$$\Pr[v_1 \oplus v_2 = 0] = q_1 * q_2 + (1 - q_1)(1 - q_2)$$

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

Now we consider the probability of an even number of ones at 3 message nodes:

$$\Pr[(v_1 \oplus v_2) \oplus v_3 = 0] \ = \frac{1}{2}[\,1 + (1 - 2q_1)(1 - 2q_2)(1 - 2q_3)]$$

We see that in general:

$$\Pr[v_1 \oplus \ldots \oplus v_n = 0] = \frac{1}{2} + \frac{1}{2}\prod_{i=1}^{n}(1 - 2q_i)$$

We must only use extrinsic information when computing a message, thus we do not use the message received from v_i when computing r_{ji}. Now we can find the message that c_j sends to v_i at round l as:

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

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

where V_j is the set of all variable nodes connected to check node c_j.

The message q_{ij} that node v_i passes to node c_j is simply the product of the incoming messages $r_{j'i}$ and the channel probability P_i. We multiply this by a scaling factor k_{ij} which is chosen so that:

$$q_{ij}^{l}(1) + \ q_{ij}^{l}(0) = 1$$

Thus:

$$q_{ij}^{l}(0) = k_{ij}(1 - P_i) \prod_{j' \epsilon C_{i \neq j}} r_{j'i}^{l-1}(0)$$

$$q_{ij}^{l}(1) = k_{ij} P_i \prod_{j' \epsilon C_{i \neq j}} r_{j'i}^{l-1}(1)$$

We also define Q_i^l as the effective probability of 0 and 1 at variable node v_i at round l.

During each iteration with calculate the effective probability of each variable node being a 0 or a 1. We terminate the algorithm if these estimates satisfy the parity-check equations. We say we have 'converged' to a codeword. We calculate these as follows:

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

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

Simulation

Using MATLAB I implemented a simulation of an LDPC soft-decision decoder. Using this I simulated the BER performance of two separate LDPC codes of different block length, allowing a maximum of 25 iterations. The results are shown in figure 3.2.

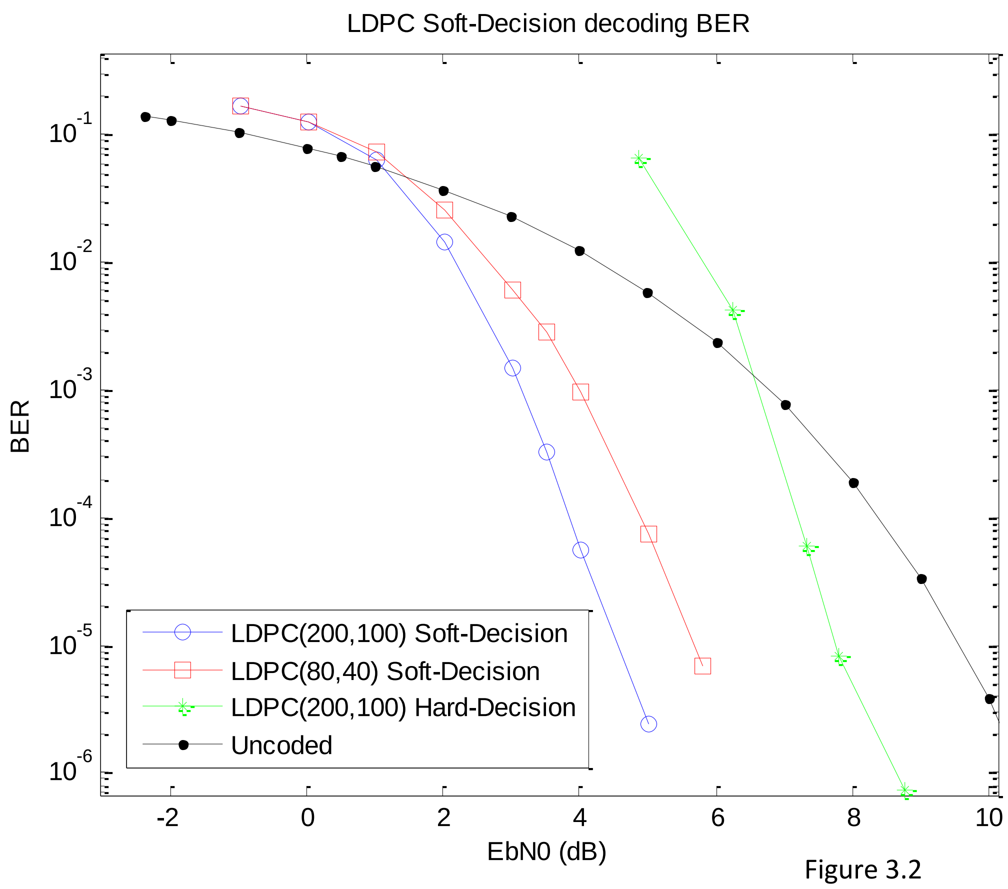


Figure 3.2

Included for reference is a hard-decision simulation of a block length 200 code. We see from the results, the soft-decision decoder greatly outperforms the hard-decision decoder, even for shorter block lengths.

Message Scheduling

LDPC codes are developing more and more as an industry standard. Because of this there is a surge in interest in developing low-complexity decoding algorithms which allow more optimised low-cost high throughput decoder implementations. The traditional flooding schedule has long been shown to offer less than optimal convergence behaviour [15] compared to methods which implement an ordered process for updating the nodes.

Schedules can be roughly broken down into two classes, adaptive and non-adaptive. A non-adaptive schedule is one whereby the decoding schedule is known entirely before decoding begins, this has the advantage of having easier hardware implementation. An adaptive schedule is one whereby the message updating order is changed based on graph structure or specific conditions which are met during decoding. This obviously allows for greater possibilities.

Using some intuition we can begin to understand why the flooding schedule may perform in a less than ideal manner.

Consider a bit node which has not yet converged. If we consider that all other nodes have converged, then we see there is no need in passing all possible messages between all nodes. We need a limited number of messages in order to provide the node with enough information. We see then that there is an unneeded complexity in ‘flooding’ the Tanner graph.

4-cycle avoidance schedule

As discussed earlier, in constructing LDPC codes we aim to avoid short cycles on the Tanner Graph. In general we can expect there to be a decline in performance, especially for larger numbers of these cycles.

$$\begin{tikzcd}
 & 0 & 1 \arrow[l, bend right] \\
1 & 1 \arrow[r] & 0 \\
1 & 1 &
\end{tikzcd}$$

Figure 4.1

It makes sense then to attempt to use scheduling in some manner that might minimise the effect of these short cycles. This was a primary focus of my initial attempts at scheduling. The first step in this procedure was to first be able to detect and index these short cycles. It was also necessary to manipulate the number of 4-cycles in a matrix so I could test the performance effects of various numbers of 4-cycles. The basic premise of how I carried this out is demonstrated in figure 4.1.

To remove a 4-cycle, we can carry out a basic substitution as shown. We simply substitute a 1 in the cycle for a 0 at a different place in the row. In order to keep the regular degree distribution, we must substitute a 1 back into the column on a different row, as shown.

With the occurrence of these cycles indexed in a particular matrix, it was now possible to adapt an updating schedule to avoid these constructs. The idea was basically to update these nodes with less regularity than the other nodes.

Let S be a subset of variable nodes such that every 4-cycle has one of its variable nodes in S. The update rule dictated that nodes S are updated with very low probability for the first 5 iterations.

Results of simulations are given below:

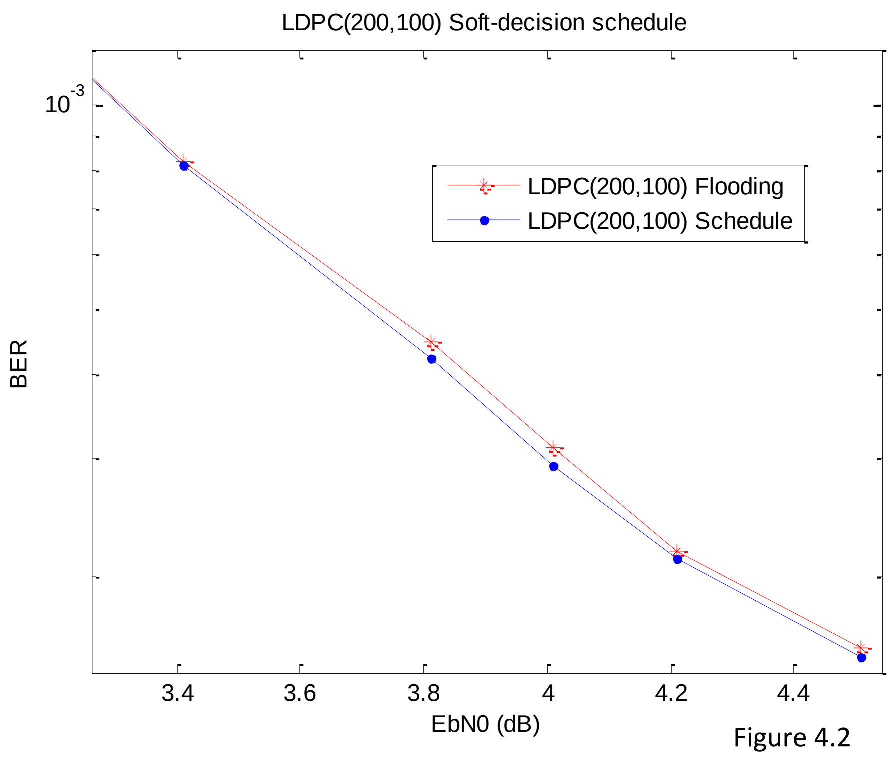


Figure 4.2

Simulations were carried out using a block length 200 code with forty 4-cycles and a maximum of 25 iterations. As we can see the schedule provides a negligible improvement in BER. We see how the schedule affected the average iterations required for convergence:

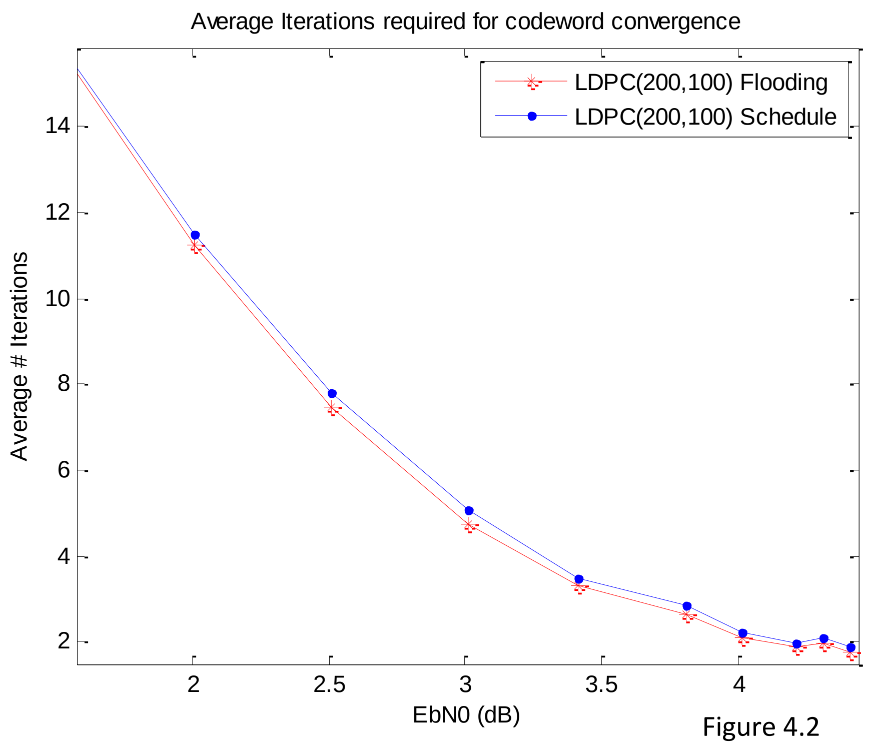


Figure 4.2

We see from this graph that the slight improvement in BER has come at a proportionately minor increase in decoding complexity.

We aim to improve convergence behaviour, however in this case we merely have a small trade-off between BER and complexity, with no net gain in performance. We see from this schedule that simply ignoring nodes which we deem to be problematic is not an effective methodology. Instead, we aim to utilise scheduling to maximise the efficiency of information propagation in the graph, so as to maximise the benefits of individual computations.

Serial Schedule

A simple alternative to the flooding schedule is to update all nodes one by one. This is known as a serial schedule [16]. Serial schedules come in two basic types – those where the check nodes or variable nodes are updated serially. A Hybrid can utilise a combination of both of these. The following discussion and experimentation is on variable node scheduling, which I shall refer to as the serial-V schedule.

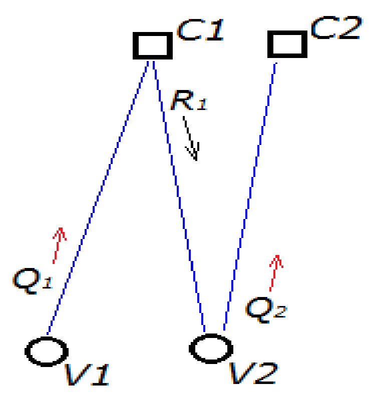


Figure 5.1

The benefit of a serial schedule is that it allows immediate propagation of messages. Consider figure 5.1. In the flooding schedule, Q_1 and Q_2 would be computed in parallel and sent simultaneously. In the serial schedule, we send Q_1 first. Now we find that R_1 is available to node V_2, thus we have more information available when computing Q_2 and so the message is likely to be more reliable.

At first glance it might appear this method requires a greater number of computations from the check node side. This is not the case: R_1 will only be computed once per iteration. Individual messages from the check node need only be computed as they are required, we do not need to compute every outgoing message at each update.

The drawback of serial scheduling is that it does not allow for a parallelized decoder hardware implementation whereas the flooding schedule can have all nodes completely parallelized. If only one processor is utilised or available for decoding then we stand only to gain from serial scheduling - otherwise it is necessary to compare the benefits of reduced complexity with the drawbacks of losing parallel decoding. Alternatively, the serial schedule can be partially parallelised. We can divide the nodes into subsets of nodes which are updated simultaneously. It's possible to choose the subsets of nodes based on certain parameters which will optimise the process. For example, we may divide the nodes into subsets such that no two variable nodes in a subset are connected to any of the same check nodes. In fact, in this case we see there will be no difference in one by one decoding within these subsets and parallel decoding of each node in the subset.

In [16], experiments were carried out on the BEC using serial-V scheduling. It was shown that for the BEC, serial scheduling reduced decoding complexity significantly, reducing the average iterations needed for convergence by approximately half without degrading the BER. These simulations allowed a large number of iterations. It follows from this that if we limit the number of iterations we might expect a better BER performance from a serial schedule than for flooding since we have faster convergence. [16] has shown this also to be the case.

I decided to carry out my own simulations based on this concept. With this in mind it's worth investigating the average number of iterations required for convergence. Simulating for a regular LDPC code of block length 200, I produced the following:

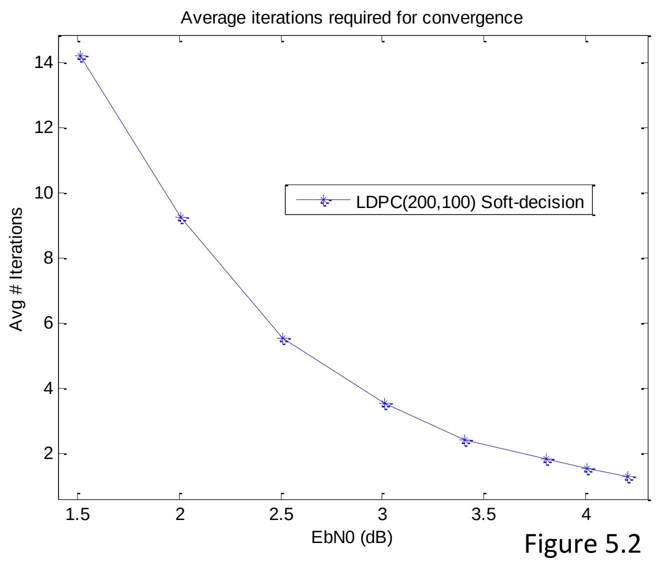


Figure 5.2

We see a sharp decline in the number of iterations required as the SNR improves.

To test how serial schedules perform with a limited number of iterations, it seems reasonable that for whatever our choice, it would make sense to simulate the particular SNR value for which this is the average number of iterations required. Then we can simulate SNR values above and below this point and see how the convergence behaviour is affected. Choosing a limit of 5 iterations, we see this corresponds to the average iterations for about 2.5dB.

In figure 5.3 we see the results from the serial-V schedule limited to 5 iterations. Soft-decision decoding is used.

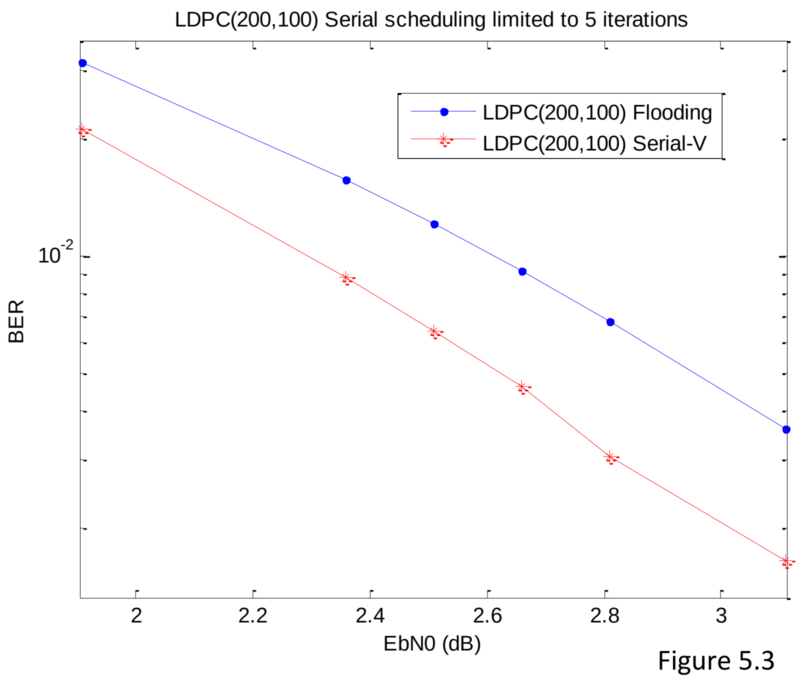


Figure 5.3

We see with the serial-V schedule we have a respectable coding gain of about 0.3dB for this BER range. This gain can more than likely be entirely attributed to the increased efficiency of the computations in the serial schedule.

In the serial-V schedule the nodes are updated one by one, however the order is arbitrary. It seems reasonable then to assume that we might improve behaviour by implementing some kind of ordered approach to serial updating. I took a number of approaches to this.

As stated above, two nodes which are not directly connected by one check node will not be affected by a choice of serial or parallel decoding. We see this clearly in figure 5.1 – if $V_1$was not linked to V_2 by C_1 then there would be no gain from serial decoding. I decided to adapt a schedule which orders the nodes in consideration of this. Instead of arbitrary ordering, the n^{th} node we update must be connected by checknode to the $(n-1)^{th}$ node. We see this in the figure below:

$$H = \begin{array}{cccccccccccc} V1 & V2 & V3 & V4 & V5 & V6 & & & & & & \\ \hline 1&0&0&0&0&1&0&1&0&1&0&0\\ 1&0&0&1&1&0&0&0&0&0&1&0\\ 0&1&0&0&1&0&1&0&1&0&0&0\\ 0&0&1&0&0&1&0&0&0&0&1&1\\ 0&0&1&0&0&0&1&1&0&0&0&1\\ 0&1&0&0&1&0&0&0&1&0&1&0\\ 1&0&0&1&0&0&1&0&0&1&0&0\\ 0&1&0&0&0&1&0&1&0&1&0&0\\ 0&0&1&1&0&0&0&0&1&0&0&1 \end{array}$$

$$H = \begin{array}{cccccccccccc} V1 & V2 & V3 & V4 & V5 & V6 & & & & & & \\ \hline \boxed{1}&0&\boxed{0}&\boxed{0}&0&\boxed{1}&0&1&0&1&0&0\\ \boxed{1}&0&\boxed{0}&\boxed{1}&1&\boxed{0}&0&0&0&0&1&0\\ \boxed{0}&1&\boxed{0}&\boxed{0}&1&\boxed{0}&1&0&1&0&0&0\\ \boxed{0}&0&\boxed{1}&\boxed{0}&0&\boxed{1}&0&0&0&0&1&1\\ \boxed{0}&0&\boxed{1}&\boxed{0}&0&\boxed{0}&1&1&0&0&0&1\\ \boxed{0}&1&\boxed{0}&\boxed{0}&1&\boxed{0}&0&0&1&0&1&0\\ \boxed{1}&0&\boxed{0}&\boxed{1}&0&\boxed{0}&1&0&0&1&0&0\\ \boxed{0}&1&\boxed{0}&\boxed{0}&0&\boxed{1}&0&1&0&1&0&0\\ \boxed{0}&0&\boxed{1}&\boxed{1}&0&\boxed{0}&0&0&1&0&0&1 \end{array}$$

On the left is an arbitrary schedule, the order is $V_1 - V_2 - V_3 - V_4$ ... etc.

On the right is an example of the proposed schedule. We update V_4 after V_1as these two variable nodes are involved in the same check equation. The order continues as shown. I will refer to this as the Serial-C schedule.

When we begin the decoding process the only information we have initially are the channel probabilities. Each bit in the codeword will be received as a likelihood of what bit it is. I decided to adapt the serial schedule to order the nodes based on these initial probabilities. There are two approaches to this; we can order the nodes starting with the perceived most reliable nodes, with the intention that they will spread reliable information around the Tanner Graph early. I will refer to this as the serial-A schedule. Alternatively we can start with the less reliable nodes, with the idea that they may be corrected earlier on, possible allowing faster convergence. I will refer to this as the serial-D schedule. The serial-A schedule seems more appealing in an ideal system because we allow more reliable belief to propagate first, however if we consider a situation where we are limited to a low number of iterations, then the serial-D schedule might become more appealing.

I simulated these three schedules under the same conditions as the serial-V schedule. Figure 5.4 shows the results.

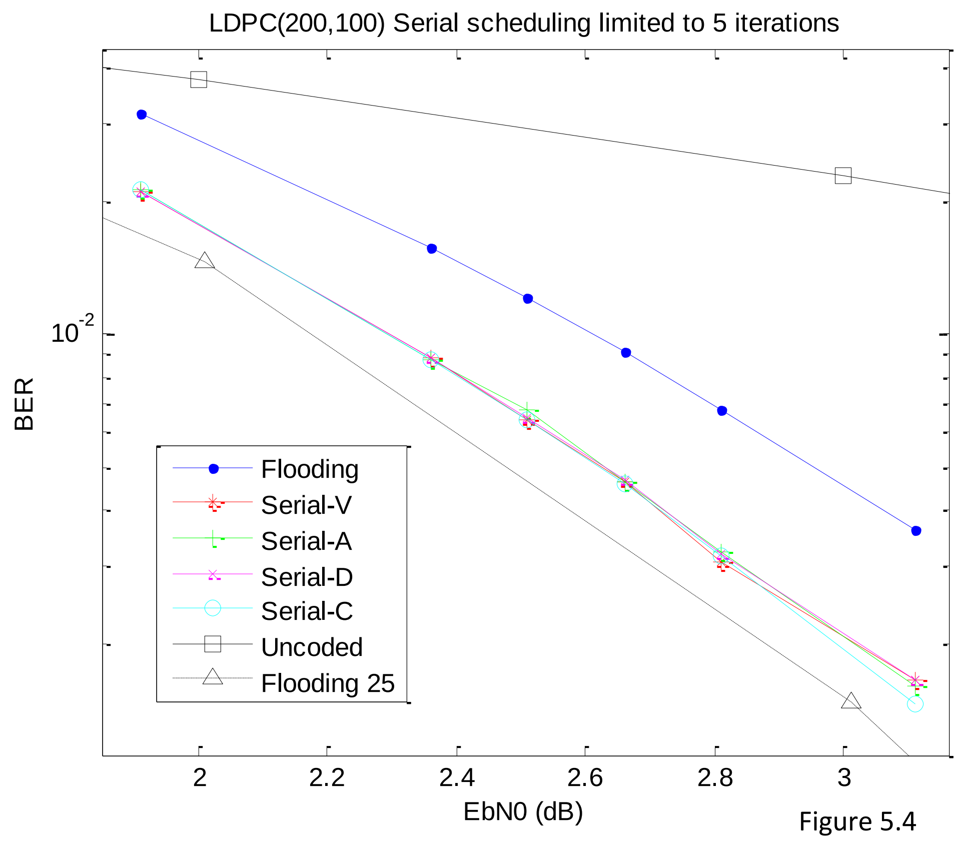


Figure 5.4

Included for reference is the same LDPC(200,100) code which is decoded with a maximum number of 25 iterations.

Comparing the schedules with respect to each other we see they have a negligible difference in BER. It seems the different serial updating orders matter significantly less than simply the decision to serially update or not. In terms of BER the flooding schedule performs significantly worse than any arbitrary serial schedule.

This point is reinforced by noting that the serial schedules come much closer in performance to the flooding schedule with 25 iterations than they do to the flooding schedule with the same number of iterations.

In figure 5.4 we compare the complexity requirements of each serial schedule.

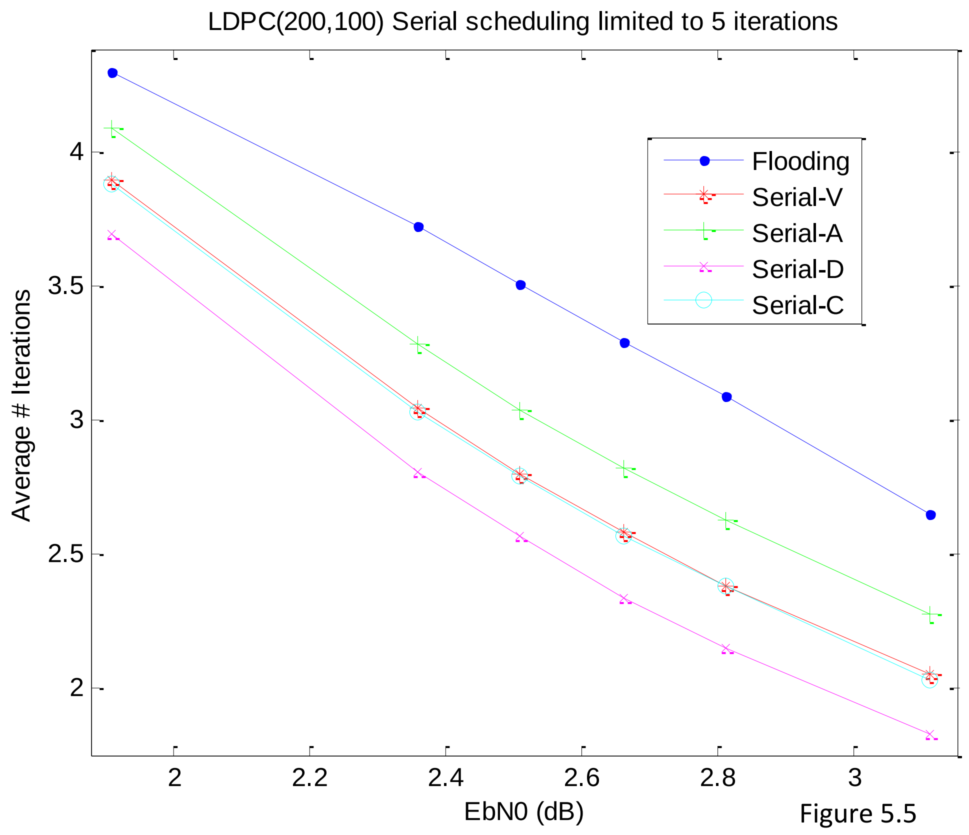


Figure 5.5

We see again the flooding schedule performs poorly, requiring a significantly greater number of average iterations despite giving a worse BER.

Comparing the arbitrary serial-V schedule with the serial-C schedule we see the serial-C schedule consistently outperforms the serial-V schedule, but only by an extremely narrow margin, so much so that it is completely negligible. It seems that choosing to sequentially update nodes connected by the same check node does not affect performance as expected. We might conclude that in an arbitrary schedule the updating order will rarely update more than a small subset of nodes without updating neighbouring ones, unlike the flooding schedule which updates all 200 nodes simultaneously.

We see much more interesting results comparing the serial-A schedule to the serial-D schedule. The serial-A schedule begins by updating more reliable nodes first. However when we are limited by a small number of iterations we see this as a bad choice. The serial-D schedule starts off updating the less reliable nodes. Since we are limited by our iterations it seems more efficient to start with these less reliable nodes. The serial-D schedule utilises an average 0.5 iterations less than the serial-A schedule for all tested SNR values.

Scheduling based on reliability

As we have seen, the flooding schedule has been shown to have a convergence behaviour which is off lower performance than alternative scheduling techniques.

The serial schemes discussed so far have had non-adaptive updating schedules. The entire schedule is known beforehand. This allows for easier hardware implementation. However by allowing an adaptive schedule we open up more possibilities.

The authors of [17] discussed what is known as a ‘lazy’ schedule. This is an example of an adaptive schedule. The updating rule behind this is based on the fact that the decoding process generally achieves a strong level of confidence of most of the variable nodes after a small number of iterations. The lazy schedule is designed to exploit this phenomenon.

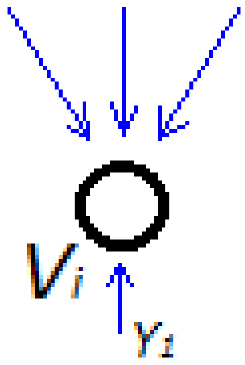
In lazy scheduling, only a subset of nodes is updated each iteration. The decision to update a node is carried out as a function of the perceived reliability of that particular node. This reliability measure is a function of Q_i^l, the effective probability of a node during iteration l. By choosing to update only those nodes which are more likely to be misperceived in a particular iteration, it was shown that a significant reduction if decoding complexity was achievable without degrading the BER.

Below I discuss a proposed adaptive schedule for hard-decision decoding. The updating rule for this schedule chooses to selectively update nodes based on a perceived level of confidence of the node. The updating rule is as follows:

During iteration l, node V_i will not update if, on the previous iteration, all incoming messages agreed with the channel value of V_i.

I will refer to this schedule as the reliability schedule(RS). The effect of this schedule can be understood as follows. If a node receives messages which are all in agreement with the channel, then we consider this node to be correct with a high probability. By removing it from the schedule we make it harder for this node to be corrupted. It is important to note that at no stage do we completely remove a node from the schedule, if a node receives conflicting messages on iteration n, we will begin updating it again on iteration $(n+1)$. We see for a node to be overruled, it must receive corrupting messages on two consecutive iterations.

This is best demonstrated with an example. Consider the following node V_i during an arbitrary iteration.



Iteration l

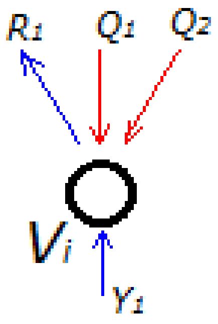
At this point in the decoding process, we see that all incoming messages to the bit node are in agreement with the channel value. Thus, in the case of RS, this node will not be updated next iteration. The logic behind this is that we consider the complete agreement of all messages to be an indication of a high reliability measure.

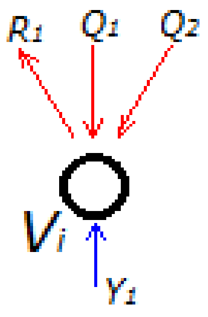
Of course it's possible, with a much smaller probability, that this node and all of its incoming messages might be incorrect. In this case it would be disadvantageous to not update the node. This is something which I will discuss later on.

Iteration l+1

Schedule *Flooding*

In this iteration, we see our node V_i receives two conflicting messages. In the flooding schedule we see this presents a problem. In computing the message R_1, the two incoming messages Q_1 and Q_2 create a





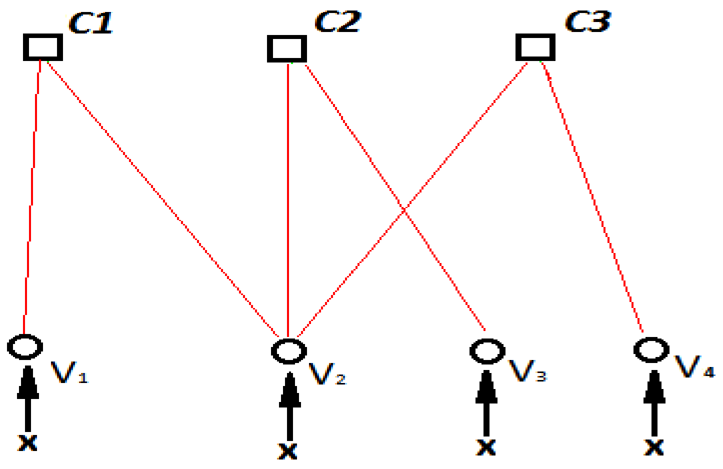
majority vote which is incorrect. Thus R_1 is incorrect.

However, with RS we see this does not occur. The node is not updated this iteration due to its perceived reliability in the previous tieration and so its outgoing messages are the same as in the previous iteration.

This case assumes a correct variable node; however let us consider the case of an incorrect bit node. There is an inherent danger in any schedule in not updating an incorrect variable node with information it might need in order to make a correct decision about the code bit. If we consider intuitively the likelihoods of incorrect variable nodes being removed from the schedule then we see that this is an unlikely problem in this scenario.

Consider first a correct node as discussed above. This node will likely receive mostly correct messages and will regularly receive all correct messages. Thus it will be removed from the schedule since all messages will agree with the channel.

Obviously we do not know beforehand which nodes are correct and incorrect, but we easily see that an incorrect node is unlikely to be removed from the schedule. An incorrect node must receive all incorrect messages in order to be left out of the updating schedule. In order for this to happen, all parity check equations for the node must receive an odd number of additional errors from other bit nodes. Consider node V_2 below, where a red line indicates an incorrect belief and a blue line indicates a correct belief:

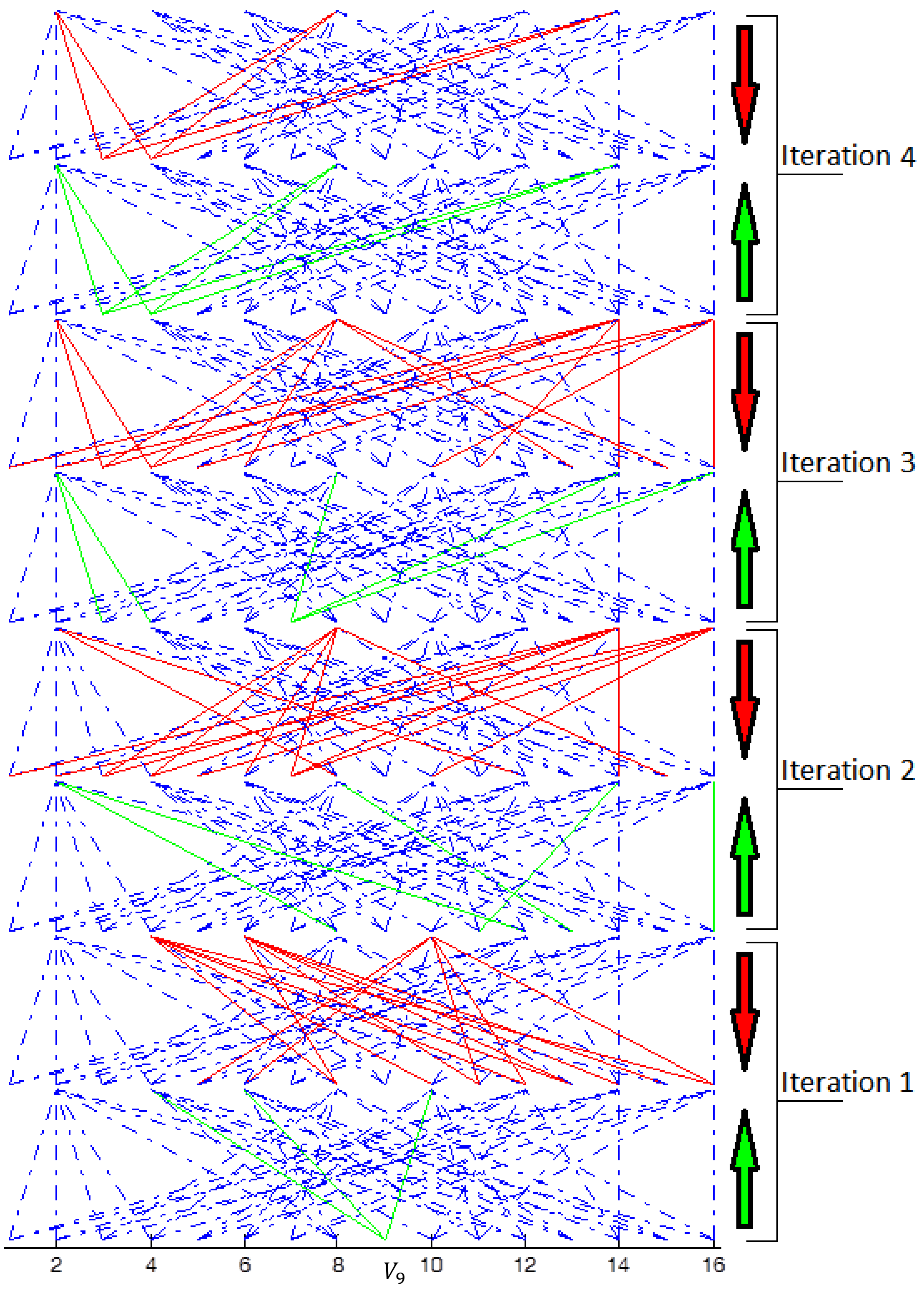


In this example, we see that all check nodes connected to V_2 indicate an incorrect belief to V_2 and that this belief agrees with the channel. Under this condition we see that V_2 will be removed from the updating schedule. We see this scenario is an unlikely occurrence.

We see from this example the strong significance in not completely removing nodes from the schedule. Without this consideration, the rare instances of the above scenario would lead to problems. In the above example, any or all of V_1, V_3 and V_4 are likely to subsequently propagate a more reliable message after gaining more reliable information from the rest of the graph, and thus V_2 will receive a different message. Thereby it will be reintroduced in the updating schedule.

A direct comparison of an LDPC decoding process is presented below which directly demonstrates a case of improved performance. First we consider the scenario in the flooding schedule. The codeword is the all zero codeword; this way we see all blue 'correct' edges are zeroes and all coloured 'incorrect' edges are ones.

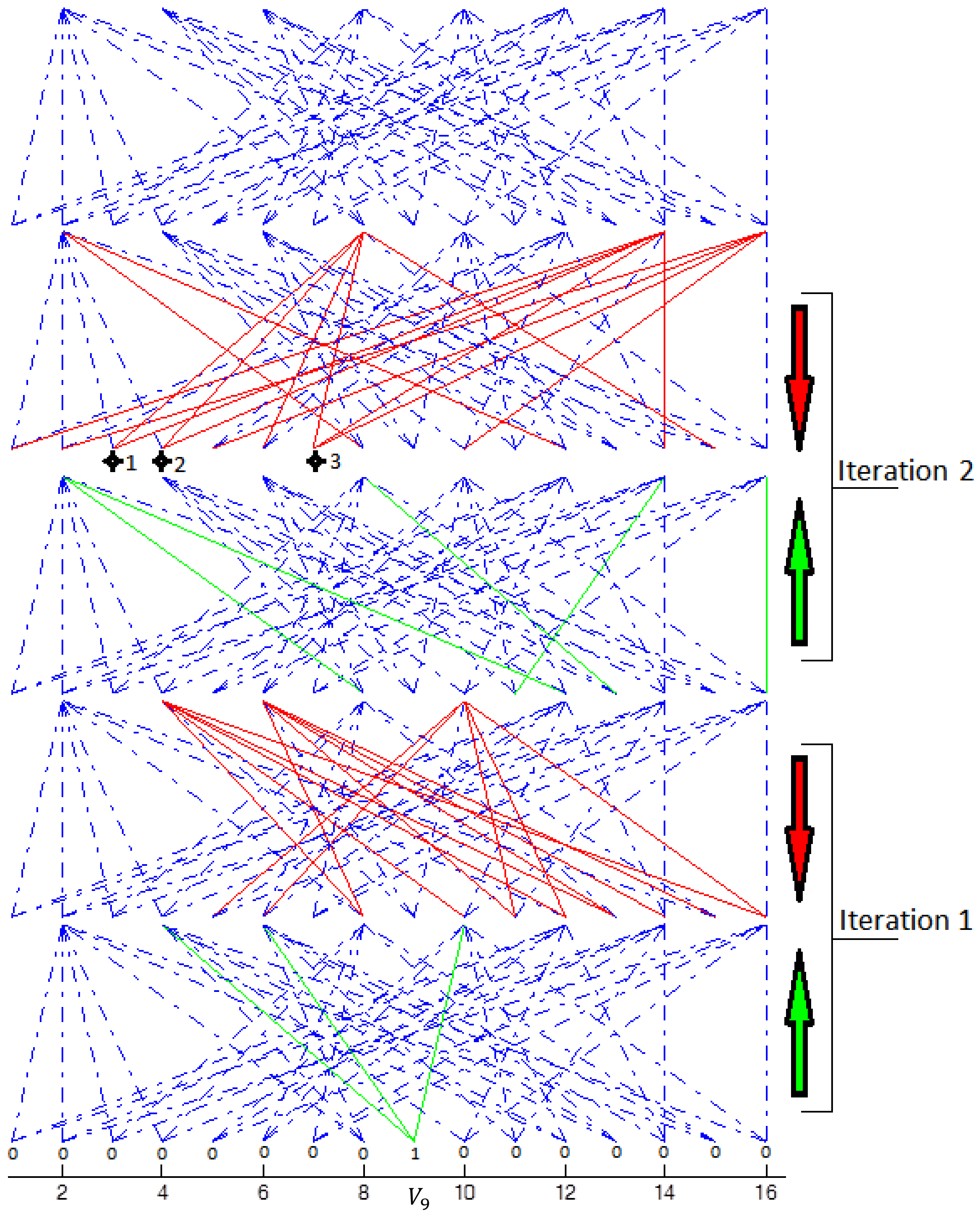
For brevity, the nodes are not all labelled. Convention is the same as earlier, the top row of nodes correspond to check nodes whereas the bottom row correspond to variable nodes. Iterations are labelled, proceeding upwards from the bottom of the page. Both steps in each iteration are shown.



Iteration 1: We see we have 1 bit error in our codeword. Variable node 9 is in error. Thus it propagates incorrect belief messages.

Iteration 4: By the fourth iteration we see the decoding process has failed to converge to a codeword. The decoding process is stuck in a stopping set consisting of nodes V_3 and V_4. As discussed earlier, a stopping set can cause the decoder to fail. The decoding process is unable to resolve this and the result is two bit errors.

Now we consider the same example, implementing RS:



Iteration 1: Node 9 is incorrect. Thus it propagates incorrect messages. At the end of the iteration, all nodes in agreement with the channel are removed from the updating schedule. Thus nodes 1, 2, 3, 4 and 7 are removed. Note that node 9 is not removed since its messages do not agree with the channel.

Iteration 2: At the end of the second iteration, we see the benefits of the schedule come into effect. Observing the 3 nodes marked on the graph in the second iteration, we see 3 nodes which previously were confident in their agreement, now being corrupted. However these nodes have been removed from the schedule and so are not updated on this iteration, thus we see the incorrect messages are 'cleaned up' from the Tanner Graph. Comparing with the flooding schedule we see this is where the decoder gets stuck in the stopping set.

We see by the third iteration that we have converged to the correct codeword, thus we have zero errors.

This example demonstrates two very valuable improvements:

1. BER is improved. The flooding schedule exits with two errors. The RS exits with zero errors.
2. The number of iterations required for convergence is decreased. We converge to the correct solution after two iterations of RS. We converge to an incorrect solution after three iterations of the flooding schedule.

Simulation results for RS for two LDPC codes are shown in figure 6.1 below:

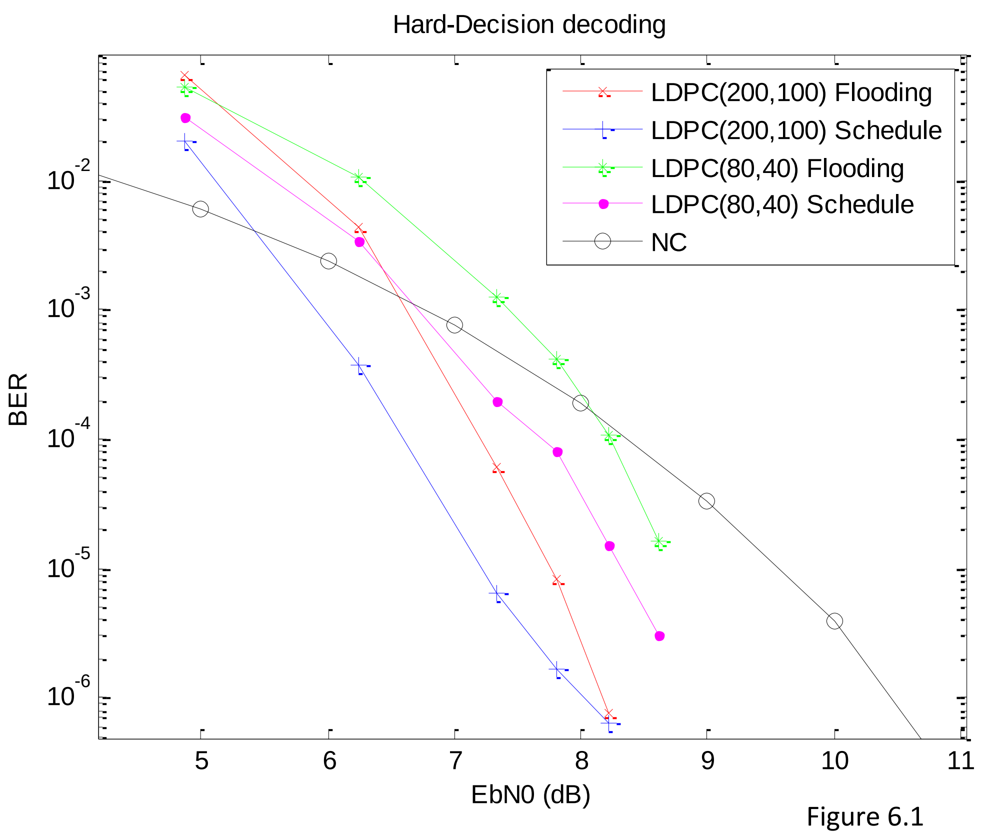


Figure 6.1

Block lengths of 200 and 80 are considered. We see the RS achieves a very respectable improvement in BER. At a BER of 10^{-5} we see a gain of over 0.5dB for a block length of 200.

We also see using RS leads to an improved performance over an un-coded scheme for lower SNR. A block length 200 code outperforms uncoded upwards of 5.5dB for scheduling and 6.5dB for flooding.

Analysis of the decoding complexity requirements for each scheme is shown in figure 6.2.

We compare complexity by comparing the average iterations required for convergence, where one iteration is defined as updating a number of variable nodes equal to the block length. Thus we see partial iterations do not count for a full iteration. This is a fair consideration as partial iterations reduce decoding complexity.

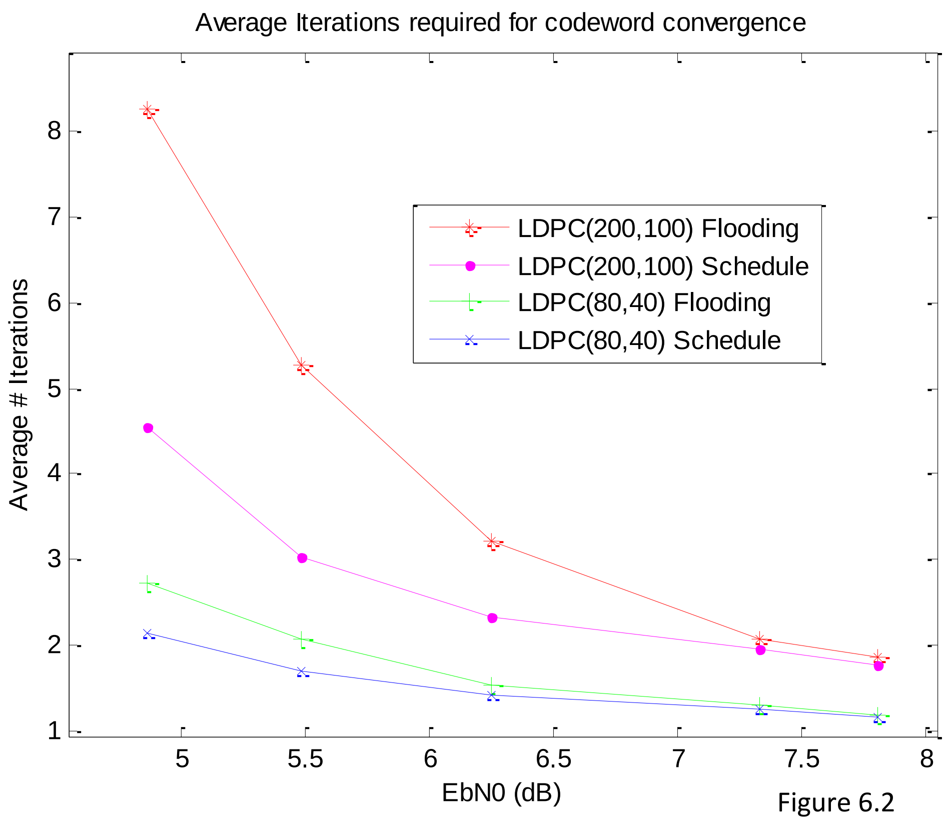


Figure 6.2

Observing the simulation results, we see a considerable saving in complexity using this method. Operating at about 6.5dB, the block length 200 code requires an average of approximately 3 iterations. The same code using the RS requires an average of approximately 2.4 iterations. We see that improvements in complexity above 7.7dB become negligible.

Figure 6.1 and figure 6.2 demonstrate a significant improvement in overall performance.

If we consider the inherent flaws of hard-decision decoding, we see this decoding scheme essentially discards large quantities of information by sticking to binary beliefs.

Considering the results of applying this updating schedule to hard-decision decoding, it might seem worthwhile investigating a means of applying the same concept to soft-decision decoding. However my attempts showed this not to be the case. For the same updating rule in soft-decision decoding, performance was degraded compared to the flooding schedule. Tailoring the requirements of the updating rule did not help significantly. I considered a rule whereby a node would not be updated if all messages had at least an 85% effective probability of a particular bit, and where the indicated bit is the same as the bit indicated by the channel. In figure 6.3 and 6.4 I show simulation results.

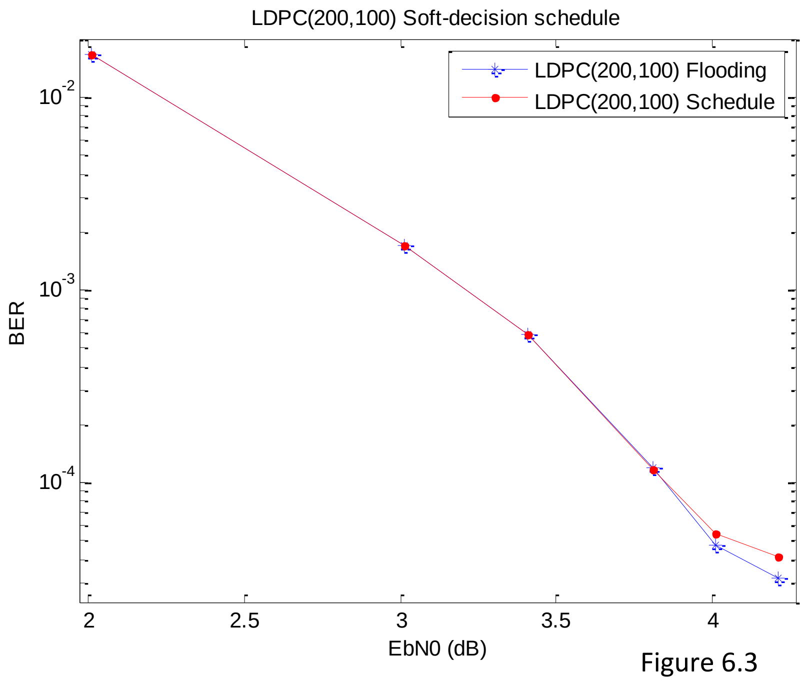


Figure 6.3

We see BER is not improved and is in fact degraded above a certain SNR.

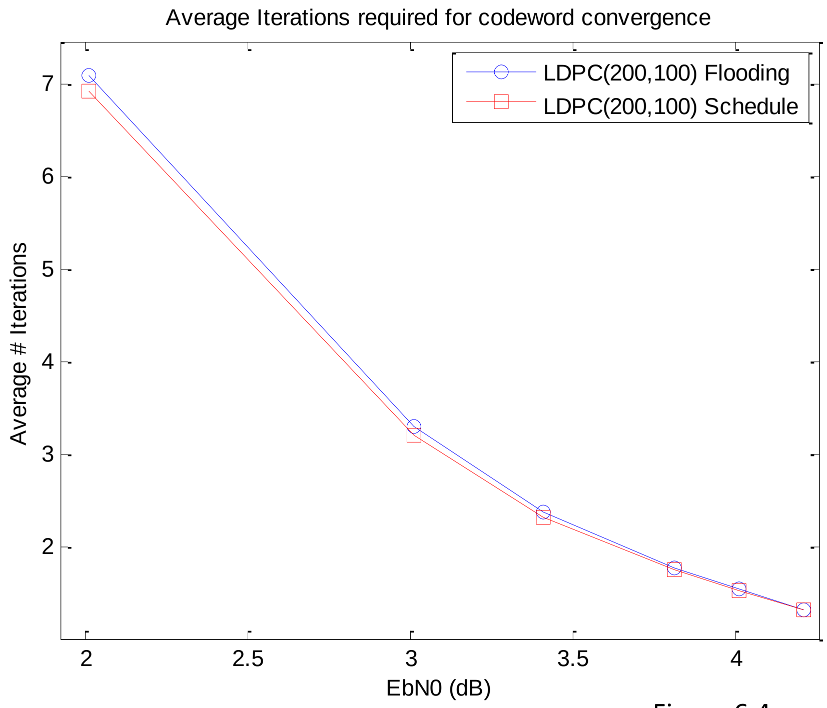


Figure 6.4

We see a negligible decrease in average iterations required for convergence. We clearly see no benefit in applying the same concept to soft-decision decoding. However this helps us in our understanding of why we see such improvement in the hard-decision case.

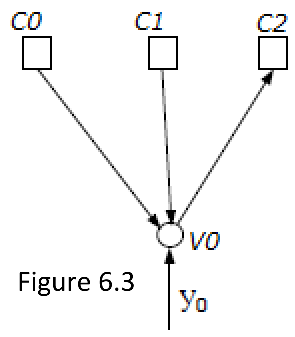


Figure 6.3

Consider node V_0 in figure 6.3. In this example if C_0, C_1 and Y_0 do not all agree with each other, then we consider a majority 2 to 1 vote on what message to send C_2. We see that the result of this vote reflects a confidence level of 0.66 about what the bit is likely to be. This is not an exceptionally strong confidence, yet with HD decoding we round the belief up to 1. This is an enormous information discard. We see the confidence schedule reduces the occurrence of scenarios such as this, and thus reduces the effect of this information loss. SD does not encounter the same problem in its decoding scheme. My understanding is that this reasoning is behind the differences in the schedules effect on SD and HD decoding.

Conclusion

The class of iteratively decoded error correcting codes known as LDPC codes are undoubtedly an extremely capable class of codes. We see that for larger block lengths these codes performance increases considerably while maintaining a linear increase in complexity. It is this fundamental characteristic that separates these codes from Maximum-Likelihood decoding methods.

Scheduling techniques offer us another tool in optimising the performance of these codes. We have seen the conventional decoding process can be affected significantly by the use of particular updating rules in each iteration of the decoding process. From the systems studied in this project we can see two distinct methodologies in how we might improve convergence behaviour through scheduling.

The first is by utilising a system which improves the efficiency by which information propagates around the graph. We aim to compute the maximum amount of reliable information possible while minimising the computations required to do so. We see this in the serial methods of updating where we demonstrated that computations carried out in parallel are less efficient than if ordered in an optimised way.

The second methodology revolves around identifying the fact that often it is unnecessary or undesirable to update every node, we attempt to efficiently update only those nodes which we can identify as less reliable. Perhaps avoiding nodes involved in problematic graphical structures such as stopping sets. We see this through our discussion of the 'lazy' schedule and through simulation of the confidence schedule. We These two methodologies loosely represent non-adaptive and adaptive schedules respectively.

We see many of the underlying concepts involved in many decoding schedules have quite an intuitive explanation. It's not hard to see that simply flooding all messages at once, every iteration is far from efficient, and that is what has been demonstrated through simulations of these codes.

References

1. "A Mathematical Theory of Communication": CE Shannon 1948: The Bell System Technical Journal, Vol. 27, pp. 379–423, 623–656, July, October, 1948.
2. "On the performance of turbo codes" Wang, C.C. 18-21 Oct 1998 Military Communications Conference, 1998. MILCOM 98. Proceedings, IEEE (Volume:3)
3. "Low-Density Parity-Check Codes": Robert Gallager, 1963
4. "Near Shannon Limit Performance of Low Density Parity Check Codes" David J.C. MacKay, July 12, 1996
5. Book: "Coding and information Theory", Richard hamming, 1986
6. "An algorithm for the computation of the minimum distance of LDPC codes", Fred Daneshgaran, Euro. Trans. Telecomms. 2006; 17:57–62
7. "On the complexity of the Rank Syndrome Decoding problem", Philippe Gaborit, 6 Jan 2013
8. "Trends and Challenges in LDPC Hardware Decoders" - Tinoosh Mohsenin and Bevan Baas, , University of California
9. "A Recursive Approach to Low Complexity Codes ", Michael Tanner, 1981
10. Sarah Johnson: "Introducing Low-Density Parity-Check Codes" ACoRN Spring School

11. Mackay and Neal: Information Theory, Inference, and Learning Algorithms, Published September 2003
12. “The Analysis of Error Floor and Graphical Structure of LDPC Codes”, Mehdi Karimi Dehkordi, September, 2013
13. “Finding Small Stopping Sets in the Tanner Graphs of LDPC Codes”, Gerd Richter, University of Ulm, Department of TAIT
14. “An Introduction to LDPC Codes”, William E. Ryan, University of Arizona, August 19, 2003
15. “Convergence Analysis of Serial Message-Passing Schedules for LDPC Decoding” , Eran Sharon, April 3, 3006, Munich
16. “Efficient Serial Message-Passing Schedules for LDPC Decoding” , Eran Sharon, IEEE TRANSACTIONS ON INFORMATION THEORY, VOL. 53, NO. 11, NOVEMBER 2007
17. “Lazy Scheduling for LDPC Decoding”, Daniel Levin, IEEE COMMUNICATIONS LETTERS, VOL. 11, NO. 1, JANUARY 2007