An Investigation into Linear Codes & The Viterbi Algorithm

Max Vincent John Simmonds
10393333

January 12, 2017

Abstract

This report focuses on linear codes (more specifically, the Hamming Code) and the Viterbi algorithm for decoding messages. It gives a concise explanation of Hamming codes, and an in-depth look into the Extended Hamming code - Hamming (8,4,3) code - and the construction of the parity check matrix ($\mathbf{H}$) and the generator matrix ($\mathbf{G}$). It compares the two trellis that can be generated from the $\mathbf{G}$ and $\mathbf{H}$ matrices, and shows that for $n-k \ll k$ the $\mathbf{H}$ matrix is the preferred method, and for $n-k \gg k$ the $\mathbf{G}$ matrix method is preferred due to the reduction in complexity. Codeword enumeration, syndrome decoding and Viterbi decoding are discussed with worked examples to show that the Viterbi decoding is the best solution. An example of decoding using the built trellises compared to the codeword enumeration approach is also shown, with a quantitative analysis on the reduction in complexity of the two methods.

0.1 Introduction

In today's communication age there is obviously a need to transmit and receive data reliably. With the increased need of faster down and up links for services such as mobile Internet, or faster data transfers on PCs with solid state drives, there is a need for better and more resilient codes. Many codes are being revisited due to an increase in computing power that rendered these 'old codes' impossible when they were first developed in the mid 90's. An increase in computing power and advancements in digital electronics is now making impossible codes, possible. The Hamming code was invented in the 1650's by Richard Hamming. There has been much research into these codes, and many papers discussing how to reduce the decoding complexity. This report attempts to investigate the decoding problems associated with this code.

0.2 Extended Hamming - The Chosen Linear Code

The requirements are that the chosen code uses at least 4 parity bits, the most common Hamming code in literature is the [7,4,3] Hamming. This of course contains only 3 parity bits, since the length of the messages is $n = 7$ we need enough parity bits to check 7 messages. In this case, the number of parity bits is 3, $2^3 = 8$. It could indeed be possible to choose a [15,11,3] Hamming code, which would satisfy the 4 parity bit requirement ($2^4 = 16$, this is $\geq$ 'n') , however the matrices involved would be very large. This added complexity of large matrices does not aid in understanding - a simpler code would suffice. It is for this reason that the *Extended Hamming* code will be used. This adds an extra parity bit to any Hamming code - this improvement increases the minimum distance from $d_{min} = 3$ to $d_{min} = 4$. The extended Hamming code (also known as SECDED, abbreviated from Single Error Correction, Double Error Detection) can not only correct one error, but detect up to 2 bit errors where the previous non-extending Hamming code could not.

Thus, the chosen code is the [8,4] Hamming. Hamming is one of the first codes that could not only detect an error, but also correct errors too. By interlacing 'error correcting bits' into the codeword itself erroneous bits can be identified. There are a set of rules that need to be followed in order to construct an [n,k] Hamming code, these are as follows [4]:

- For an [n,k] Hamming, write the binary representation as columns in a table, from 1 $\rightarrow$ n. For example see table 1:
- All bit positions that are powers of two are parity bits, IE: 1, 2, 4, 8, etc. (1, 10, 100, 1000)

Bit Number			
1	0	...	1
0	1	...	1
0	0	...	1
⋮	⋮	⋮	⋮
0	0	...	1

Table 1: Bit numbers as per the general rules

- All other bit positions are data bits.
- Each data bit is included in a unique set of 2 or more parity bits, as determined by the binary form of its bit position.
  - Parity bit 1 covers all bit positions which have the least significant bit set: bit 1 (the parity bit itself), 3, 5, 7, 9, etc.
  - Parity bit 2 covers all bit positions which have the second least significant bit set: bit 2 (the parity bit itself), 3, 6, 7, 10, 11, etc.
  - Parity bit 4 covers all bit positions which have the third least significant bit set: bits 47, 1215, 2023, etc.
  - Parity bit 8 covers all bit positions which have the fourth least significant bit set: bits 815, 2431, 4047, etc.
- In general each parity bit covers all bits where the bitwise AND of the parity position and the bit position is non-zero.

The form of the parity is irrelevant. Even parity is simpler from the perspective of theoretical mathematics, but there is no difference in practice. To help explain these general rules, they shall be applied to the [7,4] Hamming, as in Table 2:

[7,4] Hamming Code Bit Table						
p_0	p_1	i_0	p_2	i_1	i_2	i_3
$1_d = 2^0$	$2_d = 2^1$	3_d	$4_d = 2^2$	5_d	6_d	7_d
1	0	1	0	1	0	1
0	0	1	1	0	0	1
0	0	0	0	1	1	1

Table 2: Bit numbers as per the general rules, p = parity bits, i = data bits

For the [7,4] Hamming, the parity bits $(p_0, p_1, \& p_2)$ can be expressed in terms of the information bits $(i_0, i_1, i_2, \& i_3)$ by the following parity check equations:

$$A : p_0 = i_0 + i_1 + i_3 \tag{1}$$
$$B : p_1 = i_0 + i_2 + i_3 \tag{2}$$
$$C : p_2 = i_1 + i_2 + i_3 \tag{3}$$

Note: This is modulo 2 addition $(1 + 1 = 0, 1 + 0 = 1)$

These equations can be used to ensure no information bits had been corrupted during transmission, the result of these parity bits should be 0. If it is not zero, then the binary interpretation of the three parity bits points to the bit that was corrupted. For example, if the parity check equations A, B, & C sum was 1,0, & 1 respectively, then this would indicate that bit number 5 ($101_b = 5_d$) was corrupted and bit flipping (negation) would take place on bit 5. As aforementioned, this paper will be discussing the use of the extended Hamming code, with an extra parity bit. This parity bit takes the place of the 0 bit in Tables 1 and 2. The extended table is shown in Table 3 and the parity check equation for the extended Hamming is shown in equation 4.

$$D : p_4 = p_0 + p_1 + p_2 + i_0 + i_1 + i_2 + i_3 \tag{4}$$

[8,4] Extended Hamming Code Bit Table							
p_0	p_1	i_0	p_2	i_1	i_2	i_3	p_4
$1_d = 2^0$	$2_d = 2^1$	3_d	$4_d = 2^2$	5_d	6_d	7_d	0_d
1	0	1	0	1	0	1	0
0	0	1	1	0	0	1	0
0	0	0	0	1	1	1	0

Table 3: Bit table with extended Hamming parity

0.2.1 H Matrix

The parity check equations can be expressed in a different form than that of the above, a so called 'H matrix'. The matrix form is preferred over the algebraic form because with increasing values of n and k, the equations can become cumbersome. The **H** matrix is generated by transforming the table (Table 3) into a matrix, seen in equation 5:

$$H_{nsys} = \left[\begin{array}{ccccccc|c} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ \hline 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array}\right] \tag{5}$$

The lines indicate the extension to the [7,4] Hamming, to produce the [8,4] Hamming code. The **H** matrix has the same function as the parity check equations, but in a nicer form - allowing linear algebra to be used. Currently, the matrix is in the 'non-systematic' form. This means that the data and parity check bits are interleaved. Of course, mathematically this does not affect how the matrix works, but it adds complexity in understanding. There is also another reason to make the **H** matrix into systematic form, and that is that the *Generator Matrix* that generates codewords (explained below) is formed from the **H** matrix, and this can only be done when the **H** and **G** matrix satisfy equations 6 and 7 respectively:

$$\mathbf{H_{sys}} \coloneqq \left[\begin{array}{c|c} P & I_{n-k} \end{array}\right] \tag{6}$$

$$\mathbf{G_{sys}} \coloneqq \left[\begin{array}{c|c} I_k & -P^T \end{array}\right] \tag{7}$$

Elementary row operations can be performed on the non-systematic **H** matrix (equation 5) to transform it into the form of equation 6, thus making it the systematic H matrix, H_{sys} (equation 9). This H_{sys} matrix can then be used to obtain the G_{sys} matrix. However, the G_{sys} matrix and H_{sys} matrix are not used in practise, there non-systematic form, with the rows in the following order: p_0, p_1, i_0, p_2, i_1, i_2, i_3, p_3.

$$
\begin{aligned}
H_{nsys} &= \left[\begin{array}{cccc|cccc} 1&0&1&0&1&0&1&0\\ 0&1&1&0&0&1&1&0\\ 0&0&0&1&1&1&1&0\\ 1&1&1&1&1&1&1&1 \end{array}\right] \\
&= \left[\begin{array}{cccc|cccc} 1&0&1&0&1&0&1&0\\ 0&0&0&1&1&1&1&0\\ 0&1&1&0&0&1&1&0\\ 1&1&1&1&1&1&1&1 \end{array}\right] R_2 \leftrightarrow R_3 \\
&= \left[\begin{array}{cccc|cccc} 1&0&1&0&1&0&1&0\\ 1&0&1&1&0&1&0&0\\ 0&1&1&0&0&1&1&0\\ 1&1&1&1&1&1&1&1 \end{array}\right] R_2 \leftarrow R_1 + R_2 \\
&= \left[\begin{array}{cccc|cccc} 1&0&1&0&1&0&1&0\\ 1&0&1&1&0&1&0&0\\ 1&1&0&1&0&0&1&0\\ 1&1&1&1&1&1&1&1 \end{array}\right] R_3 \leftarrow R_2 + R_3 \\
&= \left[\begin{array}{cccc|cccc} 0&1&1&1&1&0&0&0\\ 1&0&1&1&0&1&0&0\\ 1&1&0&1&0&0&1&0\\ 1&1&1&1&1&1&1&1 \end{array}\right] R_1 \leftarrow R_1 + R_3 \\
&= \left[\begin{array}{cccc|cccc} 0&1&1&1&1&0&0&0\\ 1&0&1&1&0&1&0&0\\ 1&1&0&1&0&0&1&0\\ 1&0&0&0&0&1&1&1 \end{array}\right] R_4 \leftarrow R_4 + R_1 \\
&= \left[\begin{array}{cccc|cccc} 0&1&1&1&1&0&0&0\\ 1&0&1&1&0&1&0&0\\ 1&1&0&1&0&0&1&0\\ 0&0&1&1&0&0&1&1 \end{array}\right] R_4 \leftarrow R_4 + R_2 \\
H_{sys} &= \left[\begin{array}{cccc|cccc} 0&1&1&1&1&0&0&0\\ 1&0&1&1&0&1&0&0\\ 1&1&0&1&0&0&1&0\\ 1&1&1&0&0&0&0&1 \end{array}\right] R_4 \leftarrow R_4 + R_3
\end{aligned}
\tag{8}
\tag{9}
$$

Thus, we now have the matrix $\mathbf{H}$, in systematic form as shown in equation 6. From this the $\mathbf{P}$ matrix can be found:

$$
\mathbf{P} = \begin{bmatrix} 0&1&1&1\\ 1&0&1&1\\ 1&1&0&1\\ 1&1&1&0 \end{bmatrix} \tag{10}
$$

$$
\mathbf{P^T} = \begin{bmatrix} 0&1&1&1\\ 1&0&1&1\\ 1&1&0&1\\ 1&1&1&0 \end{bmatrix} \tag{11}
$$

0.2.2 G Matrix

The systematic **G** matrix can now be made using the equations 11 & 7, shown in equation 12:

$$\mathbf{G_{sys}} = \left[\begin{array}{cccc|cccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{array}\right] \tag{12}$$

Since elementary row operations have been used to obtain this matrix, the parity bits are in fact linearly independent of one another (which allows for error detection and correction), but this means the matrix (equation 12) is currently in the form: i_0, i_1, i_2, i_3, p_0, p_1, p_2, p_3. In this form, the result of the error detection will not be a binary number. Therefore, equation 12 should be column swapped to match the non-systematic **H** matrix (equation 9):

$$\mathbf{G_{nsys}} = \left[\begin{array}{cccccccc} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \end{array}\right] \tag{13}$$

0.3 Methods of Decoding the Extended Hamming

Two forms have been shown to describe the extended Hamming code. The first is the general form given as a set of algebraic equations. The second, is transforming these equations into matrices **G** and **H**. There are several methods of decoding that can be used: syndrome decoding, Viterbi decoding, and codeword enumeration decoding will be discussed here. These three methods can be grouped into either one or both of two categories. Namely, these are soft and hard decision decoders.

0.3.1 Hard Decision Decoding

Hard decision decoders are ones that *decide* the value before the decoding algorithm, IE coerce the value to '-1' or '+1'. So for example, the syndrome decoder is an example of a hard decision decoder.

Syndrome Method

An example of the syndrome decoder method is shown below; assuming a received vector of:

$$\mathbf{r} = \left[\begin{array}{cccccccc} +1.1 & -0.3 & +1.3 & -0.4 & +0.7 & -0.3 & +0.1 & -0.5 \end{array}\right] \tag{14}$$

$$\widetilde{\mathbf{r}} = \left[\begin{array}{cccccccc} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{array}\right] \tag{15}$$

Then, to decode this using the matrix method, one must simply hard decides the values, based on its sign. '-' is mapped to a 0, and '+' is mapped to a 1. Error detection is done by multiplying the received vector by the parity check matrix. The result is the error syndrome, which is also the index of the erroneous bit (unless a 2 bit error occurs, then the syndrome indicates this but the error cannot be rectified through this method) when converted to binary. An example calculation is shown in equation 16.

$$ErrorSyndrome = \mathbf{H} \cdot \mathbf{r^T} = \left[\begin{array}{cccc|cccc} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{array}\right] \left[\begin{array}{c} 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{array}\right] \tag{16}$$

The problem with this method of hard decision decoding is values near the threshold (in this case, the threshold is 0) can be decided incorrectly. For example, the received vector (equation 14) has a very suspicious '+0.1'. Being close to the threshold means that it could also quite easily be '-0.1', which would be hard decided to '0', thus introducing errors. In

this case, the error could easily be accounted for due to the error detection and correction powers of the Hamming code. Assuming that an error had occurred on this bit then:

$$\mathbf{r_e} = \begin{bmatrix} +1.1 & -0.3 & +1.3 & -0.4 & +0.7 & -0.3 & -0.1 & -0.5 \end{bmatrix} \tag{17}$$

$$\widetilde{\mathbf{r_e}} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{18}$$

The result of equation 16 is now no longer 0, indicating an error has occurred:

$$ErrorSyndrome = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \tag{19}$$

This is 2_d, and therefore bit 2 needs to be flipped to correct the error. However, it may not always be the case that only one error occurs, in the event that more than one happens the code will be irreversibly changed if we use hard decision decoding.

0.3.2 Soft Decision Decoding

Soft decision decoders resolve the issue outlined above with regards to wrongly decided values. When values are hard decided, information is lost due to rounding. Soft decision decoders do not remove any information, and therefore are the most accurate and can sometimes correctly decode received vectors with more than 1 error.

Codeword Enumeration

Codeword enumeration is a general method that can be used to decode any code. The method works by exhaustively comparing every possible codeword, with the received vector. The comparison is based on the principle of *Maximum Likelihood*, IE, given the received vector, what was most likely the transmitted codeword? This is done by calculating the correlation sum (equation 20) for every codeword. An example of this correlation sum, using the same codeword as in equation 14, is shown in equation 23.

$$M_i = \sum_{j=1}^{N} v_j^i r_j \tag{20}$$

$$\mathbf{r} = \begin{bmatrix} +1.1 & -0.3 & +1.3 & -0.4 & +0.7 & -0.3 & +0.1 & -0.5 \end{bmatrix} \tag{21}$$

$$\mathbf{v} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \tag{22}$$

$$\begin{aligned} M = (+1.1 \times +1.0) + (-0.3 \times -1.0) + (+1.3 \times +1.0) + (-0.4 \times -1.0) \\ + (+0.7 \times +1.0) + (-0.3 \times -1.0) + (+0.1 \times +1.0) + (-0.5 \times -1) \\ = 4.7 \end{aligned} \tag{23}$$

Of course, this number alone does not state that the codeword in equation 22 was the one transmitted. One needs to calculate all possible correlation sums and choose the highest value. Therein lies the problem with codeword enumeration. The number of codewords is equal to $2^k = 2^4 = 16$, each codeword requires 8 multiplications, and 7 additions. For the [8,4] Hamming, the complexity is not too high, but this is a simple code. Most codes have hundreds of information bits, hence k is large. The complexity of this method of decoding then grows exponentially and eventually becomes impossible to use.

0.3.3 Viterbi Decoding

The Viterbi decoder is a solution to both the problems outlined above, it is a Maximum Likelihood, Soft Decision (MLSD) decoder, that uses the Viterbi algorithm. It does not have the same complexity as the codeword enumeration approach, with increasing k, the complexity simply increases linearly and not exponentially. Trellises are often used to graphically display the viterbi decoder, and they are generated from the **G** and **H** matrices. The construction of the trellis for both matrices is discussed in the next section.

A trellis is a time-indexed graph that represents a given linear code, containing *states/vertices's/nodes* and *edges*. States are generally represented by dots, and edges are lines. States/nodes are grouped into vertical slices, called times. And the times are ordered such that the edges connect a node from one time, to the next. Every edge is given a *symbol*, following these symbols along the branches gives a codeword. Therefore, a trellis must contain all codewords of a code, and nothing more. All nodes used in the trellis must have at least one edge in, and one edge out of the node - except the two extreme nodes (named root and toor).

0.4 Trellis Construction

As mentioned, the trellises are constructed from the two matrices, **H** and **G**. Each one has its advantages and disadvantages over the other. These will be discussed in the next section, for now only the construction of the trellis will be discussed.

0.4.1 Trellis Construction from Parity Check (H) Matrix

The trellis can be expressed in terms of partial syndromes, each partial syndrome becomes a state in the trellis at one time coordinate. Equation 16 calculates the entire syndrome, to calculate the partial syndrome the product of **H** with the codeword generated thus far is calculated. The equation for state transitions is shown in equation 24.

$$S' = S + h_i C_i \tag{24}$$

Where S' is the next state, S is the current state, h_i is the column vector of the **H** matrix, and C_i is the bit at index i from the codeword C.

For example, assuming the current state is at the root of the trellis, and the non-systematic parity check matrix is used (equation 8) then S' for the two possibilities $C_i = 0, 1$ is calculated as follows:

$$S'_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \cdot 0 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{25}$$

$$S'_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \cdot 1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{26}$$

The partial syndromes are calculated for all possible transitions through the trellis, starting at the root. A valid code word results in a transition that ends in the toor state. Therefore, once all possible transitions have been generated, invalid transitions must be stripped back from the trellis, leaving only transitions that go from the root to the toor nodes. Because **H** is an $m \times n$ matrix, where $m = n - k$. The partial syndromes generated will be m-bit vectors. Thus, 2^m nodes are needed in each time slice. It should be noted that if $C_i = 0$ then the result will always be $S' = S$, and therefore there is no need to compute these values, which reduces complexity by half. If one was to carry on with calculating all the partial syndromes from the above example, then one would generate the trellis shown in figure 1.

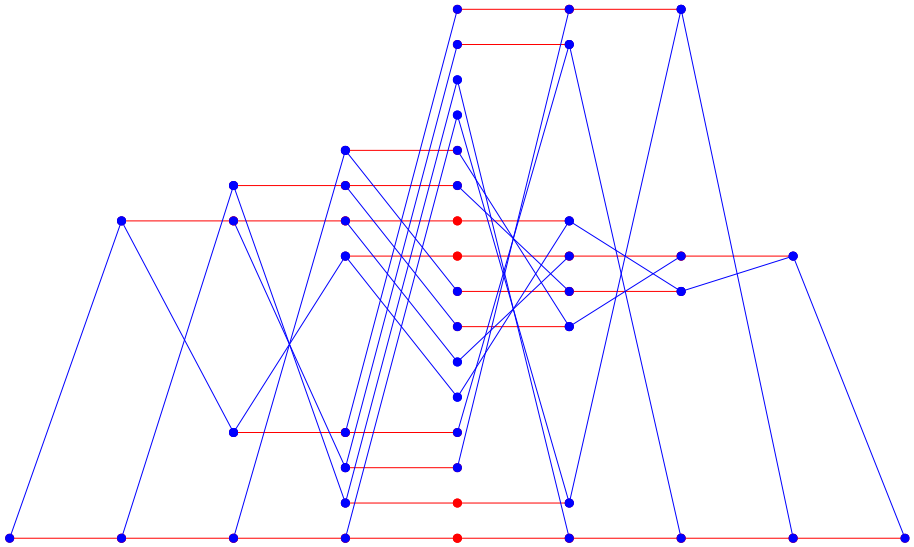


Figure 1: Trellis obtained from the non-systematic **H** matrix (equation 8). Blue lines indicate the edge symbol 1, and red lines indicate an edge symbol 0

Figure 1 shows only valid codeword transitions, A shows all possible transitions, with valid ones highlighted in blue. Note how the two calculated transitions used in the example above are on this trellis.

0.4.2 Trellis Construction from Generator (G) Matrix

To construct the trellis from the generator matrix, it must first be transformed into a Trellis Orientated Generator Matrix (TOGM). In this form, each row of the matrix must have a span that starts and finishes in a different column to any other row. A very simple example of this is shown in matrix 28.

$$Vector \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \tag{27}$$

$$Span \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \end{bmatrix} \tag{28}$$

It can be seen that the span of the first row starts at column 2, and ends at column 5 - the span of the second row starts at column 1 and ends at column 4. Thus there are no overlapping spans, and therefore this is a TOGM. In order to put the non-systematic **G** matrix into TOG form, one can perform elementary row operations (similar to that used in Gaussian elimination). The row operations performed on the **G** matrix are shown in equations 29.

$$
\begin{aligned}
\mathbf{G_{nsys}} &= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix} R_1 \leftarrow R_1 - R_2 \\
&= \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix} R_2 \leftarrow R_2 + R_3 \\
\mathbf{G_{TOG}} &= \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 1 & & \\ & 1 & 1 & 1 & 1 & & & \\ & & 1 & 0 & 1 & 1 & 0 & 1 \\ & & & 1 & 1 & 1 & 1 & \end{bmatrix} R_2 \leftarrow R_2 - R_4
\end{aligned}
\tag{29}
$$

Note: Some zeros have been omitted from the matrix 29 to improve readability, this allows the span of the rows to be seen easier to ensure it is in TOG form.
Each row of $\mathbf{G_T OG}$ can now be thought of as a subcode of the Hamming [8,4] code. Each row in this case has two codewords associated with it. For example, the first row consists of 00000000 and 11001100. These subcodes can then be used to build the *minimal trellis* of the Hamming code, for said **G** matrix. A minimal trellis is one where each node has, at most, two edges in and two edges out. All nodes in a given time must have the same left degree as right degree, IE if there are two edges entering a node, then all nodes in that time must also have two entering.
Building the minimal trellis from these subcodes is easy, one simply follows these rules:

1. The width of the trellis (the number of nodes) is equal to n+1, the message length
2. Traversing from the root to toor should produce all the subcodes, and no more
3. At the start of the span, the trellis changes to state $2^{row-1} \bmod 2^{n-k}$ where 'row' is the index of the row the subcode originated from and n = number of messages, and k = the number of information bits
4. At the end of the span, the trellis changes to the 0 state
5. All edges of the trellis are labelled in the order that they appear in the subcode

To better explain this, the matrix in 29 will be used. The first subcodes have already been given: 00000000 and 11001100. Therefore, we can see that there will be a change from state 0000_b to state $2^1 \bmod 2^{8-4} = 2 - 1 \bmod 4 = 0001_b$ at the start. And a change from 0001_b to 0000_b at time = 6. This is shown in figure 2:

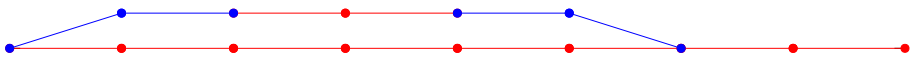


Figure 2: Trellis generated from the subcode originating from row 1 of the non-systematic G matrix

Using the same rules, all four trellises can be generated, these are shown in figure 3.

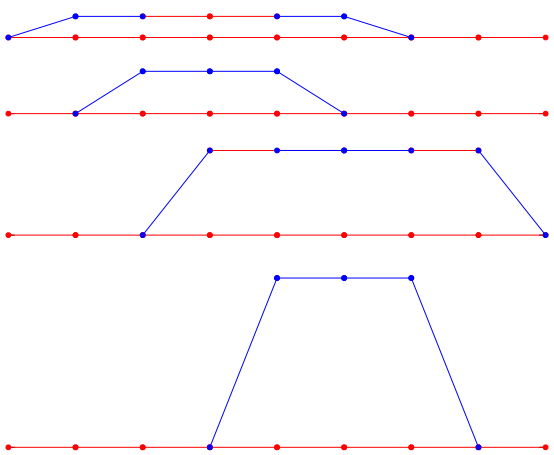


Figure 3: Trellis generated from row 1, 2, 3, and 4 respectively

These sub-trellises now need to be combined, to give the overall trellis for the code. To do this, again we follow a set of rules:

1. The trellises are built incrementally, that is the trellis from the subcode of row 1 is added to row 2. Then row 3 is added to this, then row 4 etc.
2. Nodes within the span of the new subcode are duplicated, as are the edge symbols of the original trellises. Essentially, each sub-trellis is left untouched and overlay each other
3. The nodes of each sub trellis are added together to give a new transition
4. The edge symbols associated to the new transition is the modulo 2 addition of the edge symbol from one trellis, and the other (for example $1 + 1 = 0$)

To better illustrate this, figure 4 shows the the incrementally built trellis from the non-systematic $\mathbf{G}$ matrix.

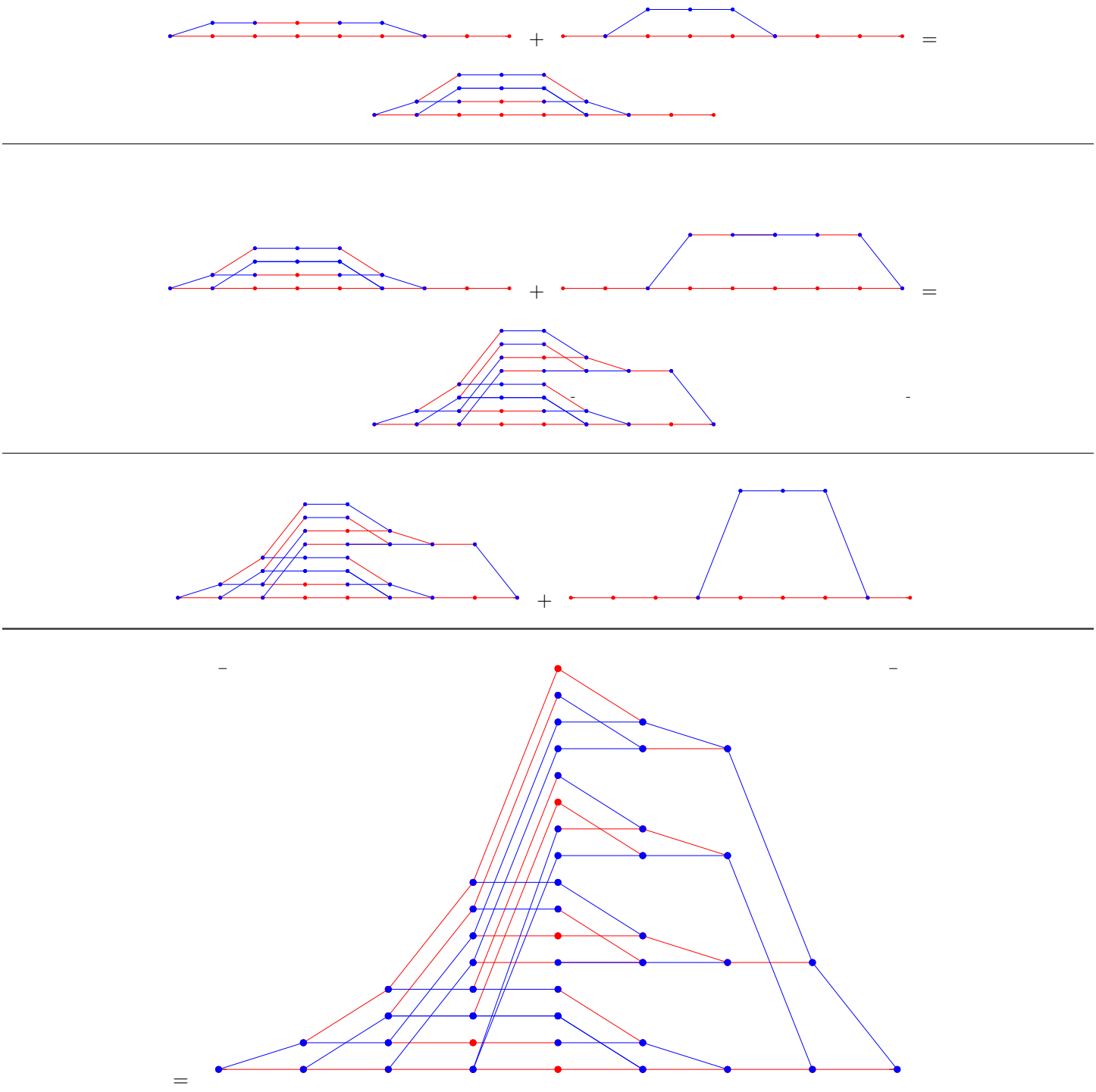


Figure 4: Trellis generated from the addition of row 1, 2, 3, and 4 respectively. As before, blue lines indicate the edge symbol 1, and red lines indicate an edge symbol 0

It is noteworthy to mention that both trellises in figures 1 and 4 (constructed from the **H** and **G** matrix respectively) are noticeably different, yet give precisely the same set of codewords as each other.

0.4.3 Decoding Using Viterbi and Trellises

As mentioned, The Viterbi decoder is MLSD based, and therefore very resilient to errors. The method of using a trellis is easy, one must first place the received values along the top of the trellis, one per section. This then allows the weighting of each edge to be calculated as follows: if the symbol on the edge is a 0, then the weighting is equal to the *negated* received value, if the edge symbol is 1, then the value is equal to the received value. One then starts from the root node at an initial value of 0. The node metrics are calculated by adding the transition weighting to the previous node metric. This is done for all nodes. For each node, keep only the *largest* value, and remember which node it came from. An example of this is outlined in figure 5. The received vector is the same as that shown in equation 17.

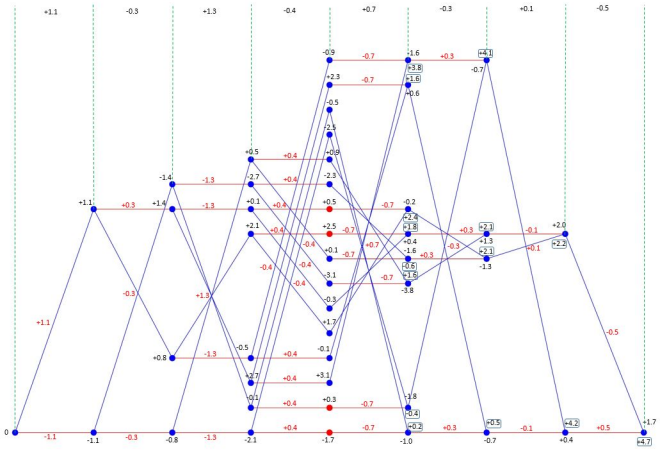


Figure 5: Decoding a received vector using the trellis generated from the parity check matrix

Transition weightings are coloured red, while node metrics are black. A ring around a value indicated that it is the largest at that node.

Finally, when the toor node is reached, one should trace back through the trellis along the nodes with the largest values. Upon reach the root node again, the codeword can be read directly from the trellis using the edge symbols, this is shown in figure 6.

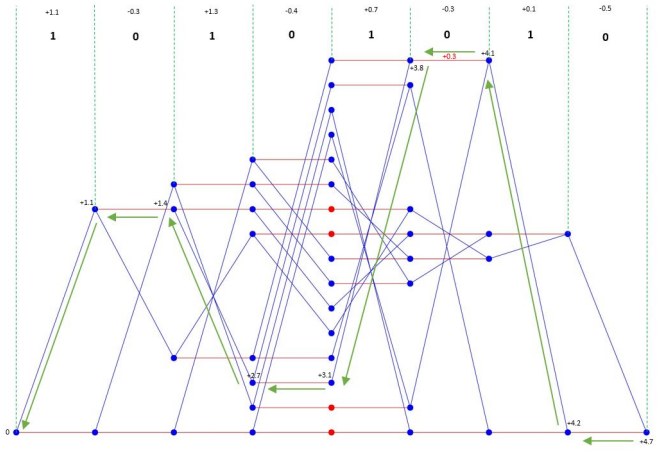


Figure 6: The decoded code word - 10101010

It can easily be seen that the transmitted vector was 10101010, and by either using a lookup table, or the generator matrix, one can see that the transmitted information was 1101. Note, that this method can also correct more than one errors, figure 7 shows a two bit error (bits 7 & 8), however, it does no effect the result and the decoded value is still the same.

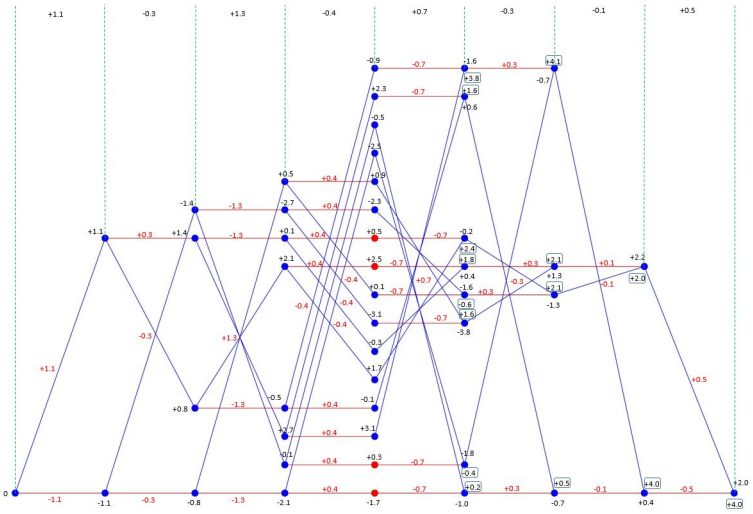


Figure 7: The decoded code word is still 10101010 even after two bit errors

0.5 Comparison of Decoder Complexity

As mentioned, it is possible to decode every known code via the codeword enumeration method - however, this has incredibly high over heads. For the example of the Extended Hamming, with a message length of 8, the reduction in complexity is not that high. The number of possible codewords are $2^k = 2^4 = 16$. For the codeword enumeration approach, one must to 8 multiplications and 7 additions per bit per codeword. Hence, the number of calculations (and therefore the complexity, $\mathcal{O}$) is given by:

$$\mathcal{O}_{CE} = 2^k \cdot (n + (n-1)) \tag{30}$$

Which in the example of the extended Hamming, gives a complexity of 240. With the trellis, the complexity is given by:

$$\mathcal{O}_T = 2 \cdot \sum_{i=1}^{i=k} 2^i \tag{31}$$

Again, in the case of Hamming this gives a complexity of 60, 4 times less calculations. These are also only additions, which are far less computational than multiplications (additions are usually 1 machine cycle, whereas multiplications can be up to 4 or more). A better example of reduced complexity would be a 1/2 rate convectional code - such as the (1000, 500). The possible codewords are $2^{500} \approx 3 \times 10^{150}$ and therefore impossible to enumerate all codewords, making a trellis (which would have only 8 states at each stage) the only possible decoding method.

Having said this, generating the trellis initially has added overheads, and therefore there are considerations to take into account when deciding which method to generate a trellis from; the parity check or the generator matrix. The method of generating a trellis from the parity check matrix is not memory efficient, and is computationally wasteful. In the example of the parity check matrix (figure 1), 158 states are computed $(2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^5 + 2^5 + 2^5)$ for each state of the trellis respectively, but only 56 of these are actually used $(2^1 + 2^2 + 2^3 + 2^4 + 2^4 + 2^3 + 2^2 + 2^1)$ - the rest are discarded. This means that $\approx \%65$ of the calculations done are irrelevant.

When the value of $n - k \ll k$, IE when the width of the trellis is much smaller than the number of information bits, the problem of redundant calculations via the parity check matrix becomes negligible and hence this is the best method to use. With the generator matrix trellis construction, the time taken to form the TOGM outweighs the time in redundant calculations. However, when the width of the trellis becomes larger, $n-k \gg k$, (and therefore more redundant 2^k calculations are performed) it becomes more efficient to use the generator matrix trellis. The reason for this is that it generates the minimal trellis for a given $\mathbf{G}$ matrix permutation, with no redundant calculations. While it is true that the parity check trellis also is minimal for a given permutation, it has redundant calculations.

0.5.1 Complexity Reduction Via Matrix Permutation

It was mentioned in the previous section, generating a trellis from the **G** and **H** matrix results in the minimal trellis design for a given matrix *permutation*. This implies that permuting a matrix changes the trellis design. This could lead to a decrease in complexity, if the new design has a significant difference in nodes or branches. In the limited experiments carried out during this report, no change in complexity was found, however there are documented results of a decreases in complexity [1].

An investigation was carried out to see if the complexity of the trellises could be reduced. The parity check matrix was permuted (via the column operations outlined in equation 33) to give the matrix shown in equation 32.

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{32}$$

$$C_1 \leftrightarrow C_4 \mid C_3 \leftrightarrow C_1 \mid C_4 \leftrightarrow C_3 \mid C_5 \leftrightarrow C_6 \mid C_7 \leftrightarrow C_8 \tag{33}$$

Figure 8 shows the trellis generated from this permuted matrix.

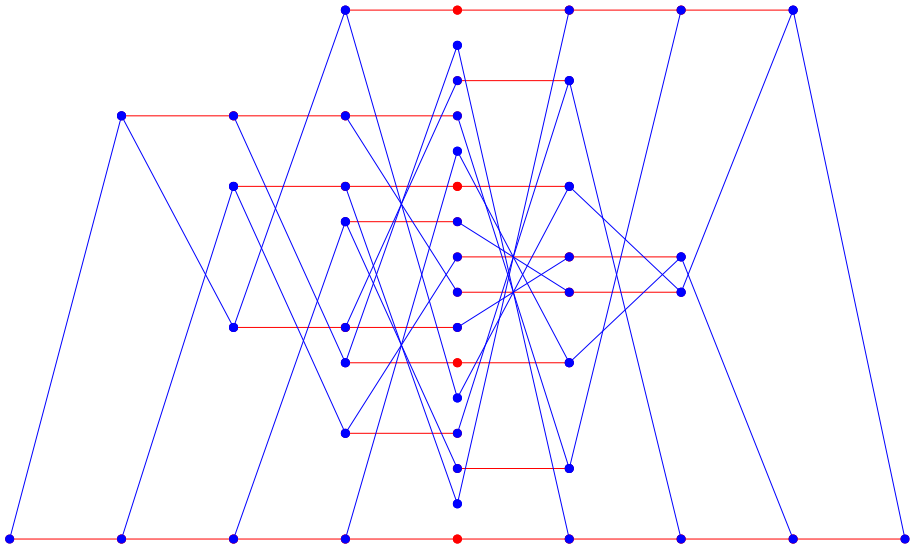
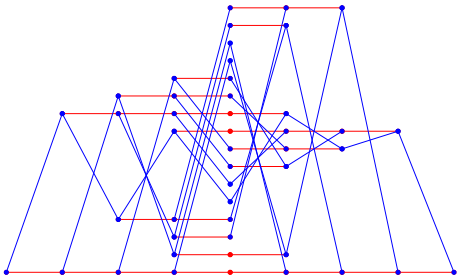


Figure 8: Trellis generated from permuted H matrix

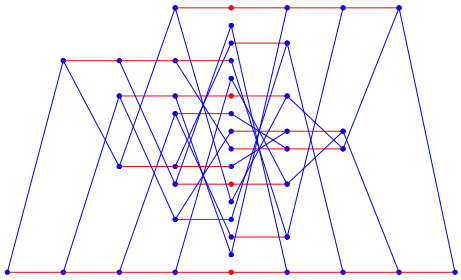
The trellis complexity can be compared by a number of metrics. The number of nodes and branches was decided to be used as this is a direct relation to complexity. Table 4 shows the relative complexity's of all the trellises, show in figure 9.

Trellis	Nodes	Branches
Figure 9a	46	60
Figure 9b	46	60
Figure 9c	46	60

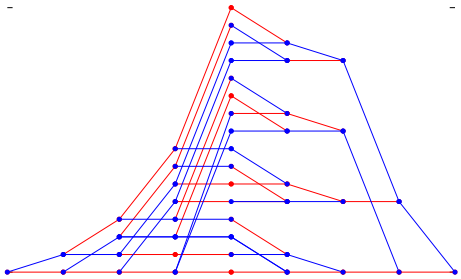
Table 4: Comparing complexity of different trellis designs



(a) Trellis generated from **H** matrix (equation 8)



(b) Trellis generated from $\mathbf{H_p}$ matrix (equation 32)



(c) Trellis generated from **G** matrix (equation 13)

Figure 9: The 3 trellises generated from the 3 different matrices

Note: since the trellis in 9b is from a permuted matrix, the order of the codeword would need to be rearranged using the column permutations outlined in equation 33

It is obvious from table 4 that no change in complexity was achieved. Due to the number of permutations of the matrices, the optimum permutation is still unknown in research. Even if a random permutation tried during this investigation proved to decrease complexity, it would do so by only a small amount. Generally speaking, $\approx 20\%$ reduction in complexity is achievable.

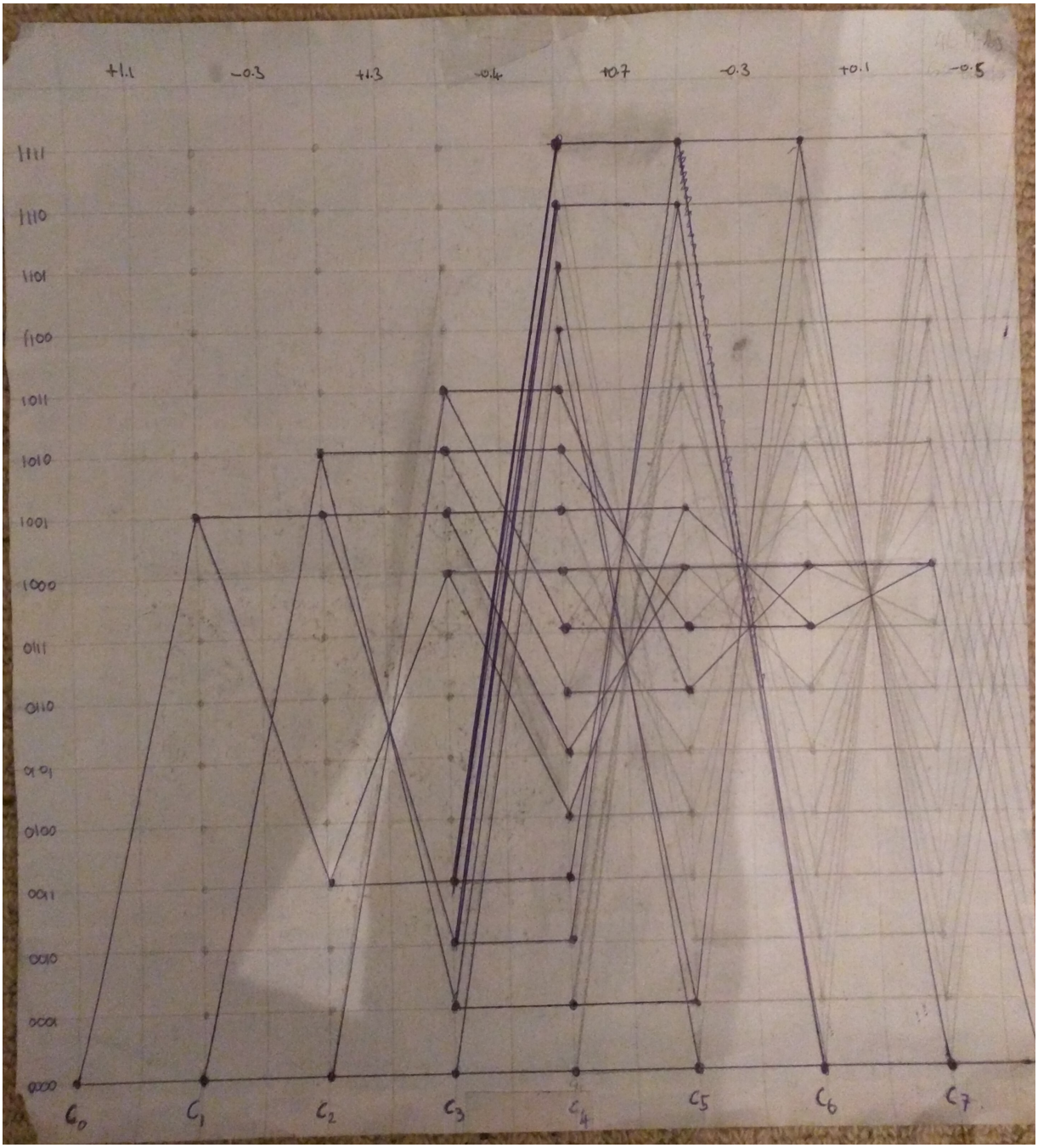
0.6 Conclusion

In conclusion, it has been shown that the Viterbi decoder combines the benefits of soft decision decoders (such as the codeword enumeration approach) with the reduced complexity of decoders such as the syndrome decoder. It has shown how it is advantageous to use the **H** matrix over the **G** matrix when the trellis width is much less than the number of information bits - but the **G** matrix method of generating a trellis is far more efficient when the trellis width is much greater; due to the lack of redundant calculations. Examples of building both the **G** and **H** matrix have been shown, and then a received vector decoded. It was proven that even with 2 bit errors, the Viterbi decoder could successfully decode a message. Finally, it was discussed how permuting the parity check and generator matrix could lead to a reduction in complexity - although this was not seen in practise due to limited experiments. If the reader is interested in reading more about permuted matrices for minimal trellis designs, then the author would suggest [3] and [2].

Appendices

Appendix A

Parity Check Trellis



Bibliography

[1] David J.C. MacKay. *Information Theory, Inference, and Learning Algorithms*. 1995. URL: http://www.inference.phy.cam.ac.uk/itprnn/book.pdf (visited on 01/12/2017).

[2] H. H. Manoukian and B. Honary. "BCJR trellis construction for binary linear block codes". In: *IEE Proceedings - Communications* 144.6 (1997), pp. 367–371. ISSN: 1350-2425. DOI: 10.1049/ip-com:19971611.

[3] X. H. Peng and P. G. Farrell. "Efficient permutation criterion for obtaining minimal trellis of a block code". In: *Electronics Letters* 32.11 (1996), pp. 983–984. ISSN: 0013-5194. DOI: 10.1049/el:19960632.

[4] Wikiepedia. *Hamming Code*. 1995. URL: https://en.wikipedia.org/wiki/Hamming_code (visited on 01/12/2017).