Randomised Construction and Dynamic Decoding of LDPC Codes

by

Joakim Grahl Knudsen

Thesis submitted in partial fulfillment of the requirements for the degree of *Master of Science*.

University of Bergen
Department of Informatics

November 30, 2005, updated 14:46

Preface

This thesis is the result of my work as a Master student at the University of Bergen, Department of Informatics.

I would like to thank my supervisor Matthew G. Parker for his excellent guidance and help with my thesis, and for always having time to discuss new ideas and details which occurred underway.

Also, I must thank my fellow students for their help and support towards submitting this thesis; Tom F. Danielsen, Sondre Rönjum, Martin Arver and Raymond Hilseth. Special thanks and love to Marthe and Blanco for their patience and understanding while I have been hardly ever home. Finally, I would like to thank my family, who have been behind me from the beginning, and encouraging me to keep going.

Bergen, November 30, 2005,

Joakim Grahl Knudsen

Contents

1 Introduction **10**

2 Objects **12**
2.1 $GF(2)$–Binary Fields 12
2.2 Matrices 12
2.2.1 Various Definitions 12
2.3 Linear Codes 13
2.3.1 Standard Forms 13
2.4 The Graph Structure 14
2.4.1 Adjacency Matrix 15
2.4.2 Bipartite 15
2.4.3 Directed 15
2.4.4 Girth 15
2.4.5 Pivot 15
2.5 Factor Graphs 17
2.5.1 Background 17
2.5.2 The Decoding Problem 17
2.5.3 Extrinsic Principle 18
2.6 Distributed Work 18
2.7 Forward Error Correction 19
2.7.1 Minimum Distance 19

3 Channel Models **20**
3.1 Channel Types 20
3.1.1 Modulation 20
3.2 Noise 21
3.2.1 Discrete Modulated Channel 21
3.2.2 Additive White Gaussian Noise Channel 22
3.2.3 Shannon's Noisy Channel Theorem 23
3.3 Modelling and Simulation 23
3.3.1 Bandwidth Expansion 23
3.3.2 Generating Gaussian Noise 24

4 Constructing LDPC Codes **28**
4.1 Random Constructions 28
4.1.1 What Code is H? 29
4.1.2 Equivalence of Random Codes 30
4.1.3 Gallager Codes 31

4.1.4 Ensembles of Codes . . . 31
4.1.5 Random, (N) . . . 32
4.1.6 Regular, (N, γ, ρ) . . . 32
4.1.7 Irregular, $(N, \gamma(x), \rho(x))$ . . . 33
4.1.8 Density Evolution . . . 36
4.2 Structured Constructions . . . 37
4.3 Cycles and Girth . . . 37
4.3.1 Are Cycles Harmful? . . . 37
4.4 Randomized Construction Algorithms . . . 38
4.4.1 Gallager's Pseudorandom Procedure . . . 39
4.4.2 Lin and Costello . . . 40
4.4.3 Complete Acyclic . . . 41
4.5 Bit-Filling . . . 42
4.5.1 Heuristics: Adding Variables . . . 44
4.5.2 Maximising Rate . . . 45
4.5.3 Maximising Girth . . . 45
4.5.4 Look-ahead; the sets U_j and $\mathcal{N}_c$ . . . 46
4.5.5 Relaxing Girth . . . 47
4.6 Extending the Bit-Filling Algorithm . . . 49
4.6.1 Improvement 1: Relaxing Girth . . . 49
4.6.2 Improvement 2: Updating U . . . 51
4.6.3 Extension 1: Local Girth Detection . . . 53
4.6.4 Extension 2: jumpBack . . . 54
4.7 Results . . . 56
4.7.1 Maximizing Rate . . . 56
4.7.2 Maximizing Girth . . . 57

5 Encoding **59**
5.1 Matrix Encoding . . . 59
5.1.1 Decoding in Standard-Form . . . 60
5.1.2 Appending I_m to H . . . 61
5.1.3 Standard LDPC Encoding . . . 63
5.1.4 Efficient Encoding . . . 64

6 Sum-Product Decoding **66**
6.1 Maximum Likelihood Decoding . . . 66
6.2 Distributed Decoding on Factor Graphs . . . 67
6.2.1 Syndrome Decoding . . . 67
6.2.2 Cyclic Factor Graphs . . . 68
6.3 Sum-Product Algorithm . . . 69
6.3.1 Iterative Decoding . . . 69
6.3.2 Initialization: Demodulation . . . 70
6.3.3 Messages . . . 71
6.3.4 Function Types . . . 71
6.3.5 Generalized Update Rule . . . 73
6.3.6 Optimized SPA for Decoding . . . 75
6.3.7 Likelihood Ratios . . . 76
6.3.8 Scheduling . . . 77
6.3.9 Stopping Criterion . . . 78
6.3.10 Comments . . . 78

7 Simulations and Results **80**
7.1 Components . . . 80
7.1.1 Channel . . . 80
7.1.2 Receiver . . . 81
7.2 Bit-Error Rate Simulations . . . 81
7.2.1 Uncoded Transmissions . . . 82
7.2.2 Word-Error Rate . . . 83
7.3 Characteristic Data . . . 83
7.3.1 Error Floors . . . 84
7.4 Simulations . . . 86
7.4.1 Flooding Schedule . . . 86

8 Experimental Decoding **88**
8.1 Feedback, Short Cycles . . . 88
8.2 Detecting Cycles . . . 89
8.2.1 Using the SPA . . . 89
8.2.2 Flooding Scheduling . . . 91
8.2.3 Implicit Feedback . . . 91
8.2.4 Practical Comments . . . 91
8.3 Avoiding Cycles . . . 92
8.3.1 Delaying 4-Cycles . . . 92
8.4 Dynamic Decoding . . . 93
8.4.1 Rotating H using Pivot . . . 94
8.4.2 Breaking Oscillation . . . 95
8.4.3 Consequences of Pivoting . . . 95
8.4.4 Maintaining Sparsity . . . 96
8.4.5 Protecting Soft Information . . . 97
8.5 Alternative Scheduling . . . 99
8.5.1 Thresholding . . . 99
8.6 Hybrid Decoding . . . 100
8.7 Comments . . . 101
8.7.1 Unfinished Results . . . 102

9 Concluding Remarks **104**
9.1 Open Problems . . . 104
9.1.1 Ant Traversal Decoding . . . 104
9.1.2 Avoid Going Round Cycles . . . 104
9.1.3 Strong Subcodes . . . 104
9.1.4 Graph-Based Encoding . . . 105

A Approximated Discrete Log **106**

B Tools **108**
B.1 1: Augmented EBF . . . 108
B.1.1 Shortcuts . . . 109
B.1.2 No Optimisation . . . 109
B.2 2: Code Library . . . 109
B.3 4: SPA Decoder . . . 110
B.4 11: Channel Simulator . . . 110
B.5 21: Check Girth . . . 111

B.6 22: Draw Graph 111
B.7 Etcetera 111
B.7.1 Convert Maple - Alist 111

Bibliography **112**

List of Tables

4.1 The BFT is extended to also keep track of *where* the girth-bound was relaxed, such that we may resume construction from any position $v_{i'}$. 55

4.2 Maximising rate using EBF, compared to results of MacKay. Columns labelled 'I' and 'II' are from [1], while our results are in the rightmost subtable, starting with the column 'III.' 56

4.3 Maximising girth using EBF, again compared to [2] with our results in the two rightmost subtables. Column 'IV' is the results of using the extensions suggested in this thesis. 57

6.1 Θ_4, $p = 4, o = 1$, truth table τ_{XOR}, and an example calculation of $\mu_{u \to v_3}$. 72

A.1 Θ_{10} (abridged), $p = 10, o = 5$, truth table τ_{DL}. 107

List of Figures

2.1 Pivoting on edge (u, v) of the bipartite graph. Removed edges are dotted, and created (new) edges are solid, black lines. Edges that are not part of the operation, are colored gray. Note how v becomes the systematic edge. 16
2.2 The factorization of (2.8), in FG form. 17
2.3 The Factor Graph representation of the code defined by the Parity-Check matrix of (??). 18

3.1 Binary Phase-Shift-Keying Modulation. 21
3.2 The Discrete Memoryless Channel is a probabilistic mapping from b-ary inputs to q-ary outputs, and is completely specified by the transition probabilities. 22
3.3 Simulating AWGN noise; note how distribution (shape) depends not only on SNR, but also on coderate. Also, the figure illustrates the *offsets* corresponding to means $\mu = \pm E_s = 1$. 25
3.4 Uniform distribution: $n = 10^5$ random samples of `lrand48()`, over the interval $[\mu - 4\sigma, \mu + 4\sigma]$. 26
3.5 Approximations of the Normal Distribution. 27

4.1 Comparison of 7 random $[250, 125]$ codes from the same $(250, 3, 6)$-ensemble. The performance is almost identical, as expected. . . . 30
4.2 The evolution of the convergence in decoding a small, irregular LDPC code, at SNR 6dB (y-axis shows Bit-Error probability). . 35
4.3 The irregular LDPC code shows gain at high SNR, due to lowered flooring effect from reduced word-error rate–Fig. 4.3(b). 36
4.4 Two small cycles; the 'butterfly' 4-cycle (in bold), and the 'bow-tie' 6-cycle. 38
4.5 Comparison of the same ensemble, varying over increasing girth. Note the expected gain in avoiding 4-cycles, and, conversely, the similarity of $g = 6$ and 8. 39
4.6 For $m = 7$ and $\gamma = 3$, $\iota = 3$ bits are connected while $\mathcal{G}$ is still acyclic. 42
4.7 The sets U and $\mathcal{N}$ after connecting $c^\star$ to v_i; $\mathcal{N}_{c_0} = \{c_1, c_2\}$, $\mathcal{N}_{c_1} = \{c_0, c_2, c_4\}$, $\mathcal{N}_{c_2} = \{c_0\}$, $\mathcal{N}_{c_3} = \{c_1, c_4, c^\star\}$, $\mathcal{N}_{c_4} = \{c_1, c_3\}$, $\mathcal{N}_{c^\star} = \{c_3\}$, and $\mathcal{N}_{c_6} = \emptyset$. 48
4.8 All $m = 7$ checks infeasible; $F = \emptyset$. 48
4.9 After connecting c_2, the regular updating of U handles any re-ordering of subsets; note the grey checks have been "moved down" to their correct subsets. 51

5.1 Systematic versus non-systematic. 62

6.1 The noisy channel symbol from the input bit is adjusted by the bias of the local constraint nodes. Hence, the tentative decoding is contained in bit nodes, and its *protection* is proportional to the size of its support, $|n(v)|$. 68
6.2 Equivalent Factor Graph representations of the XOR_4 function. The double-circled node is an auxiliary 'state-node,' containing only the end result of the chaining. 75

7.1 Comparison of simulated uncoded BER, and theoretical uncoded BER according to (7.3). To gather sufficient data, we simulated 5×10^3 transmissions over the interval $[0, 4\rangle$; 5×10^4 over $[4, 7\rangle$; and 10^6 over $[7, 10\rangle$. 82
7.2 The average number of "decoder iterations" is independent of timeout, and only weakly dependent on N [3]. 85
7.3 Increased precision (no flooring) as `max` is increased. 86
7.4 Our simulation software validated against the results of MacKay. 87

8.1 Girth Monitor on a small LDPC code. Within 4 flooding iterations, all bits have determined their effective girth (which, in this case, equals local girth). Age fields of messages are not shown. . 90
8.2 An example showing the iteration updating c_j. 92
8.3 The simplified graph, $\mathcal{G}^\star$, suitable for pivot. 94
8.4 Density of 48×96 LDPC Code (MacKay) over 1000 random pivots. The code is $(3, 6)$-regular, which gives $\Delta_0 = 3/48 = 1/16$. 95
8.5 By restricting the application of pivot, we are able to control the increase in density, while still rotating the rowspace of H. The code is the same as in Fig. 8.4. 96
8.6 Hybrid Scheduling, which consists of regular Flooding iterations, interspersed with one pivot operation (with probability p). 100
8.7 Hybrid Scheduling, but with 'avoid4' scheduling instead of Flooding. 101
8.8 Dynamic decoding, using pivot. 102
8.9 An unexplained gain at low SNR. 103

Chapter 1

Introduction

Low-Density Parity-Check (LDPC) codes were originally invented by Gallager [4] in his 1963 thesis. These asymptotically optimum codes were among the first results to verify Shannon's 'Noisy Channel Coding Theorem' [5] from 1948, which claimed that, for any rate $R = k/N < C$ (channel *capacity*), there exist a random code that is can achieve arbitrarily low decoding error. However, as predicted by Shannon, Gallager's codes were extremely large in order to achieve this optimum, and, as such, ahead of their time, mainly in terms of what was technologically possible at the time, but also what was actually needed. Current throughput needs were well satisfied by the conventional, short blocklength Reed-Solomon Codes, developed just prior to LDPC.

Along with the explosive demand for bulk data transmissions, came the discovery of Turbo codes in 1993 [6], almost three decades later. This sparked renewed interest in capacity-approaching codes, and Gallager's findings were not completely forgotten [7, 8, 9]. In fact, variations on LDPC design quicky caught up with (and surpassed, [10, 11]) Turbo codes, pushing the record even closer towards the Shannon Limit [12].

The results of Shannon are well established, and it does not appear likely that one may exceed the Shannon Limit. Hence, the code approaching capacity most closely will definitely become the choice for next-generation communications standards, ranging from mobile (IEEE802.16) to long-haul optical communication and broadcasting [13, 14]. For instance, in 2004, LDPC became the new standard for satellite broadcasted, high-definition TV (HDTV),[1] replacing the 10-year-old previous standard. Also, in LAN and Internet protocols, where entire packets of information are lost underway to receiver, the long blocklength of LDPC codes[2] is well-defined for encoding redundant messages at the packet-level (as opposed to bit-level). The receiver may then *restore* a certain number of missing packets locally, significantly reducing latency of online audio and video streams–as predicted by Luby *et al.* in 1997 [10].

In this thesis we will explore the basic concepts regarding LDPC codes; how they are constructed, encoded, and–in particular–how they are decoded. By operating on the equivalent graph-representation of the sparse code, the Sum-Product Algorithm approximates optimum decoding at a complexity that

[1] Digital Video Broadcasting-2 and IEEE802.3an.
[2] Several hundreds MBytes are common [15].

is linear in blocklength. Pieces of information flow independently through the graph, where they are subject to an extremely simple, generalised update rule. Based on local decisions, individual nodes determine when to 'fire,' and in which direction. The success of the process is measured in terms of *convergence,* which–to a large extent–depends on avoiding certain 'bad topologies' in the graph.

The aim of this thesis is to explore such topologies, and their effects on the performance of LDPC codes. Conventionally, codes constructed specifically to minimise the occurrence of such problems in decoding. In addition to this, we suggest the converse approach of modifying the decoding rules such that information may simply *avoid* such topologies alltogether.

This should be of interest not only as it simplifies the construction requirements, but more importantly as it illustrates how to achieve reliable communications in a *dynamic* network, in which the optimum code may change structurally from transmission to transmission.

The structure of the thesis is as follows. **Chapter 2** presents a brief overview of the terminology and concepts used in the thesis. **Chapter 3** describes the most important channel models, and how these may be modelled to within a satisfactory accuracy in a computer simulation. The problem of constructing good, optimised LDPC codes is the topic of **Chapter 4**, with an emphasis on *randomised* constructions–following the lead of [4]. A specific construction algorithm, the 'Extended Bit-Filling' algorithm [1], is discussed in detail. The remainder of the chapter is devoted to our improvements and extensions to the algorithm (or, 'scheme'), as well as some performance results. **Chapter 5** is a brief overview of the encoding problem, which until recently[3] has been the major bottleneck reducing the application of LDPC codes. **Chapter 6** contains a somewhat unorthodox description of the well-known Sum-Product Algorithm, illustrating its decomposability into one, unified (function-independent) update rule. **Chapter 7** begins with a look at the details of simulating the Bit-Error Rate performance of a communications system (code, channel, and decoder). **Chapter 8** takes this one step further, and begins looking at some of the novel SPA schedules mentioned above. What local decisions can be made at the receiver end (i.e., decoder) to improve convergence; and, thus, code performance? Finally, the thesis is completed with a short summary, followed by some appendices describing our software (enclosed CD).

[3]Encoding via sparse matrix operations has pushed the complexity down to match the linear decoding algorithm.

Chapter 2

Objects

When using computers to model and simulate real-world systems, there will often exist no simple internal ordering between the components. Such relations might otherwise be used in the design of highly streamlined, perhaps distributed algorithms, using, say, mathematical formulas to solve the problem efficiently. Often, such simulations are *realized* by matrices storing datasets, which, then, are manipulated through basic linear algebra.

In the more arbitrary situations, where internal relations are unstructured, or even truly random, general purpose data structures–such as graphs–are intuitively very helpful. By linking dependent components, complex systems of unevenly sized components are naturally expressed as a graph.

2.1 $GF(2)$–Binary Fields

Without going into detail on field arithmetics, we mention that the finite field over which we will define our codes, is the *binary* (Galois) field, $GF(2)$, consisting of elements $\{0, 1\}$.

2.2 Matrices

A matrix is defined as a compact container (rectangular array) for storing "the essential information of a linear system" [16]. With this construction follows the vast terminology and the transformations defined in linear algebra. In the following, we will review some of the most important definitions and notation used in this thesis.

2.2.1 Various Definitions

The size of a matrix, denoted by $a \times b$, describes its height (in *rows*), and width (in *columns*), respectively–and always in that order. In this thesis, we are most interested in the row vectors, $\vec{r}_j \in GF(2)^N$. A set of non-zero vectors $\mathcal{B} = \{\vec{r}_0, \vec{r}_1, \ldots, \vec{r}_{k-1}\}$ is linearly dependent if the sum (in $GF(2)$) of these $|\mathcal{B}| = k$ vectors is 0 (otherwise, they are lin. independent). Such a set of independent vectors is called a *basis* since they span out a *vector space, V*. The *dimension* of this space is determined by the size of its basis; $\dim(V) = |\mathcal{B}| = k$; such that

"any set in V containing more than k vectors must be linearly dependent" [16]. The space consists of all linear combinations of the vectors in the basis, giving a total of $|V| = 2^k$ vectors–always including the all-zero vector.

The *rank* of a matrix is the dimension of its row or column space. The *weight* of a vector is defined as the total number of non-zero (i.e., '1') entries. The *density* of a matrix is the average weight distribution taken over all row vectors in the matrix. A matrix is called *sparse* if its density is less than 0.5, and *very sparse,* if the density remains constant while $N \to \infty$. [8].

2.3 Linear Codes

The vector space V_N, spanned out by some basis consisting of independent vectors of *length* N, is a linear code. This space is usually denoted as a $[N, k]$-code, $\mathcal{C}$, where $k = \dim(\mathcal{C})$. The 2^k vectors in $\mathcal{C}$ are called *codewords,* and the fact that $\mathcal{C} \subset GF(2)^N$–that there exist vectors that are *not* codewords–is the necessary and sufficient condition for error-detection (and, a certain amount of error correction). This redundancy, $m = N - k$, determines the *code rate,* $R = k/N$, a measure on how much *information* is sent per codeword.

The definition of a *linear* binary code, is that the sum of any subset of codewords always equals some codeword (commutative). The *dual code,* $\mathcal{C}^\perp$, is a $[N, N-k]$ linear code, which is called the *null space* of $\mathcal{C}$, owing to the fact that the cross-product of two orthogonal vectors (i.e., some codeword of $\mathcal{C}$, and some of $\mathcal{C}^\perp$) is 0.[1] This leads to the most fundamental fact in working with linear codes;

$$GH^T = \vec{0} \mod 2. \tag{2.1}$$

The (Hamming) *distance* between two codewords is defined as the number of positions in which they differ (can be easily calculated as the weight of the sum of the vectors, modulo 2). Since all codewords in a linear code are equally likely, we measure the *minimum distance,* $\mathrm{d_{min}}$, as the weight of the minimum weight codeword. Often, this is included in the code definition; $[N, k, \mathrm{d_{min}}]$.

In the theory of linear codes, it is common to represent vectors as *columns* vectors. We will follow this convention in this thesis.

2.3.1 Standard Forms

Usually, codes are constructed (via the generator matrix, or polynomial) to have specific rate, $R = k/N \leq 1$, where k denotes the amount of *information* sent per block. Hence, the error-protection, or *redundancy,* are the remaining $m = N - k$ bits. From (2.1), we see that $k = \mathrm{rank}(G)$, and $m = rank(H) \geq k$. Constricted to N bits per block, there is a tradeoff between rate and protection; increasing k means sending more information at a time–at a higher rate–but with less protection, m. Conversely, we may design a code with low rate, say $R = 1/4$, which means that only one fourth of each block is information, the rest redundancy. From this, we see that $m \geq k$.

To simplify the above mentioned relationship between H and G (2.1), conventions exist on the internal order of bits (i.e., columns) so that it is trivial

[1] Their internal angle is 90°.

to calculate the one matrix from the other. The $k \times N$ *standard-form* of the generator matrix is

$$G' = [\, I_k \,|\, P \,], \tag{2.2}$$

where I_k is the $k \times k$ identity matrix, and P is a random, $k \times m$ matrix corresponding to the $m = N - k$ redundant (protection) bits. In the following, we will use the notation M' to denote that a matrix, M, is in its standard-form. Due to the identity part, we see that (2.2) defines a *systematic code,* in which the k information bits are transmitted in 'raw form.' Also, the identity part ensures that G' has *full rank* (all k rows are linearly independent–see Ch. 2).

Any k-bit information vector, $\mathbf{s}$, may be encoded to an N-bit *codeword,* $\mathbf{x}$, by multiplication with G'

$$\mathbf{x} = \mathbf{s}(G')^T = G'\mathbf{s}^T, \tag{2.3}$$

which is of complexity $\mathcal{O}(N^2)$. More efficient encoding schemes for sparse codes exist, which is discussed in Ch. 5. Any row of G'–as well as any linear combination of rows in G'–is a codeword. In both cases, we see that the set of codewords (the vectorspace) spanned out by G' (i.e., the *code,* $\mathcal{C}$), has dimension k, and consists of 2^k codewords.

The Parity Check matrix, H, is defined to span out the null space of $\mathcal{C}$, such that all codewords may be identified by their common *syndrome* (checksum), $\vec{0}$ (2.1);

$$\mathcal{C} = \{\, \mathbf{x} \in GF(2)^N \,|\, \mathbf{x}H^T = \vec{0} \,\}. \tag{2.4}$$

Since $|\mathcal{C}| = 2^k < 2^N = |GF(2)^N|$, we may use this fact to detect (and correct, as discussed in Ch. 6) N-bit vectors that are *not* valid codewords, typically due to channel noise. Using (2.2), we define the $(N - k) \times N$ (standard-form) H' as

$$H' = [\, P^T \,|\, I_{N-k} \,], \tag{2.5}$$

where P^T is the transpose of the random part of G'. It is quite possible to switch the internal ordering of P and I parts, as long as H' and G' are 'opposite' in that they null each other out.[2]

2.4 The Graph Structure

In essence, the graph, $\mathcal{G}$, consists of a set of vertices, $\mathcal{V}$, and a set of edges, $\mathcal{E}$. We will denote the number of vertices as $|\mathcal{V}| = N$. Edges can be one- or two-way, making for a *directed* or *undirected* graph, respectively. In this sterile form, the graph only expresses the structure of the problem at hand. By superposing a particular instance of the problem onto these objects, this structure allows the use of simple, yet powerful graph-based algorithms to solve the original problem. Consider, for instance, the classical problem of finding the shortest path between two cities on a map containing N cities. Here, the graph

[2] Often, standard-form is defined by $G' = [\, P \,|\, I_k \,]$, e.g. [17]. This is only a matter of convention, in that the receiver must know *where* in the codeword the information is stored.

setting is very intuitive, where cities are represented by vertices, and roads by edges. By superposing the particular instance (city names, road distances, and perhaps even road traffic indicators etc.), Dijkstra devised an $\mathcal{O}(N)$ algorithm for evaluating all possible routes between two cities, and returning the optimum route comprising the fewest number of edges. Some graph-theoretical definitions will be useful, and are introduced briefly in the remainder of this section.

2.4.1 Adjacency Matrix

The Adjacency Matrix, $\mathcal{A}$, of a graph with N nodes is a symmetric $N \times N$ matrix, in which position (j, i) indicates that node j is connected to node i by exactly $a_{j,i} \geq 0$ edges. Also, nodes can (normally) not be connected to themselves, which ensures that $a_{i,i} = 0, \forall i$.

2.4.2 Bipartite

A graph is called bipartite iff $\mathcal{V}$ can be partitioned into two subsets, a and b, in such a way that all edges have one vertex in a and the other vertex in b; i.e., no edges connect vertices within the same subset. This is called an $(|a|, |b|)$-bipartite graph, and means that $\mathcal{A}$ is non-zero only in the submatrices, H and H^T, which map nodes in a to nodes in b;

$$\mathcal{A} = \begin{bmatrix} \mathbf{0} & H \\ H^T & \mathbf{0} \end{bmatrix}.$$

Hence, bipartite graphs are compactly represented by the $|a| \times |b|$ H-part of $\mathcal{A}$. A bipartite graph is said to be (γ, ρ)-*regular* if each node in a has degree γ; and each node in b has degree ρ.

2.4.3 Directed

In a directed graph, an edge is a one-way connection between two nodes. An undirected graph is easily modelled by a directed graph, by doubling up every edge.

2.4.4 Girth

Any set of edges connecting two (not necessarily different) vertices, u, v, is called a *walk*. If no vertex is visited more than once, the walk is called a *path*. A *cycle* is defined as a path where $v = u$. Since $\mathcal{G}$ may contain several cycles of various lengths, the length of the shortest cycle is refered to as the *girth*, $G(H)$, of the graph (where, in this notation, H is the adjacency matrix).

2.4.5 Pivot

Later in this thesis, we will investigate the operation of LDPC on dynamically updated graphs. The update will use an operation called Pivot, which is most commonly defined on a matrix, and is the main engine of Gaussian Reduction. By pivoting on a non-zero entry (j, i) in the matrix, the result is that row j is added (modulo 2) to all other rows $j' \neq j$ iff $(j', i) \neq 0$. This has the effect of clearing all non-zero entries in column i, except for position j which has become

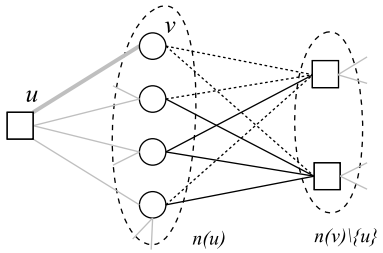


Figure 2.1: Pivoting on edge (u, v) of the bipartite graph. Removed edges are dotted, and created (new) edges are solid, black lines. Edges that are not part of the operation, are colored gray. Note how v becomes the systematic edge.

the *pivot* of column i. A weight 1 column that is non-zero in position j is refered to as the j^{th} *identity vector*, $\vec{e}_j$.

In [18], the action of pivot on an edge (u, v) of a graph is defined as a transformation which takes $\mathcal{G}$ to an equivalent, yet structurally different, graph, $\mathcal{G}'$. The resulting set of unique graphs, obtained by repeating the operation, is called the "pivot orbit," whose size is an important parameter of the graph (see [19] and [20]).

The *local neighbourhood*, $n(u)$, of a node u is defined as the nodes adjacent (r eachable via one edge) to u. In a bipartite graph, all nodes in $n(u)$ must per definition be in the opposite partition to u. If (u, v) is an edge in $\mathcal{G}$, then the nodes in $n(u)$ and $n(v)$ must also be in opposite partitions as well. Obviously, $n(u)$ and $n(v)$ can not have any nodes in common; the *overlap*, $\mathcal{O}_{u,v}$, (of nodes) is zero. As described in [18], pivot on a bipartite graph can then be carried out by simply complementing the set of edges between $n(u)$ and $n(v) \setminus \{u\}$. Fig. 2.1 shows an example.

Where column j is transformed to the identity vector, the corresponding node; v, is *disconnected* from the graph, save for the pivot edge, (u, v). Hence, the ordering of nodes is important, as it does not make sense for the row (check node) to become systematic. We define the graph-based pivot operation as an operation on an edge from a check to a bit node. This is the explanation for the *skewed* bipartition, in which u is excluded from $n(v)$, while v is included in $n(u)$. Save from the pivot edge (u, v), the column node, v, is then completely disconnected from the graph–as defined by the identity vector, $\vec{e}_j$, in the matrix description.

Define the overlap, $\mathcal{O}^E_{u,v}$, *of edges* between $n(u)$ and $n(v)\setminus\{u\}$ as the number of edges connecting the two local neighbourhoods. To be precise, we count the number of "direct links" across the local bipartition. In Fig. 2.1, these correspond to the 4 dotted lines. Then, we know that the number of edges removed (and created) by pivoting on edge (u, v), is

$$\mathcal{E}^\dagger = \mathcal{O}^E_{u,v}, \qquad \text{and,} \tag{2.6}$$

$$\mathcal{E}^\star = (|n(u)|)(|n(v)| - 1) - \mathcal{E}^\dagger, \tag{2.7}$$

respectively. The pivot operation has complexity $\mathcal{O}(|n(u)||n(v)|) = \mathcal{O}(\gamma\rho)$.

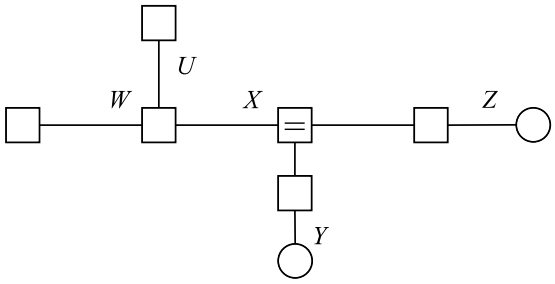


Figure 2.2: The factorization of (2.8), in FG form.

2.5 Factor Graphs

Factor Graphs (FG) are a specific class of graphs, which have long been used to model systems across a wide range of sciences. By exploiting the reducibility of the problem at hand, the "FG strategy" is to separately solve the resulting chain of subproblems. Following an example of [21]; the function

$$p(u, w, x, y, z) = p(u)p(w)p(x|u, w)p(y|x)p(z|x), \tag{2.8}$$

can be solved more efficiently by using the factorization provided on the right-hand side, since answers to subproblems can be reused. For instance, this is achieved by message-passing on the corresponding FG (Fig. 2.2), which encodes the factorization of (2.8) in its structure. Two vertices are linked via edges iff dependencies exist between the components they represent. Several articles give in-depth introductions on FG, in particular [22, 21], and we will focus mainly on their application to the decoding problem.

2.5.1 Background

As reviewed in [22], FG's are "generalization[s] of the 'Tanner Graphs' of Wiberg *et al.*," that Tanner used to model the internal system of the iterated LDPC decoder. In general, FG is a versatile tool for solving many problems, by having edges represent *variables*, and vertices represent *functions*. The dependencies mentioned above are expressed as an edge being connected to a vertex iff the corresponding variable (edge) is in the domain of the corresponding function (vertex). This variable/function dependency is a two-way relationship where both objects use the information of the other, so FG's rely on undirected edges.

By identifying the particular network of functions and variables, systems ranging from artificial intelligence networks (belief propagation), statistics, and filtering problems can be modelled by an FG. As discussed in Ch. 6, this allows for the common solution by the surprisingly simple Sum-Product Algorithm.

2.5.2 The Decoding Problem

In the context of LDPC decoding, the system is modelled by a network of simple linear functions; the binary XOR[3], and the "equality function[4]" By viewing the

[3]Addition in GF(2), s.t. $\mathrm{XOR}(1, 1) = 1 \oplus 1 = 0$.

[4]Conventionally, these are refered to as "variable nodes", although there is no particular need for such a distinction between vertices, as will be discussed in Ch. 6.

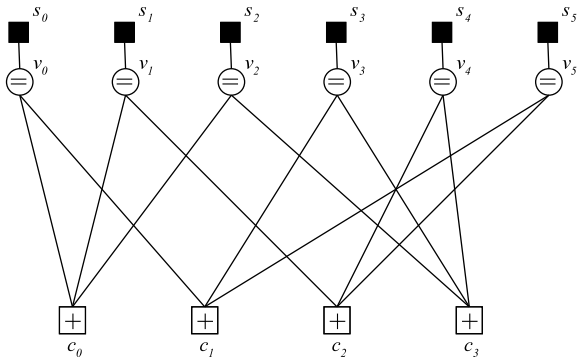


Figure 2.3: The Factor Graph representation of the code defined by the Parity-Check matrix of (??).

$m \times N$ Parity-Check matrix, H, as an adjacency matrix of a bipartite graph, each of the m check-equations can be modelled by XOR-functions, while each of the N bits of $\mathcal{C}$ are modelled by equality functions. In the following, we will refer to these function vertices as simply *checks* and *bits,* respectively.

This system encodes a non-unique[5] factorization of the linear code's *characteristic function*, $\chi_{\mathcal{C}}$, which indicates whether the global constraint $H\mathbf{v}^T = 0$ is satisfied (valid codeword). Continuing the example of (??), we have the following factorization, which is visible from the check nodes of Fig. 2.3

$$\begin{aligned}\chi_{\mathcal{C}}(v_0, v_1, \ldots, v_5) &= [(v_0, v_1, \ldots, v_5) \in \mathcal{C}] \\ &= [v_0 \oplus v_1 \oplus v_2 = 0]\,[v_0 \oplus v_3 \oplus v_5 = 0] \\ &\quad [v_1 \oplus v_4 \oplus v_5 = 0]\,[v_2 \oplus v_3 \oplus v_4 = 0]. \end{aligned} \tag{2.9}$$

2.5.3 Extrinsic Principle

In working with the convergence of Belief Propagation algorithms, such as the Sum-Product Algorithm (Ch. 6), an important principle is to minimise the occurrence of self-regulating processes; or *feedback*. Extrinsic information is understood as information that is collected exclusively from *other parts* of the system. By attempting to keep all calculations extrinsic, the system is more capable of repairing errors.

2.6 Distributed Work

Most research on LDPC codes, and error-correction software in general, works explicitly in a linear algebra setting. Many convenient algorithms for code design, encoding, and decoding are implemented using matrices and vectors as data structures. In this project, we have decided on maintaining a distributed approach, thinking more in terms of hardware implementations where the graph-based Factor Graph implementation is a natural environment. For instance,

[5]Note that the Parity-Check matrix of a linear code represents an immediate factorization of $\chi_{\mathcal{C}}$ into m XOR-functions, which are also highly reducible (see Ch. 6).

in Ch. 6, we investigate the design of a distributed decoder running the Sum-Product algorithm using local operations on nodes. As such, information flowing through the FG is represented as *messages* stored in Edge objects.

Matrices representing LDPC codes are sparse and, as such, a matrix-oriented algorithm needs to search through the length N (m) row (column) vectors to locate, and action on, the non-zero positions. Typically, for (software) SPA implementations, there are separate $m \times N$ matrices for storing the Parity Check ("adjacency") matrix, as well as the input and output SPA floating point soft messages [17, 23]. Again, the sparsity and large blocklength, makes this a quite wasteful design, both in terms of memory (space) and complexity (time).

The added memory requirements of actually constructing the graph objects, is alleviated by the increased efficiency of each vertex having direct (constant time) access to its adjacent input-objects. Furthermore, since Vertex and Edge objects essentially consist only of the fields corresponding to the matrices described above, the memory usage is reduced. This gives an increase of both speed and memory.

Working in a distributed environment has many interesting real-life implications, which may present novel applications for LDPC coding.

2.7 Forward Error Correction

From the perspective of the receiver, there are several ways of combating the disturbances caused by channel noise. In short, coded transmissions permit two countermeasures; error detection, and correction. Basic Coding Theory shows that by adding redundancy to the transmissions, it is possible for the receiver to detect the presence of error by, basically, comparing versions of the same message. For instance, if each message is repeated three times in succession, error is determined when there are discrepancies among the received versions (which should otherwise be identical). However, this redundancy does not provide information on the *location* of error, and the only recourse is asking for retransmission. Such Automatic Repeat Request (ARQ) schemes require a two-way channel, which is not always a feasible option, for instance in long-haul, deep space transmissions.

Using mathematical relationships, it is possible to apply the redundancy more efficiently such that the receiver may infer also the positions of error. This Forward Error Correction alleviates the need for a two-way channel, trusting the receiver with the responsibility of ensuring reliable communications.

2.7.1 Minimum Distance

The minimum distance of a code is defined as the minimum number of bits that differ between any pair of codewords. This count is a fundamental property of linear codes, and reveals the capability of the code; detecting $s \leq \mathrm{d}_{\min}(\mathcal{C}) - 1$ errors, and correcting $t \leq (\mathrm{d}_{\min}(\mathcal{C}) - 1)/2$ errors (see [24] for proof).

By using the redundancy to increase the distance between codewords, the error correction capabilities of the code are improved.

Chapter 3

Channel Models

The primary purpose of Error Correcting Codes is the ability to counteract the inevitable presence of interference, or *noise*, during transmission. Such noise comes in many different shapes, and amount to a major challenge in any communications scenario.

For simulation purposes – when we want to predict certain features of code constructions – we must acknowledge the fact that simulations, and idealized channels, will not provide accurate information on how the code will perform in real life situations. Following Gallager's warning; "such insight should be used with caution [4]."

3.1 Channel Types

Wikipedia defines a channel as "the medium through which information is transmitted from a sender (or transmitter) to a receiver[25]." There is a wide variety of systems that satisfy this definition, where some may be less obvious than others. All systems involving a cable, or similar physical link across some distance, immediately come to mind. Also, a multitude of wireless channels exist, ranging across the entire low wavelength-end of the electromagentic spectrum[26]. Lending from [27], typical examples include "twisted-pair telephone wires, shielded cable-TV wire, fiber-optic cable, deep-space radio, terrestrial radio, and indoor radio." However, channels are also found discretely integrated within the plastic hoods of CD/DVD units, hard-drives of computers; largely, any system where data is read or written to a storage medium.

3.1.1 Modulation

Although digital channels exist, where bits are transmitted directly in their quantized form, e.g. pulses of light through an optical cable, most channels in use today are analog. Bits of the digital source must be converted, or *modulated*, to distinguishable peaks of energy, i.e. waveforms of a specific duration T, prior to transmission. At the receiving end, these waveforms can be demodulated back to bits, by sampling the stream in intervals of length T.

Even without coding, modulation by itself provides some amount of protection against transmission errors. By ensuring that the different bits are mapped

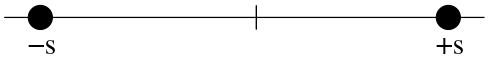


Figure 3.1: Binary Phase-Shift-Keying Modulation.

to waveforms of maximum difference,[1] we have the best chance of still being able to tell the distorted waveforms apart at the other end. In the binary input case, we map bits 1 and 0 to waveforms of opposite phase, i.e. shifted by π (180°)

$$\begin{aligned} s_1(t) &= \sqrt{\frac{2E_s}{T}}\cos(2\pi f_0 t) \\ s_2(t) &= \sqrt{\frac{2E_s}{T}}\cos(2\pi f_0 t + \pi) = -s_1(t), \end{aligned} \tag{3.1}$$

where the carrier frequency f_0 is a multiple of $1/T$, and E_s is the energy of each transmitted signal, or *channel symbol*. As is obvious from Fig. 3.1, the two signals in the BPSK constellation are maximally separated. To further increase the difference, we have to 'scale' the entire constellation, thereby pulling the symbols further apart. However, this comes at an obvious cost, namely, increased energy usage per symbol, E_s.

3.2 Noise

One common problem with any type of channel is the inevitable presence of noise, affecting sections or individual elements of the stream (bits).

These disturbances are caused by many different sources. Some are due to natural conditions affecting the link, such as electrostatic energy from lightning affecting a copper wire, or the intereference of hard weather, or solar flares in deep space on wireless transmissions. In addition, links are often part of a dense network, where magnetic fields create 'cross-talk' across adjacent streams.

At the receiving end, the stream of waveforms is demodulated back to bits, but, in practice, their shapes will be altered by noise. Depending on the amplitude, or strength, of the original signal, and the amount of noise, some number of waveforms will always be demodulated to the wrong bit. Unless carefully handled, such channel errors would render any channel useless at all but very high signal power levels, where errors are less frequent.

3.2.1 Discrete Modulated Channel

The Discrete Memoryless Channel (DMC) is a digital channel, in which bits are transmitted directly, without modulation. Such channels are also called Discrete-Input, Discrete-Output channels. In other words, there is no soft information available, and we must perform hard decisions during decoding.

Any DMC is completely described by a set of *bq transition probabilities*, $\mathcal{P}$, which are the probabilities of each b-ary source bit being mapped to each q-ary target bit.

[1] By treating symbols as ary coordinates in a q-dimensional space, we may apply geometrical distance measures to optimise such mappings–or *signal constellations* [17].

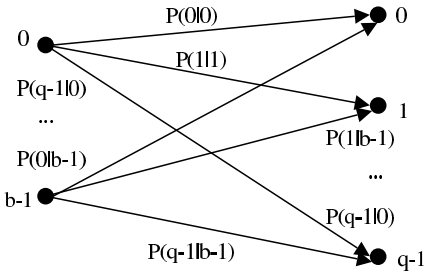


Figure 3.2: The Discrete Memoryless Channel is a probabilistic mapping from b-ary inputs to q-ary outputs, and is completely specified by the transition probabilities.

Since this is a discrete channel, only *bit-crossover* errors are possible, where a bit is received at a different value than defined by its intended mapping. In other words, there are no "gray areas" between symbols of the q-ary output alphabet. However, there is some amount of soft information to use in decoding. For each received symbol, c, we know it must have originated from exactly one of the b possible source symbols, such that

$$P(c|0) + P(c|1) + \cdots + P(c|b-1) = 1. \tag{3.2}$$

Each symbol, c, comes with an implicit probability distribution, which depends on the respective transition probabilities. This set of probabilities completely defines the DMC.

Since the channel is memoryless, there are no statistical dependencies among the individual symbols. What happens to any given symbol, is completely unaffected by what may have happened to the previously transmitted symbols.

3.2.2 Additive White Gaussian Noise Channel

In most real-life situations, we are working with analog channels, and unquantized channel symbols. The required modulation amounts to an increased complexity of the system. However, the soft values contain more information about the a priori source symbols they represent, which can be used to increase code performance by several decibel.

Consider the Binary Symmetrical Channel (BSC) which is a DMC with $b = q = 2$. This is a completely quantized channel, where the only soft information is the reliability measure provided by knowing the channel transition (error) probability, $p = P(0|1) = P(1|0) = 1 - P(0|0) = 1 - P(1|1)$. By increasing the range of output symbols, q, we increase the amount of soft information available to the receiver.

The Additive White Gaussian Noise (AWGN) channel is a Binary-Input, Unquantized-Output channel that can be viewed as the result of extending the output range to the real numbers, $q = \infty$. This channel subjects source symbols, s_i, to noise in the form of random peaks of energy, which are successively *added* onto each transmitted symbol. The amount of noise at any time instant can be

described by a random, normally distributed (i.e., *White Gaussian*) variable, n_i, such that channel symbols become

$$c_i = s_i \oplus n_i.$$

The randomness of the Gaussian noise has a one-sided power spectral density (PSD) N_0, which depends on the "noise level," or variance, σ^2

$$N_0 = 2\sigma^2. \tag{3.3}$$

3.2.3 Shannon's Noisy Channel Theorem

Shannon's theorem [5] shows that each channel has a *capacity,* C, and that for any rate $R < C$, there exist codes of rate R that can achieve arbitrarily low decoding error, $P(e)$, when decoded using Maximum-Likelihood decoding (MLD). LDPC codes are based on this theorem, which also requires that blocklength, N, must be allowed to be sufficiently large, such that [17]

$$P(e) \leq 2^{-NE_b(R)} \rightarrow 0,$$

for fixed R. The (positive) function $E_b(R)$ is determined by the channel characteristics.

This is the basis for the asymptotically optimum performance of LDPC codes using Sum-Product Algorithm decoding, which approximates MLD (see Ch. 6).

3.3 Modelling and Simulation

Error correcting codes can be compared based on their most important parameter; their ability to handle channel noise. By simulating a channel model, in software or hardware, a plot is made of the average *Bit-Error Rate* (BER) as the *Signal-to-Noise Ratio* is incremented from low to high. Any code can only correct a certain amount of bit errors. By simulating the transmission and decoding of codewords, the BER at the current SNR can be calculated. To achieve a sufficient degree of statistical confidence in the BER, the simulation must be repeated, and the averaged BER reported. The plot is then generated by calculating the average BER at the simulated SNR intervals.

3.3.1 Bandwidth Expansion

When modelling channel noise, it is very important to be aware of what parameters that have the highest effect on the outcome, and how to adjust these. These factors are not always obvious.

In real transmissions, an output from the channel encoder is produced every T seconds, i.e. the *transmission rate* is

$$\mathcal{R} = R/T = k/NT \ [bits/sec.]. \tag{3.4}$$

A reliable channel should have a *bandwidth* of

$$W \geq 1/2T \ [Hz]. \tag{3.5}$$

In the uncoded case, code rate is $R = k/N = 1$, and every single channel symbol represents one source symbol. In other words, we are transmitting at maximum transmission rate, $\mathcal{R} = 1/T$, and with minimum (none) error protection, $N - k = 0$. Given (3.4) and (3.5), we note that the uncoded transmission rate,

$$\mathcal{R}_{uncoded} = 1/T \leq 2W, \tag{3.6}$$

is only limited by available bandwith, W.

In the coded case, where $N - k > 0$ and $R < 1$, we have some protection against noise, which comes at the cost of reduced efficiency, $\mathcal{R}$. Per definition, we always output one channel symbol per T seconds, but the added redundancy at the source requires the use of several channel symbols to represent *one* source symbol. Again observing (3.4) and (3.5), we find that $\mathcal{R}_{coded}$ is limited by an additional factor; namely, code rate $R < 1$:

$$\mathcal{R}_{coded} = R/T \leq 2RW. \tag{3.7}$$

To produce accurate simulations for coded transmissions, we need to expand the bandwidth of the (virtual) channel by a factor of R^{-1} to maintain constant transmission rate, when compared to the uncoded case. If the energy per channel symbol is E_s, and we are using a code of rate $R = k/N$, we define the energy per information-bit, or *source symbol*, as

$$E_b = \frac{E_s}{R}, \quad R \leq 1. \tag{3.8}$$

This distinction between channel and source energy is important when producing accurate simulations.

In a coded transmission, the probability of error can be expressed as the ratio of source symbol energy to noise PSD. Taking the above observations into account, (3.8) and (3.3) give the Signal to Noise Ratio (SNR)

$$\frac{E_b}{N_0} = \frac{E_s}{RN_0} = \frac{E_s}{2R\sigma^2}. \tag{3.9}$$

3.3.2 Generating Gaussian Noise

Soft decoding (Ch. 6) of LDPC codes makes the AWGN channel an obvious choice of simulation environment in which to evaluate code error performance. The unquantized, modulated channel symbols can easily be demodulated and mapped to *a posteriori* bit-value probabilities (APP's), which are fed directly into the SPA decoder.

To sample White Gaussian noise in a computer simulation, we need a source of random numbers with a *normal distribution*. As defined in [28], "the Gaussian function is the probability function of the normal distribution," which is expressed as

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp^{-(x-\mu)^2/2\sigma^2}, \tag{3.10}$$

with mean $\mu = \pm\sqrt{E_s} = \pm 1$. The shape of the distribution; i.e. the range over which the output samples are spread, is regulated by the variance, σ^2. As

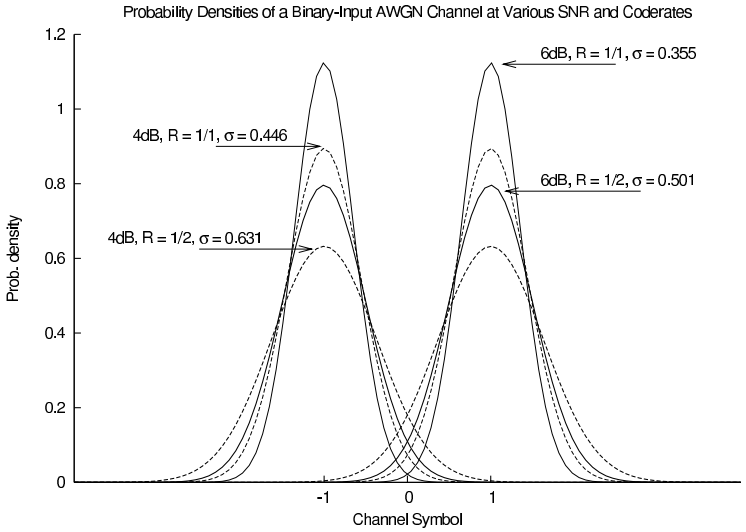


Figure 3.3: Simulating AWGN noise; note how distribution (shape) depends not only on SNR, but also on coderate. Also, the figure illustrates the *offsets* corresponding to means $\mu = \pm E_s = 1$.

Algorithm 1 The Box-Muller Algorithm [29].

Pairs of uniform random numbers serve as a basis for creating two Rayleigh-distributed variables, as part of a uniform distribution. By using a simple transformation function, these are converted to one normally distributed random number.

indicated by Fig. 3.3, the variance is narrowed by increasing SNR; increasing rate, R; or both.

The standard software methods, like `rand()` and `lrand48()` in `C++`, are designed to output a *uniform distribution* of pseudorandom[2] numbers, but simple algorithms exist that alter the distribution without corrupting the original degree of randomness.

The Box-Muller algorithm effectively produces normally distributed numbers at a constant complexity. Yet, as pointed out in [30], regular patterns appear in the output, due to the "continuous and differentiable mapping" by the trigonometrical functions. The same article concludes that the distribution can be improved by "using a dicontinuous transformation mapping" scheme, such as Von Neumann's rejection method.

Given (3.9), we can calculate the noise variance, σ^2, corresponding to the SNR we want to simulate

$$\sigma^2 = (2R\frac{E_b}{N_0})^{-1}, \tag{3.11}$$

and use one of the methods described to sample the corresponding normal distribution, typically with mean $\mu = 0$.

[2]Actual randomness is very difficult to reproduce in software, but clever mathematical algorithms can approximate randomness to a satisfying degree.

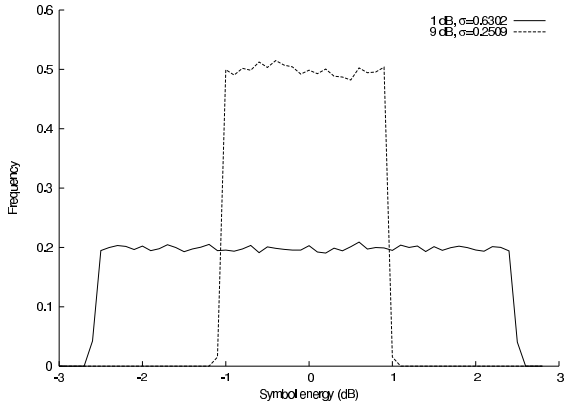


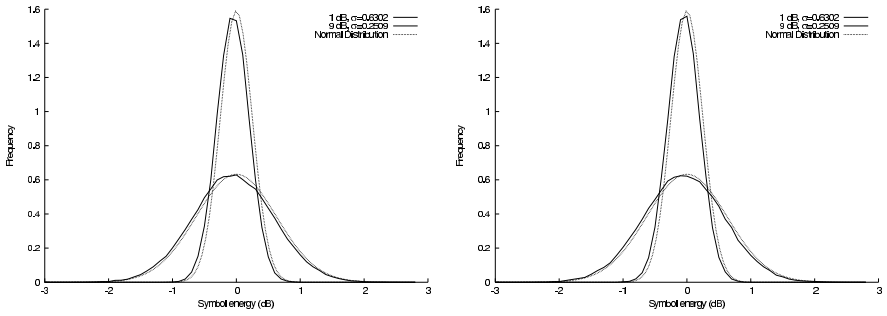
Figure 3.4: Uniform distribution: $n = 10^5$ random samples of `lrand48()`, over the interval $[\mu - 4\sigma, \mu + 4\sigma]$.

Algorithm 2 Von Neumann's Rejection Method.

The uniform distribution can be transformed to any shape, as defined by an arbitrary density function, $h(x)$. A uniform, random point, x, within the domain of $h(x)$ is selected, along with a random "weight", y, selected within the range of $h(x)$. Since (3.10) is a PDF, the domain can be clipped to $[\mu - 4\sigma, \mu + 4\sigma]$ with a confidence of $> 99\%$ (ref. Fig 3.4), and the range is $[0, f(\mu)]$.
The point x is returned as the normal-distributed value if and only if it satisfies the constraint, $y < h(x)$. The method is repeated until a valid *(point,weight)*-pair is picked.

As indicated by Fig. 3.5, for our purposes, both methods approximate the normal distribution with satisfactory accuracy. However, for our simulations, we are using the slightly superior method of Von Neumann.

In the following chapters, we will conveniently refer to processes as "random," while understanding the actual, pseudorandom processes involved.



(a) Box-Muller

(b) Von Neumann

Figure 3.5: Approximations of the Normal Distribution.

Chapter 4

Constructing LDPC Codes

LDPC codes have received a great deal of attention since the rediscovery (ref. Ch. 1) of their outstanding potential. Along with the increase in computer processing power, the once considered impractical codes have long since proved to be quite usable in modern transmission systems. Already in 1997, Luby *et al.* considered LDPC codes to be "extremely useful for applications such as real-time audio and video transmission over the Internet" [10].

Gallager's original description [4] described a quite specific structure which was based on a random construction of the Parity-Check matrix, H. In his work, which was ahead of its time both in terms of computational resources and performance, he defined a type of LDPC code which can be viewed as the foundation for current work in the area.

In this chapter, we will explore typical considerations involved in constructing optimal, or *capacity approaching* codes, along with a selection of construction algorithms. As discussed in Ch. 1, the (linear) codes are completely described by H. As observed by Gallager, this gives a practical approach to the construction problem, in which H is typically filled with non-zero entries according to some (random) scheme.

As with all probabilistic methods, such random construction schemes often fail; and the codes produced, although near-optimal[1], may not be very usable. For practical application of LDPC codes, deterministic, or *structured*[2], methods are more valuable [35, 36]. Although we will mainly adopt the guidelines of Gallager, we will also review some of the important structured constructions.

4.1 Random Constructions

In 1948 Shannon [5] introduced the concept of using random codes to achieve arbitrarily good Forward-Error Correcting (FEC) codes. With this, an upper bound on the possible throughput of any channel–the Shannon limit–became the common goal of research. Shannon's proofs were not supported by any practical results, and were generally considered only theoretically interesting for several

[1] Chung *et al.* were in 2001 able to approach the Shannon limit with a margin of merely 0.0045 dB [12]. Similar results are frequently reported, [31, 32, 1, 33] etc.

[2] Such methods are in some cases refered to as *explicit* [9, 34].

decades. In fact, the general assumption was that a suboptimum rate, R_0, was a more realistic channel bound [32, 31].

The findings appeared to be realizable by Gallager's 1963 LDPC codes, however, due to the large blocklengths, these were mainly of theoretical interest at the time. With the improvements in technology, these ideas are no longer infeasible. Also, unlike the recent Turbo Codes, restrictive patents have conveniently expired, making the technology publich domain. Due to practical approximations of Maximum Likelihood Decoding (e.g., Belief Propagation, and the Sum-Product Algorithm; See Ch. 6), the seemingly insurmountable problem of transmitting close to capacity has effectively been solved.

However, most of the excellent results in LDPC research are based on the prerequisite that blocklengths are allowed to grow arbitrarily large. In fact, the results of both Shannon and Gallager are based on random codes that are only *asymptotically* very good – as N goes to ∞. This is also pointed out in more recent results [31, 32]. The random nature of the constructions also implies that any code will be rigorously fixed to its particular blocklength. This has very real consequences in practical scenarios. For instance, in packet-based protocols employing variable blocklengths, such as TCP/IP, one would require multiple code definitions, consuming a "significant [amount] of non-volatile memory storage [35]." Reckognizing these practical issues, we will try to maintain a practical focus while working with random LDPC constructions.

4.1.1 What Code is H?

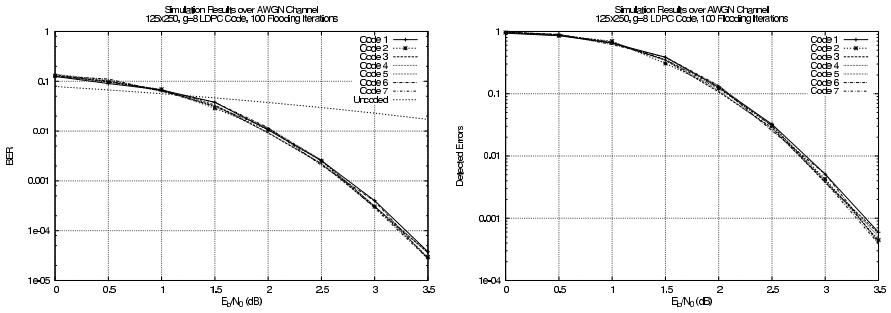
In Ch. 2, we see how linear codes are well defined by a Generator matrix, which spans out the space of codewords (the codespace), $\mathcal{C}$. When working with LDPC codes, it is common to focus on the random $m \times N$ Parity-Check matrix, H, which only implicitly defines the code by spanning out the null space (i.e., the dual code, $\mathcal{C}^{\perp}$). In a construction setting, this approach is somewhat backwards. For instance, we typically do not know the precise rank of the resulting construction, $\text{rank}(H) = N - k \leq m$, which defines the dimension $\dim(G) = k$ of $\mathcal{C}$. This is a very characteristic parameter of a linear $[N, k]$ code.

Before we proceed further, we briefly repeat the conventions on standard matrix forms (indicated with a tick; e.g. M') that we adhere to in this thesis (see Ch. 2). The k systematic bits of G' are placed at the beginning of codewords, such that $G' = [\, I_k \,|\, P \,]$. The Parity-Check matrix spans out the null space of the code, which requires the ordering $H' = [\, P^T \,|\, I_{N-k} \,]$. This gives us the fundamental relationship (2.1), $GH^T = \vec{0} \mod 2$.

When we construct codes using a randomized scheme, we initially assume that the resulting matrix H will have full rank, $m = N - k$. Recalling code rate, $R = k/N < 1$, the *design rate* is defined [11] as,

$$R' = (N - \text{rank}(H))/N = 1 - \gamma/\rho \leq R, \tag{4.1}$$

as the desired code rate. If $\text{rank}(H) < m = N - k$, then we have constructed a *different code*. If $k' = N - \text{rank}(H) < k$, then this is a $[N, k']$-code, in other words a code of slightly *higher* rate than intended [32]. The rank of random codes should always be checked (using Gaussian Reduction) as part of the construction process.



(a) BER performance

(b) Detected Errors

Figure 4.1: Comparison of 7 random $[250, 125]$ codes from the same $(250, 3, 6)$-ensemble. The performance is almost identical, as expected.

Say we need a $[900, 300]$ ($R' = 1/3$) code. We would then aim for constructing a random 600×900 H-matrix, of $\text{rank}(H) = m = 600$ linearly independent rows, such that the resulting G'-matrix (recall that $'$ means standard form) may have a $k \times k$ identity part (2.2). Say H has only $\text{rank}(H) = 500$ linearly independent rows. The corresponding code is, then, an $[900, 400]$-code, of rate $R = 4/9$.

Although there exist *structured* LDPC construction schemes that focus on achieving full rank; e.g., based on cyclic shifts, the observation of this thesis is that most random schemes rely on a probabilistic argument. The sparsity (low density, $\Delta_H = \mathcal{O}(N), \rho \ll N$) of LDPC codes implies that the likelihood of producing two (or several) random rows that are linearly dependent, is low (as N grows large); especially when the column-weight, γ, is odd [32]. Hence, if faced with underfull rank, we may simply retry the construction, using a different seed.

In passing, we make the observation that certain codes may benefit from redundancy.

4.1.2 Equivalence of Random Codes

Random constructions can be grouped into *ensembles* [4], by observing what parameters are fixed, and serve as the common framework of the LDPC codes. An intuitive ordering of ensembles would be in terms of their respective codes' common ability to correct errors, which, as will be discussed, can be readily averaged by examining any, randomly selected member of an ensemble.

Most construction schemes for random codes are based on randomly[3] filling H with non-zero elements from $GF(q)$, within the boundaries of the framework. In other words, frameworks restrain the construction process, as features that would deteriorate performance can easily be avoided.

From a graphical perspective, the edges are connected between the two sets of nodes in the bipartite graph. Given a set of parameters (framework), there

[3] As discussed in Ch. 3 we always deal with pseudorandom processes, but write 'random,' for brevity.

exist many valid random constructions.[4] We say *similar* to emphasize the fact that codes within an ensemble are not *equivalent* in the sense that they span out different null spaces. The concept of grouping random codes is based on the fact that these codes exhibit (asymptotically) similar performance, despite their different codespaces. Hence, any code selected at random will give valid and characteristic information on the general performance of the codes in the same ensemble. This allows us to discuss optimization schemes by focusing only on design parameters (the framework of the ensemble), without going into detail of the bit-level structure (i.e., edges) of specific codes. Fig. 4.1 shows an example.

The basic parameters of a framework are blocklength, N, and code rate, $R = k/N < 1$; variable, or *bit*, node degree[5] γ; and check node degree, ρ. Note that the redundancy, m, is implicitly defined by the code rate;

$$m = N - k = N - RN = N(1 - R), \qquad R < 1. \tag{4.2}$$

4.1.3 Gallager Codes

As a simple example, consider the 'Gallager codes' [4]. Here, every column vector of H has exactly γ non-zero entries (weight γ), and every row vector has exactly weight ρ. These frameworks are denoted by (N, γ, ρ). In Gallager's notation, this reads (N, j, k).

Picture creating *sockets* on all nodes, into which edges can be plugged; γ sockets per bit node, and ρ per check node. Codes of this framework correspond to some particular permutation of the

$$|\mathcal{E}| = N\gamma = m\rho \tag{4.3}$$

edges, such that each bit node is connected to γ check nodes, and each check node is connected to ρ bit nodes. Other frameworks that optimise different parameters for want of better codes, will be discussed further on in this chapter.

4.1.4 Ensembles of Codes

Gallager's work was not on randomly chosen codes as these would most certainly contain harmful, short cycles, but rather with "explicit graphs [...] to which his analysis does apply" [33]. Recent work by Richardson *et al.* [11, Concentration statement, p. 600] has proved the assumption that the codes in an ensemble are equivalent. This confirms the advantage in working with ensembles; that the average performance of the entire ensemble is well approximated by the performance of (almost) any constituent code. This has valuable practical implications when evaluating Bit-Error Performance (BER) in Ch. 7. Even though two codes, or instances of the same ensemble, can be quite different in terms of the actual connections between nodes, we may still expect equal performance due to this proof of equivalence–as we saw in Fig. 4.1.

MacKay [32] presents a fine-grained partitioning of important random LDPC ensembles (Def. 1), ordered by what is assumed to be decreasing probability of decoding error. These six ensembles serve as illustative examples as we briefly

[4]The size of an ensemble (the number of valid codes), depends on the stringency of the framework. Some are more easily satisfied than others, resulting in a larger space of codes.

[5]Initially, node degrees are considered constant. In the following, we will see how these are generalised to weight degree sequences.

Definition 1 MacKay's *Ensembles of Very Sparse Matrices* [32].

1. Matrix H generated by starting from an all-zero matrix and randomly flipping γ not necessarily distinct bits in each column.
2. Matrix H generated by randomly creating weight γ columns.
3. Matrix H generated with weight γ per column and (as near as possible) uniform weight per row.
4. Matrix H generated with weight γ per column and uniform weight per row, and no columns having overlap greater than 1 (meaning, 'no 4-cycles').
5. Matrix H further constrained so that its bipartite graph has large girth (meaning, 'girth $g > 6$').
6. Matrix $H = [\,C_1 \,|\, C_2\,]$ further constrained or slightly modified so that C_2 is an invertible $m \times m$ matrix (see Ch. 5).

discuss the main types of random constructions, below. For each class of ensembles, we discuss the corresponding framework. In all cases, the redundancy (number of rows in H), m, is defined implicitly by the desired code rate, as in (4.2).

4.1.5 Random, (N)

The random ensemble consists of codes that are defined by a very minimum of parameters; blocklength N. By only requiring that the codes be sparse (low-density), H may be populated by arbitrary, low-weight column vectors. This means that we should expect quite poor performance, especially due to the expected high frequency of 4-cycles [33, 32]. As seen in Ch. 6, this affects the number of independent decoder iterations, deteriorating BER performance; see also Fig. 4.5.

Ensemble 1 from Def. 1 defines the random ensemble, in which an $m \times N = N(1-R) \times N$ matrix is filled with $\mathcal{O}(\gamma N)$ non-zero bits. MacKay adds the constraint of requiring that all column vectors have equal weight, γ. This does not ensure any weight distribution across the row vectors.

4.1.6 Regular, (N, γ, ρ)

Returning to the work of Gallager [4], we find the origins of the added constraint of explicitly fixing the weight of the column (γ) and row (ρ) vectors of H. The construction of regular LDPC codes–often simply called Gallager codes–can be accomplished by adding weight-γ columns to H in such a way that the total weight of any row will equal ρ. In order to obtain codes which would conform to his mathematical analysis, Gallager furhter ensured that no two columns would have overlap of more than one position; i.e. no 4-cycles [4].

As will be seen, strict regularity in both γ and ρ can be quite difficult to achieve. Construction is often simplified by requiring only that the weight along one dimension be fixed, while the other be upper bounded [1]. If the weight along the 'bounded dimension' (typically, ρ_{max} for the row vectors) is

uniformly distributed, then such *semi-regular* constructions can be quite close approximations to regular codes.

The early results indicating the excellent performance of LDPC codes (at least in theory at the time), were due to strictly regular codes. It is established [10, 32, 11] that the best regular LDPC codes are from the $(N, 3, 6)$ ensembles. These results were already encountered in [4], where column weights larger or equal to 3 were found to give "minimum distance that increases linearly with the block length for j and k [i.e., γ and ρ] constant." A general intuition is to keep the column weight high (and odd), while simultaneously minimizing the row weight [33]. Typical iterative LDPC decoding algorithms, as explored in Ch. 6, are based on local decisions where a node will try to determine its correct state by using only extrinsic information.[6] The decoder will attempt to rectify the states of bit nodes, such that it may output correct overall state–a valid codeword–that will most likely[7] equal the true, originally transmitted codeword, $\mathbf{x}$. The amount of extrinsic information available to any bit, its *support*, should therefore be maximized to increase the chance of bits converging to their correct state (see Fig. 6.1). For the check nodes, the converse is true. The check nodes also depend exclusively on extrinsic information to determine whether they are *satisfied* or not. Since their input is solely from bit nodes, which often are in error, it becomes clear that the best decisions are made if check nodes have fewer potentially confusing inputs. As the number of edges, $|E|$ (4.3), is constant, this becomes a problem of balancing conflicting requirements.

In retrospect, MacKay reckons that the somewhat simple, regular Gallager codes (1963) "would have broken practical coding records up until 1993" [32]. Ensembles 2 through 6 are examples of regular LDPC codes. The careful distribution of weight in the third ensemble is a typical semi-regular code, quite similar to the constructions we will focus on in the following sections.

4.1.7 Irregular, $(N, \gamma(x), \rho(x))$

In the wake of Turbo codes, research has been made into the gain of allowing irregular weight distributions in H. Originally conceived by Luby *et al.* in [10],[8] the main idea would be to optimize code performance by carefully adjusting the vector of bit and check node degrees, $\gamma(x)$ and $\rho(x)$ respectively.[9] As discussed above, very good codes depend on finding a fair tradeoff between maximised bit degree, and minimised check degree. By increasing the support of some bits, one can expect quick convergence with high precision at these positions; which, in turn, help the lesser supported bits to stabilize. Returning to [37], such strong bits are called "elite bits."

[6] As defined in Ch. 2, extrinsic information is gathered exclusively from *other parts* of the system, such that the node is not biased by its current (possible erraneous) value.

[7] As discussed furhter in Ch. 6, decoders may produce a *different,* yet valid, codeword. The frequency of such *undetected errors* is proportional to density; i.e, low.

[8] Actually, the results in this paper rely on the concatenation of a *cascade* of irregular codes, followed by a final convolutional code, in order to achieve capacity-approaching performance.

[9] Luby denotes column (bit) weight sequences by $\lambda(x)$.

The degree sequences are conveniently expressed using polynomials [10],

$$\gamma(x) = \sum_i \gamma_i x^{i-1}$$
$$\rho(x) = \sum_i \rho_i x^{i-1}$$

where the coefficient γ_i (ρ_i) gives the *fraction* of the bit (check) nodes[10] of degree i.

Precise theory in the area has proven difficult to find, and studies in [33] and [11, 31] rely on time-consuming computer searches to find the best ensembles of codes. Typically, one profile is given as input (say, $\gamma(x)$), from which a good "partner vector" ($\rho(x)$) can be found. Also, many cases simplify the search by fixing the distribution along one dimension; typically using constant row-weight, ρ. Interesting online resources for optimized degree distributions are found at [38, 39].

By analysing the asymptotic performance ($N \to \infty$) of a simple, iterated decoder, peformance characteristics were revealed for both regular and irregular codes. This analysis produced a differential equation expressing the convergence of the decoder. From this, an important mathematical condition was derived [10];

$$\rho[1 - \delta\gamma(x)] > 1 - x, \qquad x \in [\,0, 1\,) \tag{4.4}$$

indicating the maximum tolerable noise level,[11] δ, given some degree distributions $\gamma(x)$ and $\rho(x)$. Two important results were proved in this groundbreaking work. By observing that the best regular codes (the $(3, 6)$-ensemble) would violate (4.4) at noise levels "far from the optimal value [δ]", it was possible to prove the sub-optimality of regular codes.[12] Aided by this condition, specific constructions were found that would satisfy (4.4). Given some $\gamma(x)$ and channel threshold δ, a computer search for a check degree sequence, $\rho(x)$, that would satisfy (4.4) could be performed. This proved that irregular codes–if carefully constructed–would transmit at rates arbitrarily close to channel capacity. Similarly to the inherent gain of utilizing available soft information to increase decoder performance, partitioning the node degrees into a profile gives more flexibility in finding the optimum degree distribution.

Fig. 4.2 shows the convergence of a small, irregular LDPC code. The degree sequence, $\gamma(x)$, is not optimised in any way, yet suffices to illustrate the relationship between convergence and support; that elite bits converge fast, and may then 'assist' weaker bits. The row-weights are upper-bounded by $\rho_{\max} = 8$, and is quite regular. Notice the significant drop in BER of the 'elite bits' (degrees 5 and 7) already after the first iteration ($t = 2$). As expected, the convergence seems to propagate through the graph as a wave; where the higher connected (protected) bits aid–or, trigger–the convergence of weaker bits. The BER performance of the code is shown in Fig. 4.3.

[10] The notation where γ_i is the coefficient of x^{i-1} is due to the extrinsic principle of SPA, in which nodes exclude one input in producing (extrinsic) output. This leads to compact notation.

[11] NB!! Is this the same as the channel threshold discussed in Richurbanke/MacKay??

[12] These observations were made specifically for rate 1/2 codes, but convey information on the general performance of irregular LDPC codes.

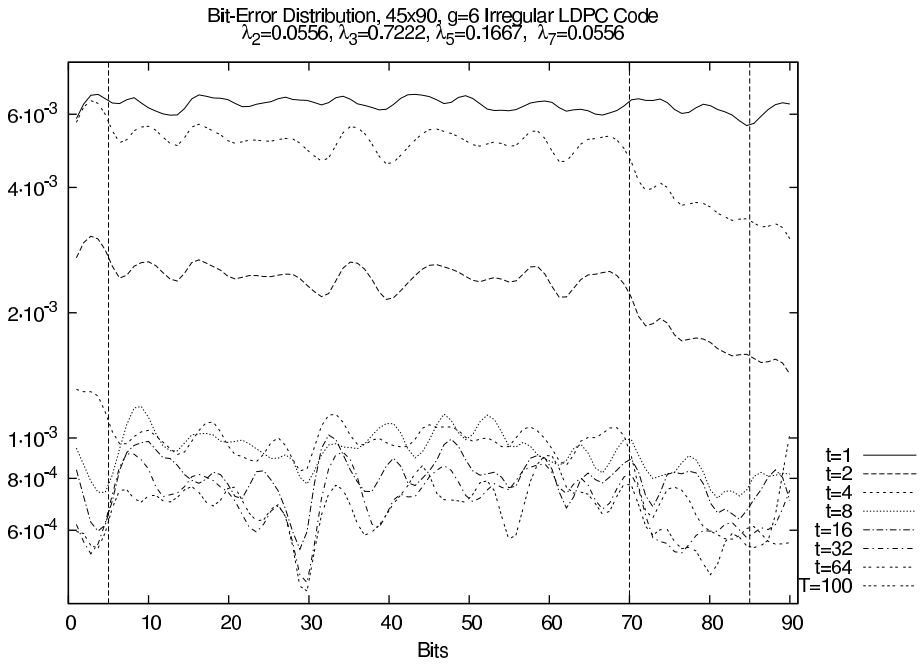


Figure 4.2: The evolution of the convergence in decoding a small, irregular LDPC code, at SNR 6dB (y-axis shows Bit-Error probability).

The bits are sorted in order of increasing degree; i.e., the first 5.56% have degree 2. Vertical seperation lines indicate the transition from one degree to the next. Each horizontal 'layer' shows a snapshot of the bit-error distribution just prior to decoder iteration t. In the following, we understand t as the point where that iteration is just about to begin; hence, $t = 1$ means that the decoding has not yet begun. The decoder *timeout*, $T = 100$, is the maximum number of iterations before declaring a (word) failure (Ch. 6).

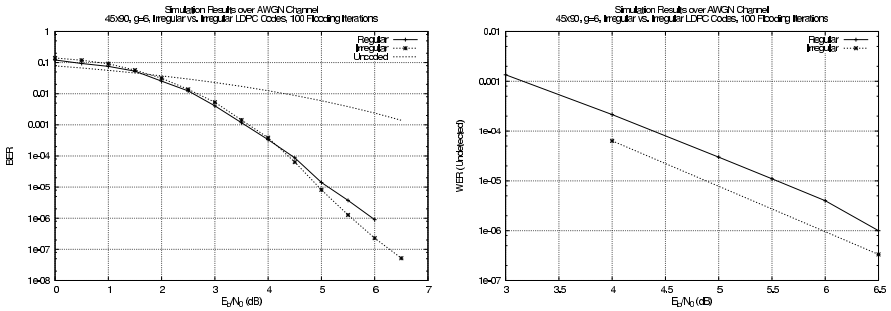
At each point t, we examine the decoder state, and count the number of times each bit, v_i, is in error.[13] Since the state changes more quickly at first, and then slows down as t grows, we observe iterations numbered by

$$t = 2^j, \quad 0 < j \leq h = \lfloor \log_2(T) \rfloor, \tag{4.5}$$

such that $t = \{1, 2, 4, 8, \ldots, 2^h\}$. We also observe the final decoder iteration, $t = T = 100$, thereby producing a $(h+1) \times N$ matrix of counters, M, of which row m_j contains the distribution for iteration t. At SNR 6dB, the majority of simulations actually converge to a valid codeword within the first 8 iterations, and these do not contribute data to the counters for $t > 8$. To produce plots of $\geq 95\%$ confidence, we must repeat the experiment S times; until all rows m_j have counted at least 100 errors.[14]

[13] This is a simulation; an artificial transmission where the receiver can compare the decoder convergence to the correct, error-free original codeword, and, this, produce diagnostics data.

[14] Assuming bit-errors are statistically independent events.



(a) BER (b) WER

Figure 4.3: The irregular LDPC code shows gain at high SNR, due to lowered flooring effect from reduced word-error rate–Fig. 4.3(b).

In summary; for iteration t, we may produce the Bit-Error Rate (BER) as

$$\mathrm{BER}_t = \frac{B_t}{NS}, \tag{4.6}$$

where B_t is the total number of bit-errors at iteration t (summed over all N bits). (4.6) is then averaged over each individual bit, v_i, by using the information in the matrix of counters;

$$\mathrm{BER}_{t,v_i} = \mathrm{BER}_t \cdot (b_i/s_t). \tag{4.7}$$

4.1.8 Density Evolution

As discussed in Chapter 6, successful and correct decoding is a question of whether the overall bit-error probability, p_i, as a function of iteration number, t, decreases towards zero. Density Evolution (DE) [31] is a technique for observing such convergence in SPA messages–at some fixed SNR level–as the decoder iterates. By observing the drop in bit-error probability (i.e., convergence) from one decoder iteration to the next (until timeout), a plot is made showing the 'evolution of density.' At certain points such plots show a tendency towards flattening out (converging) to a fixed stable point. These correspond to situations where the decoder gets stuck, and can not proceed. If this flattening occurs at a too high error-probability, then this is most likely a decoder error. However, often the flattening breaks off again, dropping towards another fixed point at a lower error-level.

Density Evolution is used to produce *Extrinsic Information Transfer* (EXIT) charts, which, essentially, consist of a plot generated using DE, which indicates the convergence of the decoder at a specific SNR. [17]. By also plotting the mirror of this curve, one may see a *tunnel* form in between the two. This indicates that the decoder is expected to converge. However, as SNR is lowered (and DE produces other curves), the curves may intersect; 'blocking' the tunnel.

This is indication that the decoder will fail to converge. The smallest SNR *not* causing failed convergence (i.e., first creating a tunnel), is called the *channel threshold,* and is a significant parameter of the decoder.

4.2 Structured Constructions

As we have seen, random constructions exist that more or less guarantee asymptotically optimum performance. While this remains a very valuable theoretical result, it presents certain disadvantages when LDPC codes are to be used in actual communication scenarios. Random constructions need to be stored explicitly in memory in order to be used for encoding or decoding. Long blocklength means very large memory usage just to store the $m \times N$ Parity Check matrix; or the $\mathcal{O}(N\gamma)$ bipartite graph. This also affects the computational efficiency of the code which, in real life, might be even more crucial than the BER performance–see Ch's. 5 and 6. An alternative approach might seem quite intuitive at this point; to use some form of structure to achieve a determinstci construction algorithm.

Analysis of structured LDPC codes is not as dependent on the grouping of codes into ensembles. Furthermore, the same generic method, given various input parameters, is able to output a range of specifically designed codes, e.g. of various blocklength [35]. The main advantages in using structure can be summarized as an increase in flexibility/adaptability, and a reduction in cost; in terms of complexity, memory usage, and transmission *latency.* The latter is due to the possibility of specifically adapting decoders to the structural pattern of the code.

4.3 Cycles and Girth

As with any random graph structure, the presence of cycles in LDPC codes is a natural, inevitable feature which is difficult to counteract. Given the bipartite nature of the graphs represented by the H-matrix, with variable nodes on one side, and check nodes on the opposite, any cycle can always be "rotated" so that it begins and ends in a check-node. Also, within the cycle, all edges must always be between two nodes of different type. In terms of the underlying H-matrix, this translates to a "zig-zagging motion" along columns and rows, changing direction at non-zero positions, in a strictly alternating manner. Due to the lack of redundant (i.e., double) edges, the shortest possible cycle is of length 4, and manifests itself in the shape of two columns having *more than one common row.* Another characteristic shape, is the 'bow-tie' shape of the 6-cycle; see Fig. 4.4.

4.3.1 Are Cycles Harmful?

LDPC codes based on cyclic graphs (girth $g < \infty$) suffer loss in error performance when decoded with the message passing algorithm (See ch. 6). Cycles introduce feedback in the flow of messages, allowing bits that are part of a cycle to stimulate themselves with their own–possibly erranoeus–state. This violates the SPA principle that messages contain only extrinsic information. Obviously, the number of independent iterations of the decoder equals the girth, g, of $\mathcal{G}$

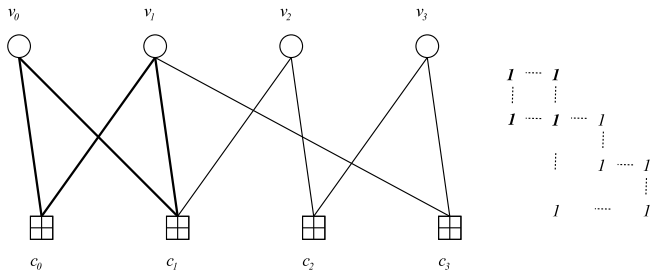


Figure 4.4: Two small cycles; the 'butterfly' 4-cycle (in bold), and the 'bow-tie' 6-cycle.

[4]. Furthermore, Tanner was able to provide proof of a definite dependency between girth and minimum distance [7].

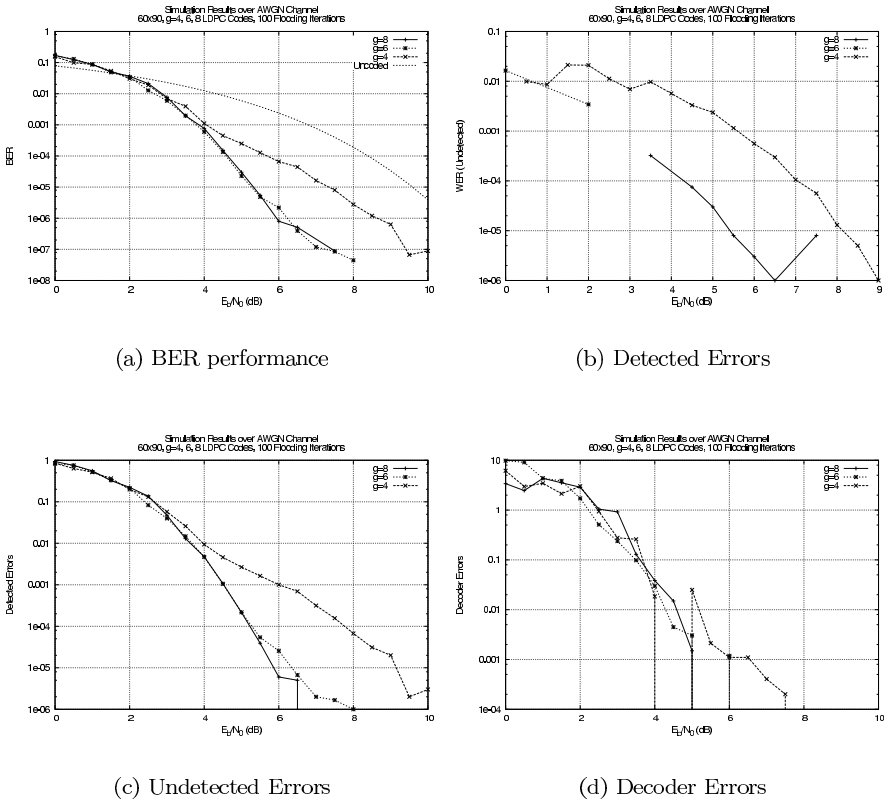
In the theory of LDPC codes, it is often understated that optimal girth (i.e., complete acyclicity) is prohibitally difficult. Furthermore, some results (e.g. [11, 32]) indicate that such violations of the *local tree assumption* might not disrupt decoding to any mentionable extent. Published work tends to rely on the convention that girth > 4 provides sufficient feedback protection. For instance, as shown in [37], such 'bad topologies' in the associated factor graph lead to unnaturally low minimum distance, giving rise to *undetected errors*– see Ch. 7. Avoiding 4-cycles is "sufficient to prevent the topology [...] from occurring," and many construction schemes produce codes that are "only" free of 4-cycles [17, 32, 4].

Simultaneously, others (e.g., Campello *et al.* [1]; which we will focus on in the following sections) are more stringent, insisting that maximising girth will generally improve code performance. This motivates construction schemes which can guarantee some minimum bound on the girth of the constructions. Some construction schemes rely on a post-processing of the graph, where cycles are identified and *removed* by deleting (or permuting) columns (or rows) of H. While this may increase overall construction efficiency, such alterations of H may easily produce unforseen side-effects – perhaps even closing new, shorter cycles.

4.4 Randomized Construction Algorithms

The previous discussions have laid the foundation for perhaps one of the most interesting areas of ongoing research within LDPC codes, namely how to actually construct codes that satisfy the desired framework (valid member of the ensemble). Regarding the near-optimum performance shown in very long codes, a valid question is whether it is possible to design smaller, more practical codes that inherit at least fractions of these capabilities.

There are numerous algorithms and schemes which, basically, use some sort of computer search to construct codes fullfilling the design requirements. Pioneering this work, was Gallager, with his "pseudorandom procedure" for generating random, regular codes.



(a) BER performance

(b) Detected Errors

(c) Undetected Errors

(d) Decoder Errors

Figure 4.5: Comparison of the same ensemble, varying over increasing girth. Note the expected gain in avoiding 4-cycles, and, conversely, the similarity of $g = 6$ and 8.

4.4.1 Gallager's Pseudorandom Procedure

The original work by Gallager [4] describes a scheme which is part structured (explicit) and part random. His constructions were parameterized by the framework (N, γ, ρ),[15] strictly regular codes.

By horizontally partitioning H into γ submatrices, the construction ensures that every row (column) contains exactly ρ (γ) non-zero entries.

$$\left[\begin{array}{cccccccccccc} 1 & 1 & 1 & 1 & & & & & & & & \\ & & & & 1 & 1 & 1 & 1 & & & & \\ & & & & & & & & \ddots & & & \\ & & & & & & & & & 1 & 1 & 1 & 1 \\ \hline & & & & \pi & (& \mathbf{H}_0 &) & & & & \\ \hline & & & & & & \vdots & & & & & \\ \hline & & & & \pi & (& \mathbf{H}_0 &) & & & & \end{array}\right]$$

[15] Original notation is (N, j, k) [4].

Construction 1 Gallager's Construction Method [4].

Initialize the first row of H with ρ non-zero entries along the first ρ positions. Each following row of this first $(N/\gamma) \times N$ submatrix is then a ρ-bit shift of the previous. Finally, the remaining submatrices are random column permutations of that first submatrix.

Construction 2 S. Lin and D. J. Costello's random method [17].

Given a partial construction, H_{i-1}, and a candidate vector v for position h_i. If candidate vector h_i does not introduce any 4-cycles with respect to the previous submatrix, H_{i-1}; and if all rows in $H_i = H_{i-1} \cup \{h_i\}$ obey the row-weight bound, h_i is permanently added to H_{i-1}. Otherwise, h_i is permanently rejected, and *another candidate* is picked from the pool. This process is repeated until N columns are added to H (success); or there are no further candidates in the pool (failure).

The "stair-case" initialization of the first submatrix guarantees that any column permutation will always have exactly *one* non-zero position in *every* column. In total, the resulting H matrix must be a strictly (γ, ρ)-regular LDPC code. However, the girth of H depends on the permutations chosen, and Gallager suggested avoiding 4-cycles without describing any specific method for determining good permutations of the submatrices. Hence, this scheme depends on computer search.

4.4.2 Lin and Costello

A quite straight-forward technique, due to Lin and Costello [17], is to use a variation on random construction, where the $m \times N$ matrix (where $m = N - RN$) is grown column-by-column. In addition to fixed weights due to regularity, design considerations also include $g > 4$. In other words, we have a (N, γ, ρ)-ensemble, similar to no. 4 in Def. 1.

By means of brute force computer search, the algorithm tries to find a valid column vector, h_i, to add to the previous construction H_{i-1}. The process is based on a heuristic approach, where candidate columns are picked at random from a pool, P, of all possible binary, weight-γ vectors of length m. Each candidate is subject to the constraints of the ensemble, which determine whether it can be permanently added to H_{i-1}; See Constr. 2. Given a subcode H_{i-1} satisfying all constraints γ, ρ_{max}, and g, we still need to evaluate each candidate for the vacant position by means of a time-inefficient *look back* scheme. In our simple implementation, this had complexity $\mathcal{O}(im\gamma^2)$. As the search space H_{i-1} grows, it becomes increasingly difficult to find a valid vector, and the above evaluation is repeated exponentially many times.

Following this method, a matrix H is constructed, in which all columns have equal weight; no row has weight exceeding an upper bound; and no two columns share more than *one* common row. Furthermore, due to the random selection of candidate columns, we are not guaranteed full rank, yet the code rate, R, is lower bounded due [17], and upper bounded by our 'design rate' (4.1), giving;

$$1 - \frac{\gamma}{\rho} \leq R < R'. \tag{4.8}$$

A straight-forward implementation of this method reveals the inherent disadvantages involved in such a relatively brute-force approach. The crucial element in this method is to avoid early exhausting of the pool of candidates. By proper adjustment of the design parameters N and γ, the pool, P, of candidate vectors h_i can easily be made considerably larger than the required number, N, of suitable vectors;

$$|P| = \binom{N-k}{\gamma} = \binom{N(1-R)}{\gamma} \gg N. \tag{4.9}$$

Increasing the offset between $|P|$ and N certainly improves the probability of completing a construction. Yet, the method is not particularily adapted to handling such large sets of vectors in an efficient way. For instance, we found it impractical to keep track of *which* of the $|P|$ candidate vectors had previously been considered. This would require extensive bookkeeping, which would also lead to look-up latency. By randomly selecting a vector–which might very well be in use, or previously rejected–the subsequent look-back validation will in any event determine whether it can be used;

1. A previously rejected vector will never be accepted as the conflict causing the initial rejection, will still exist in H_{i-1}.
2. If v is already in use in H_{i-1} we would obviously detect an overlap of $\gamma > 2$ positions, causing rejection.
3. Otherwise, it will *only* be accepted if it satisfies the normal requirements of this framework.

Our analysis of this algorithm concludes that the major disadvantage is the repeated scan through the previous submatrix, H_{i-1}, to (re)evaluate candidate vectors. A practical approach towards constructing real-life codes should not rely on any form of 'backward-looking' computer search.

4.4.3 Complete Acyclic

Before we introduce the most important construction algorithm in this thesis, the Extended Bit-Filling Algorithm, we describe a straight-forward method of constructing completely acyclic graphs. This is not meant as an attempt at designing a code construction algorithm, but rather a way of bounding what is achievable in terms of girth.

The first bit, v_1, is connected to γ arbitrary checks. Obviously, each subsequen bit v_i could select γ among the $m - \gamma i$ remaining unused checks, but, since we are trying to maximize rate, we need to be careful not to exhaust all m checks prematurely.

At stage $i > 1$, there are $u_i = (i-1)(\gamma-1)+1$ used checks. Bit v_i can connect to exactly *one* of these checks without closing any cycle; while the remaining $\gamma - 1$ checks must be unused. Hence, this scheme may proceed like this for as long as $u_i \leq m - (\gamma - 1)$; i.e., for

$$\iota = \lfloor \frac{m-\gamma}{\gamma-1} \rfloor + 1 \tag{4.10}$$

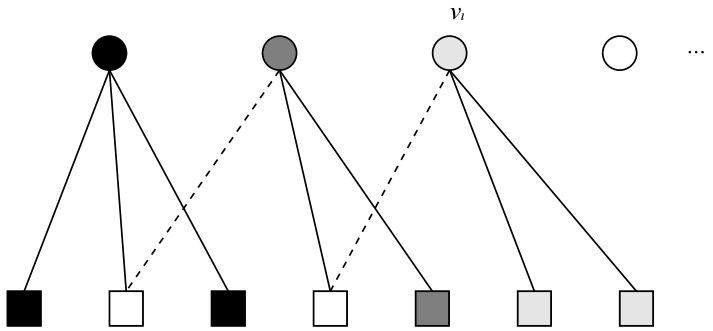


Figure 4.6: For $m = 7$ and $\gamma = 3$, $\iota = 3$ bits are connected while $\mathcal{G}$ is still acyclic.

bits. Beyond this point, we can not connect any bit (using $\gamma = 3$ edges) without closing a cycle in $\mathcal{G}$. Fig. 4.6 shows an example where the 2nd and 3rd edge of $v_{\iota+1}$ will close cycles.

Using a similar reasoning, it is possible to bound the length of the longest possible cycle as a function of ι. For the sake of argument, consider the situation where each bit prior to v_ι places its one 'backward' edge within the checks unique to the previous bit. Define the set of checks unique to bit v_i as C_i (the shaded checks in Fig. 4.6). Then it is possible for $v_{\iota+1}$ to connect to both 'ends' of $\mathcal{G}$; namely, C_1 and C_ι. This Hamiltonian cycle will then traverse the entire (sub)graph for a girth of

$$g_{\max} = 2(\iota + 1). \tag{4.11}$$

Note that any subsequent connection (edge) can not possibly close any longer cycle in $\mathcal{G}$, since we have exhausted all ι 'acyclic bits.' Interestingly, $g_{\max}$ depends only on m, and not on the number of bits, N. Note that the maximal cycle in the example of Fig. 4.6 is $2(\iota + 1) = 8$.

By investigation of the graph, it is apparant that the underlying matrix is redundant. For instance, the rows corresponding to the shaded check nodes in Fig. 4.6 are identical (connected to one and the same bit, only). Also, there are other dependencies. As we have discussed earlier, tree graphs define poor codes. However, this tree is only a intermediate, acyclic construction on which larger, cyclic yet high girth codes may be built. In the following, we will use the girth-bound (4.11) in the design of more complex LDPC construction schemes.

4.5 Bit-Filling

Viewing an LDPC code as some instance of an ensemble, which may consist of a large number of similar (not equivalent) codes comes in very useful when faced with the complex problem of randomly constructing LDPC codes. Given some framework of desired parameters this problem boils down to sampling *one* of the (perhaps numerous) codes in this ensemble.

After reviewing several publications on the randomised construction of LDPC codes, the work of Campello *et al.* on the 'Bit-Filling' (BF) algorithm [2, 1]

Construction 3 J. Campello and D. S. Modha's *Extended Bit-Filling algorithm* [1].

Given a subgraph $\mathcal{G}_{i-1}$ of girth $G(H_{i-1}) > g' \geq g$, the next bit node, v_i, is connected to a check, $c^\star$, selected from a subset $F = \mathcal{V}_C \setminus \{U \cup \bar{A}\}$. Any $c' \in U$ would close a cycle (if connected to v_i), and $\deg(c'') = \rho$, $\forall\, c'' \in \bar{A}$ (max degree). By updating these sets after connecting $(v_i, c^\star)$, we can safely select the next check from F. If $F = \emptyset$ and $g' > g$ we can relax the girth bound, $g' := g' - 2$, and free checks from $U \to F$ (otherwise, construction fails). Repeat until $\lfloor m\gamma_i \rfloor$ checks are connected to v_i, and proceed to v_{i+1} with $U = \emptyset$. If $i = N$, construction is successful.

appeared to be quite useful. Randomised construction algorithms often rely heaviliy on computing power. Several investigated articles, e.g. [32] and [17], check that design parameters are upheld by repeatedly *looking back*–searching through the intermediate construction–and using this information to guide the construction process so that it may steer clear of violations.

Randomised algorithms tend to get stuck in situations from which they can not proceed without violating some constraint. As such, a fail-then-retry approach is required in order to increase the frequency of complete constructions. The Bit-Filling algorithm is such an algorithm that often requires many repetitions to complete a construction. Clever, yet simple, measures are taken to improve the difficulties of probabilistic design, most importantly, the wasteful look-back process is replaced by a more efficient *look-ahead* design. Simple bookkeeping underway increases overall efficiency, while–as with look-back–aids the construction, guiding it away from pitfalls and dead ends.

The BF algorithm was initially introduced in 2001 [2]. The same year, it was modified to the Extended Bit-Filling (EBF) algorithm [1], described in a follow-up paper. The main principle remained the same, but several ideas were improved. In this section we will reproduce the characteristics of this algorithm, and also discuss some of our implementational modifications which appear to improve the algorithm.

The EBF algorithm is a multi-purpose construction scheme which is capable of producing LDPC codes from different ensembles (ref. Def. 1; no. 5 in particular). By choosing one parameter as the optimisation the scheme attempts to add as many columns (bits) to the construction as possible without violating the constraint. However, compared to Constr. 2, EBF employs a more fine-grained approach by locally searching for the optimal vector, check by check. By means of a *greedy approach,*[16] it is possible to solve the construction problem in polynomial time, as opposed to other, often exponential solutions.

This improvement enables one to construct more extreme codes than what is generally possible using other algorithms. In comparison with other schemes, EBF shows competitive results, often even outperforming the competition.[17]

As an example, we consider a regular construction which is optimized on

[16] Where local decisions are based solely on the information currently available–the state of the construction so far–"in hope of finding the global optimum." [40]

[17] At the time, Campello *et al.* used the codes of MacKay's design [41] as a test of strength. See Sect. 4.7 for our results.

girth, $G(H)$. Given parameters $(m, g, \gamma(x), \rho_{max})$ locally good (not necessarily locally optimum) column vectors of length m (and weight $\lfloor m\gamma_i \rfloor$) are found, and added to H. In this example, vectors must satisfy the requirements that no row vectors outweigh ρ_{max} (regularity), and the girth of the graph remains $g \leq G(H) \leq \bar{g}$ (optimisation). Here, $\bar{g}$ is an upper bound on $G(H)$.

In its straight-forwardness of design, EBF is a quite versatile algorithm which can easily be modified to optimise on other parameters. For instance, there is often the need for maximum-rate codes where we try to minimise m, while all other parameters are fixed. In other scenarios we may be forced to operate at a given rate, where the aim might be to maximise girth. In addition to the above example, Campello *et al.* demonstrated irregular constructions by fixing a column weight sequence, $\gamma(x)$; and graphs of maximized girth by using a technique of "relaxing" girth, which we will discuss in the following. This flexibility is perhaps the most interesting aspect of the EBF algorithm. In the following discussions we will focus on the optimisation of girth, and later show how EBF is applied to other frameworks.

4.5.1 Heuristics: Adding Variables

Quite conventionally, H is 'grown' one column at a time. When column h_i corresponds to bit node v_i in the graph, this is analogous to selecting $\lfloor m\gamma_i \rfloor$ unique check nodes from the pool,

$$F = \mathcal{V}_C \setminus \{U \cup \bar{A}\}. \tag{4.12}$$

The checks in F are refered to as *feasible,* indicating that they may safely be connected to bit v_i without violating the main design constraints girth and row-weight. Infeasible checks are stored in U and $\bar{A}$, which are defined in Constr. 4.5.[18] The idea is that, by maintaining thorough 'bookkeeping' (4.12), we can add the next check in constant time; alleviating the need for complex look-back schemes. In short, this is achieved by updating (4.12) after connecting each bit, using an efficient *look-ahead* search which will be discussed shortly.

We find it more convenient to focus on the graph representation of the construction problem, so designing column h_i is thought of as identifying $\lfloor m\gamma_i \rfloor$ feasible checks to connect to bit node v_i. These checks are kept in a 'workspace,' U_1, and, at completion of this bit, these are written to column h_i of the underlying matrix, H. Since we must avoid double edges[19] in the graph, the workspace is a part of the exclusion set, U. The notation U_1 also reveals a partitioning of U, which will also be discussed later.

There is a subtle distinction between the exclusion sets U and $\bar{A}$. Checks in the former set are deemed infeasible due to the threat of closing cycles. As we move on to the next bit, v_{i+1}, it is obvious that the first edge can not close any cycle, since it is a 'dead-end,' $\deg(v_{i+1}) = 1$; as node v_i in Fig. 4.7. Hence, we reset U at the end of each completed bit (Constr. 4.5). On the other hand, once a check has reached its maximum degree, ρ, it is *permanently* excluded by moving it to $\bar{A}$. This set is never purged. In this sense, the weight constraint

[18] The original description [2] defines A, the set of (feasible) checks that are *not* fully connected. We find it more intuitive to refer to the complementing set, $\bar{A}$.

[19] Column h_i is prescribed $\lfloor m\gamma_i \rfloor$ *unique* non-zero entries; hence, no double edges in $\mathcal{G}$.

is more strict than the girth constraint, as $i \to N$. However, it is important to observe that the number of checks excluded due to girth does also grow with i, which is seen when we update F after connecting the first check to v_{i+1}.

The success of a greedy algorithm depends on the quality of the heuristic choices made during each iteration, and Campello *et al.* explore several approaches. The EBF algorithm works independently of the heuristic used, so any such scheme can be 'plugged in' as needed. The heuristics suggested in [1] are motivated by the desire to keep row weights ρ_j as homogenous as possible, by picking each $c^\star$ among the feasible check nodes least used so far. This "first-order heuristic," called `1-h`, is shown to "yield quite competitive codes," and is part of our EBF implementation. Campello *et al.* extend this method towards "complete-homogeneity" (`c-h`) by optimising the local choices; i.e. trying to identify the very least connected check in F. The search space is thereby reduced to $F_1 \subseteq F$, consisting of the checks in F that have the sparsest neighbourhood. Next, $F_2 \subseteq F_1$ is produced by considering the degree of the neighbouring checks of those in F_1. In general, $F_j \subseteq F_{j-1}$ by keeping only those checks that have the minimum-degree neighbours. The optimal (i.e., least connected in $\mathcal{G}$) check is found as soon as some subset contains *one* check; $|F_l| = 1, l \geq 1$. If, at some stage, l, $|F_l| > 1 \bigwedge |F_{l+1}| = 0$, the search fails and we apply `1-h` (select randomly) over F_l; which is still an improvement over applying `1-h` over the entire F.

4.5.2 Maximising Rate

Simulation results in [1] (and those observed in this thesis) indicate that EBF is quite competetive. Aside from the convenient reduction in complexity, what discerns EBF from other construction schemes is the flexibility in what parameter one wants to optimize. In the original publication of the BF algorithm [2], this objective was formulated as maximising rate such that $\mathcal{G}$ has girth $G(H)$ exactly equal $\bar{g} = g$.[20] By fixing parameters m and g', the construction proceeds in adding columns until, at some bit, v_i, the girth bound is violated. Since N–the actual number of bits we are able to construct–is unknown, the column-weight sequence $\gamma(x)$ is undefinable, and all columns must have equal, fixed weight, γ. On the other hand, since ρ is obviously dependent on N, no restriction can be placed on the row weights [2]. Hence, the codes produced in this setting can not be irregular (due to columns), nor completely regular (due to rows).

4.5.3 Maximising Girth

In this detailed discussion of the EBF algorithm, we will maintain focus on the optimised-girth constructions. The EBF version of the algorithm was designed more or less exclusively with the aim of solving the problem of maximizing girth–see Def. 2.

In short, EBF solves problem I.2 by repeatedly applying the original BF algorithm over a floating girth bound which is allowed to decrement from $\bar{g}$ to g. At this point, it is valuable to use the result of (4.11) to bound the maximum

[20] A lower bound of, say, $g = 4$, means "*no* 4-cycles."

Definition 2 Constructions of Maximized Girth; *Problem I.2* in [1].

Suppose that we are given positive integers m, N, ρ, and $\gamma(x)$. We would like to construct a $m \times N$ parity check matrix H with the *largest possible girth* $g \leq G(H) \leq \bar{g}$ such that H has exactly $\lfloor m\gamma_i \rfloor$ ones in the i-th column, $1 \leq i \leq N$, and at most ρ ones in each row.

girth. By letting $\bar{g} = g_{\max}$, we avoid wasting time working with girth bounds that are inachievable given design parameters R and $\gamma(x)$.

Even though the extended version (EBF) is improved to resume construction if it gets stuck, it is important to note that it is not an improvement on the main design problem–avoiding the minimum (length-g) cycles. EBF can only resume construction while $g' > g$, in which cases BF would proceed undisturbed. However, the general assumption is that EBF might produce codes of slightly higher girth than the prescribed minium bound, g. In order to continue, and possibly complete the construction (all N bits), EBF must accept the presence of ever smaller, yet all acceptable ($g' > g$), cycles in the graph. Even though there are different opinions on the actual effect of feedback in SPA decoding, there is consensus in that very small cycles will negatively affect performance (assuming a 'flooding' schedule–see Ch. 8). In summary, EBF, like BF, *does* work with a lower bound on girth, g; at which point both algorithms will fail.

4.5.4 Look-ahead; the sets U_j and $\mathcal{N}_c$

As mentioned, the first connection to a bit node, v_i, can not possibly close any cycle in $\mathcal{G}_{i-1}$. Hence, U–the set of infeasible checks due to the girth constraint–is reset at the beginning of each bit;

$$U = \emptyset, \tag{4.13}$$

such that F is updated according to (4.12). In this section, we will show that as long as U and F are maintained, no subsequent connections to v_i can close any cycle.

Since the construction process is conducted strictly from the perspective of the check nodes, we define 'neighbours' as two or more *checks* that are connected via the same bit. This bit is, then, defined as their 'parent.' Since we normally wish to avoid 4-cycles in $\mathcal{G}$, we may assume that any pair of checks has exactly *one* parent. Otherwise, if two checks were connected via several bits, we would–per definition–have a 4-cycle.

After connecting the first check, $c_1^\star$, to v_i, effort is made to look ahead and exclude from F all checks that, if also connected to v_i, would close any cycle of length $\leq g'$. To this aim, two sets are meticulously updated; $\mathcal{N}_c$–the neighbours of c; and U_j–those checks in $\mathcal{G}_{i-1}$ threatening to close a cycle of length $2j$. In order to avoid small cycles, a *breadth-first traversal* (BFT, Def. 3) from each new $c^\star$ identifies all neighbours within a radius of $g'-2$ edges[21] from the checks in U_1. BFT has complexity $\mathcal{O}(|V| + |E|)$ [42], so the complexity of connecting $\gamma(i)$ checks to bit v_i is

$$\mathcal{O}((m+i)\gamma(i)). \tag{4.14}$$

[21]Or, equivalently, since $\mathcal{G}$ is strictly bipartite, $g'/2 - 1$ *hops* (check → bit → check) away.

Definition 3 Properties of Breadth-First Traversal (BFT).

1. All neighbours of $c^\star$ within a radius of $g'-2$ edges will be identified, and marked infeasible.
2. The traversal finds the minimum distance (from v_i) to all neighbours of $c^\star$.

Fig. 4.7 illustrates the situation after connecting $c^\star$. The look-ahead function needs accurate information on the neighbours of each check in $\mathcal{G}$, so the process begins by updating $\mathcal{N}$-lists. The new edge links $c^\star$ (via v_i) to all/any previous checks $c' \in U_1$, and, conversely, links each c' to $c^\star$:

$$\begin{aligned} \mathcal{N}_{c^\star} &:= \mathcal{N}_{c^\star} \cup U_1, \\ \mathcal{N}_{c'} &:= \mathcal{N}_{c'} \cup \{c^\star\}, \ \ \forall\, c' \in U_1. \end{aligned} \tag{4.15}$$

The neighbours of $c^\star$, $c' \in \mathcal{N}_{c^\star}$, threaten to close cycles of length 4. If $g' \geq 4$ (which is normally the case), this would violate the girth bound, so these checks are appended to subset U_2. Next, $c'' \in \mathcal{N}_{c'}$ at a distance of $j+1=4$ edges from $c^\star$, threaten to close cycles of length $2j=6$. If $g' \geq 6$, then these checks are also infeasible, and are appended to subset $U_j = U_3$. Continuing in this fashion, we encounter all checks that are infeasible according to the current girth bound. The search stops when $j = g'/2$, since we must accept cycles of length $> g' \geq g$;

$$U_j = \bigcup_{c' \in U_{j-1}} \mathcal{N}_{c'}, \ \ 2 \leq j \leq g'/2 \tag{4.16}$$

where U_1 is the current connections to v_i, such that;

$$U = \bigcup_{j=1}^{g'/2} U_j. \tag{4.17}$$

When $\mathcal{N}$ and U are updated carefully, we can safely select the next $c^\star$ from F according to (4.12). Each subset U_j can be viewed as a protection against currently unwanted cycles of length $2j \leq g'$.

The degree of check-nodes is upper bounded by ρ, and the look-ahead function explores neighbours within a maximum distance of $g'-2$ edges. Hence, the complexity of updating U according to $c^\star$ is

$$\mathcal{O}(g'\rho). \tag{4.18}$$

As an example, let $g' = 12$ which gives a search radius of $j \leq 6$. Using the graph of Fig. 4.7, we note that the checks in U_4 have no unexplored neighbours ($U_5 = \emptyset$), so the updating can stop early at these "dead-ends."

To guarantee safe selection from F, the EBF algorithm requires that Def. 3 (BFT) is satisfied.

4.5.5 Relaxing Girth

As a steadily increasing number of bit nodes are randomly connected to a constant number of check nodes (that is also smaller, $m < N$, $R < 1$), it becomes

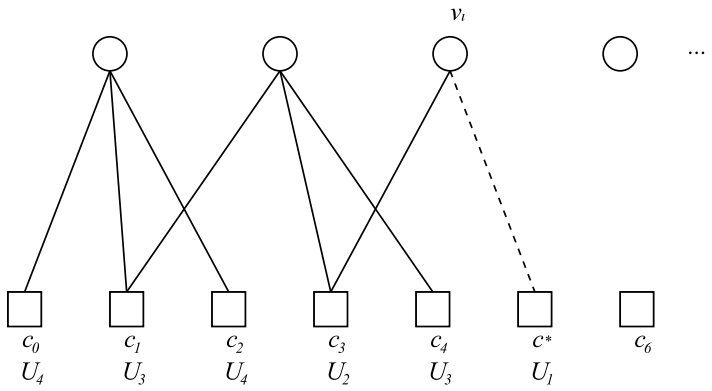


Figure 4.7: The sets U and $\mathcal{N}$ after connecting $c^\star$ to v_i; $\mathcal{N}_{c_0} = \{c_1, c_2\}$, $\mathcal{N}_{c_1} = \{c_0, c_2, c_4\}$, $\mathcal{N}_{c_2} = \{c_0\}$, $\mathcal{N}_{c_3} = \{c_1, c_4, c^\star\}$, $\mathcal{N}_{c_4} = \{c_1, c_3\}$, $\mathcal{N}_{c^\star} = \{c_3\}$, and $\mathcal{N}_{c_6} = \emptyset$.

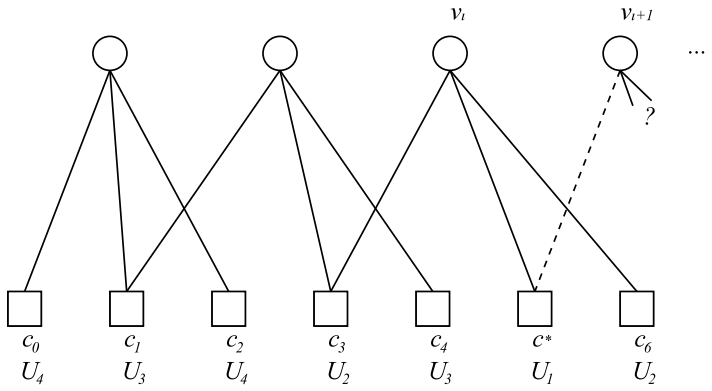


Figure 4.8: All $m = 7$ checks infeasible; $F = \emptyset$.

increasingly difficult to avoid cycles of length $\leq g'$ in the graph. Consider the situation of Fig. 4.8 where all m checks are infeasible, such that (4.12) gives,

$$U = \mathcal{V}_C \setminus \bar{A} \iff F = \emptyset. \tag{4.19}$$

At such points–where BF construction fails–the EBF construction is able to resume construction for as long as $g' > g$. To do so at the minimum expense (in terms of girth), the *least threatening* checks are freed from U, and put back in the pool, F. These are the checks that threaten to close the *longest* cycles (length g') in the graph. The girth bound is relaxed to

$$g' := g'_{\text{old}} - 2. \tag{4.20}$$

[1] suggests the freed checks are identified by *reconstructing* U (4.13) - (4.17) [2, 1] based on the current contents of U_1. Since U now only encompasses the checks out to the reduced search radius, any checks at a distance of g' edges from U_1 will necessarily be 'left behind' in F. This means that the procedure will move precisely the checks in U_j to F.

Note that is it quite possible that the corresponding subset, $U_{g'_{\text{old}}/2}$, is empty. For example, consider the situation of Fig. 4.8, as, say, $g'_{\text{old}} = 12$ is relaxed to $g' = 10$. However, $U_{g'_{\text{old}}/2} = U_6 = \emptyset$, so no checks turn out to be freed. Hence, F remains empty, and the reconstruction of U was wasted. As the algorithm tries to resume construction, this will immediately trigger an additional relaxation step (4.20). In the original EBF description [1], this process is repeated until *at least one check is freed* allowing construction to continue. Hence, girth can drop several steps within one and the same bit, each time at at the cost of reconstructing U (4.14).

During EBF execution, girth is allowed to drop from $\bar{g}$ to g, i.e., a maximum of $(\bar{g} - g)/2$ times. In our experience, girth will normally drop to its minimum value, g.[22] Furthermore, since each relaxation step requires a complete reconstruction of U "from scratch" (even in those cases where F remained empty), it is obviously valuable to streamline this operation.[23] By seperately storing the subsets, U_1 through $U_{g'/2}$, we were able to improve efficiency by never reconstructing the sets.

In the remainder of this chapter we will discuss our modifications to the Bit-Filling algorithm.

4.6 Extending the Bit-Filling Algorithm

The process of resuming construction can be performed in a more efficient manner simply by exploiting information already available in U. While implementing the EBF algorithm, we encountered several interesting improvements which we describe in this section. The aim of our adjustments are to further improve the *look-ahead* (BFT) function, see Def. 4.

The EBF algorithm does not explicitly require neighbours to be placed in their correct subset, however, maintaining this order does not require any additional computations, and can be used to increase the efficiency of the algorithm. This stronger requirement is used in the end of this section, where we introduce our modifications to the EBF algorithm.

Also, in this section, we discuss some augmentations to the scheme, which allows it to perform further tasks.

4.6.1 Improvement 1: Relaxing Girth

The crucial point of the improvements is to never reconstruct the entire set U. To achieve constant-time girth relaxation, as described in the first part of Def. 4, we modify the look-ahead function (BFT) so that it satisfies the second requirement. First, it is important to underline that breadth-first traversal from $c^\star$ (as described in the original EBF [1]) already provides us with the information required for the improvements. The first property of Def. 3 is sufficient to avoid cycles of length $\leq g'$; whereas the second determines the exact "threat level,"

[22] Even then, EBF often requires repeated attempts to complete a construction; see Ch. 7.
[23] Note that, in [2, 1], it does not appear as if this layer-information explicitliy used.

Definition 4 Improvements to the Extended Bit-Filling algorithm.

1. Immediately identify the "outermost" nonempty subset, $U_{j\max}$ (4.22), such that girth can be relaxed in *one* operation.
2. Never reconstruct U; the look-ahead function is modified to handle any necessary changes in U during the normal update according to $c^\star$ *only* (and not the entire U_1).

τ, of each infeasible check. Define

$$\tau(c) = \begin{cases} t & \text{if} \quad c \in U_t \\ 0 & \text{if} \quad c \in F \end{cases} \tag{4.21}$$

In the aim of maximizing girth, shorter cycles are more harmful. Consequently, we interpret low values of (4.21) as *high* threat levels.

As connections are added to the current bit, v_i, BFT indentifies all checks in $\mathcal{G}$ that are infeasible due to the particular choices in U_1. As the traversal "fans out" from U_1, (4.16) - (4.17), all neighbouring checks within a radius of $g'/2-1$ *hops* (a hop is two edges) are stored in U. We observe that BFT can be stopped early if–at some point, $j < g'/2$–we run out of neighbours to visit. Define $j_{\max}$ as the index of the 'outermost' non-empty subset;

$$j_{\max} = \max_{2 \le j \le g'/2} \{\, j \,:\, U_j \neq \emptyset \wedge U_{j+1} = \emptyset \,\}. \tag{4.22}$$

Note that, if $U_{j+1} = \emptyset$, then–by neccessity–all subsequent subsets must also be empty, since U_{j+1} leaves no neighbours to follow further. The stopping terms of both (4.16) and (4.17) can now be changed to read

$$2 \le j \le \min(g'/2, j_{\max}), \tag{4.23}$$

such that BFT may terminate as soon as possible.

According to (4.23), the search is stopped early when there are no further unexplored edges, indicating that the distance (in hops) from U_1 to its outermost neighbours may be $\le g'/2-1$. To satisfy the first improvement of Def. 4, we simply require BFT to keep track of where it terminates. $j_{\max}$ (4.22) is defined as the index of the outermost non-empty subset, at a distance of $j_{\max}-1$ hops away from U_1. This gives,

$$U_{j\max} \neq \emptyset, \text{ and, } U_i = \emptyset\,, \forall\, j_{\max} < i \le g'/2. \tag{4.24}$$

When we need to relax girth, we avoid repeatedly decrementing g' since we already know $j_{\max}$. (4.23) immediately identifies the neighbours posing the *minimum threat*, namely $U_{j\max}$, and girth is relaxed in one step to

$$g' := 2j_{\max} - 2, \;\; j_{\max} \le g'_{\text{old}}. \tag{4.25}$$

Hence, by slightly augmenting the normal bookkeeping (BFT) procedure of EBF, we are able to relax girth in constant time, using (4.22) and (4.25). Hence, we have satisfied the first demand of Def. 4.

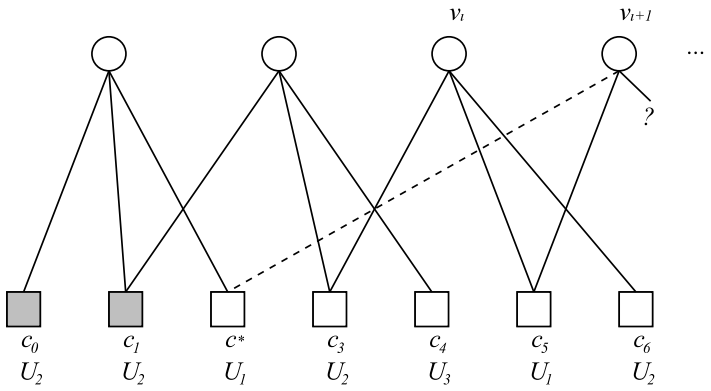


Figure 4.9: After connecting c_2, the regular updating of U handles any reordering of subsets; note the grey checks have been "moved down" to their correct subsets.

4.6.2 Improvement 2: Updating U

According to the second demand of Def. 4, we wish to avoid reconstructing U (from scratch) as a part of relaxing girth. We have seen how to immediately identify the checks that are to be freed, but what happens when we skip reconstruction, and simply proceed with the construction?

The next connection, $c_2^\star$, will be selected from these recently infeasible checks, which suddenly places $c_2^\star$ in the most immediate subset, U_1. This, in itself, is not a problem since we have already accepted the presence of cycles of length $2j_{\max}$ edges (or, $j_{\max}$ hops). However, the neighbourhood of $c_2^\star$, is now reachable via a *shortcut*–namely, via the new edge, $(v_i, c_2^\star)$. Keeping with the second demand of Def. 4, we need to avoid "reconstructing U from scratch" [1] after each girth relaxation; saving a total complexity of $\mathcal{O}(|U_1|g'\rho)$ (ref. (4.18)). To account for shortcuts without reconstructing U, we need BFT to make the necessary adjustments to U during the normal updating according to $c_2^\star$. We will now discuss the second demand.

Returning to the example of Fig. 4.8. At the start of bit v_3, the graph is still acyclic, $g' = \bar{g}$. After connecting the first check, c_5, and updating the proper $\mathcal{N}$-lists (4.15), all checks become infeasible; i.e.,

$$\begin{aligned} U_1 &= \{c_5\} \\ U_2 &= \mathcal{N}_{c_5} = \{c_3, c_6\} \\ U_3 &= \mathcal{N}_{c_3} \cup \mathcal{N}_{c_6} = \{c_1, c_4\} \\ U_4 &= \mathcal{N}_{c_1} \cup \mathcal{N}_{c_4} = \{c_0, c_2\} \\ U_5 &= \mathcal{N}_{c_0} \cup \mathcal{N}_{c_2} = \emptyset. \end{aligned}$$

As we know from (4.10), v_3 should be expected to close one/several cycles of maximum length, $g_{\max} = 8$. This coincides with (4.22), which yields $j_{\max} = 4$, and we relax girth according to (4.25) giving $g' := 6$, meaning "no 6-cycles." Now, construction can resume with $F = U_4$.

Consider what happens if we simply proceed with the construction at this point without reconstructing or updating U. Say the next connection is c_2, as

illustrated in Fig. 4.9. Using (4.16) - (4.17) we would add c_2 to U_1, and c_0 to U_2. However, $\mathcal{N}_{c_2} = \{c_0, c_1, c_5\}$ are already *in* U, so the procedure would stop, wrongfully leaving c_1 in U_3. Again, $F = \emptyset$, and the second relaxation step ($g' = 4$) would free $U_3 = \{c_1, c_4\}$, *even though* c_1 *will close a* 4*-cycle.* Somehow, c_1 must be moved down to U_2 so that its true threat-level is respected, and we do not risk mistakenly closing unneccesarily short cycles.

After connecting a new check, $c^\star$, the BFT processes its neighbours, $\mathcal{N}_{c^\star}$. However, to avoid situations as above, we suggest a simple modification of re-processing those checks that are obviously posing new threat-levels. These can be identified by checking (4.21). In short; while processing the new $c^\star$, BFT detects any 'shortcuts' in U; moves those checks down to their correct subset; and, process the neighbours of these particular checks *only*. Since BFT now re-processes only those checks that are necessary, we have satisfied the second demand of Def. 4, and the update is optimal. This is yet another improvement to the EBF algorithm. To be precise, we change the BFT rules from (4.16) - (4.17) to,

$$U_j = U_j \bigcup U_j^{\text{new}} \setminus U_j^{\text{old}}, \quad 2 \leq j \leq \min(g'/2, j_{\max}) \vee U_{j-1} \neq \emptyset, \tag{4.26}$$

using the stopping rule of (4.23). Also, the update stops early if there are no further neighbours to (re)process. In compliance with Def. 4, the update is based on $c^\star$ *only,* giving $U_1 \triangleq \{c^\star\}$[24] and,

$$\begin{aligned} U_j^{\text{new}} &= \{\, c \in \mathcal{N}_{c'} \,:\, c' \in U_{j-1} \,\wedge\, (\tau(c) = 0 \,\vee\, \tau(c) > j) \,\}, \\ U_i^{\text{old}} &= \{\, c \in \mathcal{N}_{c'} \,:\, c' \in U_{j-1} \,\wedge\, \tau(c) = i > j \,\}. \end{aligned} \tag{4.27}$$

Note that this scheme also contains the normal look-ahead function; by processing all 'undetected' neighbours for which $\tau(c) = 0$. When the updating (4.26) is complete, $c^\star$ is added to workspace U_1.

Continuing with the previous example; after connecting c_2, the following updates are made to U. First, we note the stopping term (4.23) gives $2 \leq j \leq 3$. As prescribed, BFT begins at $j = 2$, by redefining $U_1 = \{c_2\}$ (saving $U_1 = \{c_5\}$ in memory). $\mathcal{N}_{c_2} = \{c_5, c_0, c_1\}$, so (4.27) checks the threat-levels of these checks. $\tau(c_5) = 1 < j$, so we do not consider this check further. However, $\tau(c_0) = 4 > j$, so we move this down to U_2, and mark it for reprocessing. For $j = 2$, (4.26) - (4.27) give

$$\begin{aligned} & U_2^{\text{new}} = \{c_0\}, \text{ and, } U_2^{\text{old}} = \emptyset, \text{ such that,} \\ U_2 = U_2 \,\cup\, & U_2^{\text{new}} = \{c_3, c_2\}, \text{ and,} \\ & U_4^{\text{old}} = \{c_0\}. \end{aligned} \tag{4.28}$$

Next, $j = 3$ with $U_2^{\text{new}} = \{c_0\}$. Since the stopping criterions of (4.26) have not been met, we process $\mathcal{N}_{c_0} = \{c_1, c_2\}$. Since $\tau(c_1) = 2 < j$, and $\tau(c_2) = 1 < j$, c_1 and c_2 are not considered further. Now, $U_3^{\text{new}} = \emptyset$, so we have no "fresh checks" to process further. Simultaneously, at this point $j = g'/2 = 3$,

[24] For compact notation, we allow the ambiguity of redefining U_1. Note that this is only for illustrative purposes, to indicate how the update is based solely on the new check, $c^\star$.

and we have reached the boundaries of the new search radius, so the updating stops by updating $U_1 = U_1 \cup \{c^\star\} = \{c_5, c_2\}$. The updated (and safe) U is now

$$U_1 = \{c_5, c_2\}$$
$$U_2 = \{c_0, c_1, c_3, c_6\}$$
$$U_3 = \{c_4\}.$$

Since $F = \emptyset$ once again, we may safely free the outermost subset, which in this case is U_3, without risking cycles of length $< 2g'_{\text{old}} = 6$. By using the information already available, the added complexity of reducing girth by reconstructing U (4.14) is alleviated–at no extra cost to the regular BFT 'look-ahead' function.

4.6.3 Extension 1: Local Girth Detection

The EBF algorithm focuses on avoiding cycles (in $\mathcal{G}$) of length $g <\leq g' \leq \bar{g}$. Each time the running girth bound is relaxed $g' := g'_{\text{old}} - 2$, we 'accept' the presence of one (or several) length g'_{old} cycles, and proceed with the new, moderated aim of avoiding length g' cycles (where $g' < g'_{\text{old}}$). Our modified BFT guarantees that all 'infeasible' checks are in their correct, minimum subset of U. Following a girth-relax, we have that $F = U_{j\text{max}}$–the least threatening checks–so we know that the next $c^\star$ closes a cycle of length $2j_{\text{max}}$ edges (or, j_{max} hops).

If EBF construction succeeds, the girth of the graph is $G(H) = g' + 2 > g$. However, girth is (per definition) only a *lower bound* on the length of the cycles in $\mathcal{G}$, and there will obviously be *longer cycles* in $\mathcal{G}$, and it can be valuable to know the more fine-grained girth profile of the graph. The *girth of bit* v, g_v, is defined as "the length of the shortest cycle [in $\mathcal{G}$] that passes through v" [43] (see also Ch. 8). Considering the bit-by-bit and girth-by-girth construction of EBF, we suggest an extension to the algorithm which, again, uses already available information to keep a running track of the girths of all bits. Prior to construction, we initialize all bits with the maximum cycle length according to m; $g_v = g_{\text{max}}$ (4.11).

Returning to the example of Fig 4.8 where, as discussed, the graph is acyclic in the first $\iota = 3$ bits. Next, the first edge connected to $v_{\iota+1}$ can not close any cycle, so the graph is still acyclic after connecting c_5. After updating U on $c^\star = c_5$, using (4.26) - (4.27), we have,

$$U_1 = \{c_5\}$$
$$U_2 = \{c_3, c_6\}$$
$$U_3 = \{c_1, c_4\}$$
$$U_4 = \{c_0, c_2\}.$$

Once again in this small example, all checks are infeasible, and we must relax girth in order to proceed. (4.23) gives $j_{\text{max}} = 4$, such that (4.25) gives $g' := 6$, and EBF proceeds with $F = U_4 = \{c_0, c_2\}$. As the next check, c_2, is connected, we know that we close one or several 8-cycles–the question is then which bits are affected by these cycles? Incidentally, these are the 'shortcuts' through U which the modified BFT is designed to avoid (4.26) - (4.27). Using the information

in U *before updating on* c_2 (BFT), we may enumerate these 8-cycles by tracing the paths from $c_2 \in U_4$, all the way down to U_1.

Define the look-up table π as containing the parent (i.e., bit node) of all pairs of check nodes is $\mathcal{G}$. I.e., $\pi(c_a, c_b) = i$ iff $c_a \in \mathcal{N}_{c_b} \wedge c_b \in \mathcal{N}_{c_a}$; otherwise, $\pi(c_a, c_b) = -1$. Define L_j as the check(s) connecting two adjacent subsets, U_{j+1} and U_j,

$$L_j = \bigcup_{c' \in L_{j+1}} \bigcup_{c \in U_j} \{\, c \,:\, \pi(c', c) \geq 0 \,\},\ 1 \leq j < j_{\max}, \tag{4.29}$$

where $L_{j_{\max}} \triangleq \{c^\star\}$, only. Note that the total number of cycles is determined by the number of links between any adjacent subsets; i.e., as $n = |L_j|$ for any $1 \leq j < j_{\max}$ (excluding $L_{j_{\max}}$, which is defined of size 1). All cycles are obviously closed via the current bit, v_i.

(4.29) enumerates all *checks* that are part of the n cycles, of length $g_v = 2j_{\max} - 2$. The *bits* that comprise the other half of the nodes in these cycles are exactly those found in (4.29), where $\pi(c', c) \geq 0$. The EBF 'look-ahead' guarantees that there are no shorter cycles in $\mathcal{G}$, hence, g_v must be the girth of these bits; where girth is (again) defined as the length of the shortest cycle.

Continuing the example, where we have $j_{\max} = 4$ and the current bit node is v_3. Beginning with $L_4 = \{c_2\}$ (as defined), we work our way down towards U_1, finding (4.29),

$$\begin{aligned} L_3 = \{c_1\} &\Leftrightarrow \pi(\mathbf{c_2}, c_1) = v_0 \\ L_2 = \{c_3\} &\Leftrightarrow \pi(\mathbf{c_1}, c_3) = v_1 \\ L_1 = \{c_5\} &\Leftrightarrow \pi(\mathbf{c_3}, c_5) = v_2. \end{aligned} \tag{4.30}$$

We have thereby enumerated all bits that are 'touched' by the new cycle, and may update their local girths correspondingly. For illustrational purposes only (the *edges* are irrelevant), we output the $n = |L_3| = |L_2| = 1$ cycle of length $g_v = 8$. By appending the current bit, v_4, to both ends of (4.30) we have,

$$\mathbf{v_3} \rightarrow c_2 \rightarrow v_0 \rightarrow c_1 \rightarrow v_1 \rightarrow c_3 \rightarrow v_2 \rightarrow c_5 \rightarrow \mathbf{v_3}.$$

Using EBF to update the local girths of bits during construction adds only a constant overhead to the update (4.26) - (4.27). As emphasized above, the purpose of this extension is *not* to enumerate the cycles occurring in $\mathcal{G}$, but rather to ensure that the (augmented) EBF scheme terminates with the precise girth of each bit, which–as we will discuss further in Ch. 8–can be used to facilitate SPA decoding.

4.6.4 Extension 2: jumpBack

As indicated by the results of Ch. 7, the Bit-Filling algorithms often require many executions before completing constructions. The original EBF algorithm [1] is identical to 'regular' BF [2] in the way it handles failed constructions; both discard any incomplete work and initiate a new attempt "from scratch." When working with heuristic algorithms, it is often possible to *reuse* at least parts of the failed work as a basis for further constructions.

j	0	1	2	3	4	5	6	7	8
g'	20	18	16	14	12	10	8	6	4
i'	5	5	7	9	10	10	10	–	–

$\ldots$

Table 4.1: The BFT is extended to also keep track of *where* the girth-bound was relaxed, such that we may resume construction from any position $v_{i'}$.

Given the previous discussions on the randomised LDPC construction problem, it appears as if the 'hardness' of completing the next bit grows linearily with i. If so, one could claim that some proportion of constructions, say the first 70 percent of the bits, is almost independent of the overall construction problem. It is within the final part of $\mathcal{G}$ that the random choices have the most dire consequences, making this an appropriate focus for computational resources.

By simple bookkeeping, it is possible to resume construction from any arbitrary bit. Since we will resume construction from the beginning of a bit, we know that we can immediately reset U and F according to (4.12) and (4.13). Furthermore, we will need to *strengthen* the girth bound (oppsite of relaxing) to make up for the relaxations which led to $g' < g$ and failure. In order to be able to 'jump back' to any bit $i' < i$, we will need to keep track of the positions i where girth was relaxed. Defining the look-up table

$$r[j] \triangleq \text{position (bit) where } g' = \bar{g} - 2j, \qquad 1 \leq j < g'/2. \tag{4.31}$$

When resuming from position i', we can immediately strengthen g' to

$$g' = \bar{g} - 2l \ : \ l = \max\{\, i \ : \ r[i] \leq i' \,\}. \tag{4.32}$$

Recall that girth may very well have been relaxed several times within one and the same bit. Consider a small example, where, say, $\bar{g} = 20$. At position (bit) v_{10} the construction fails; see Table 4.1. Say we want to resume construction from $i' = 8$ (v_8), we find that (4.32) gives $r[2] = 7 \leq i' = 8$, such that $g' = 20 - 4 = 16$; which corresponds with Table 4.1.

Resuming construction from bit $v_{i'}$ involves the repeated 'unplugging' of all bits v_i, $i \geq i'$. Removing the edges connecting bit v_i to $\mathcal{G}$, is mainly a matter of bookkeeping on the row-weight exclusion set, $\bar{A}$, and the neighbour-sets, $\mathcal{N}_c$. Recall that the underlying H-matrix is defined as an adjacency matrix of $\mathcal{G}$, so we determine the connections to be removed by inspecting the corresponding column of $h_{i'}$;

$$U_1^{(i')} = \{c_j \ : \ H_{j,i'} \neq 0,\ 0 \leq j < m\}. \tag{4.33}$$

Since we have no double edges in $\mathcal{G}$, we adjust $\bar{A}$ by simply reducing the degree of each check $c \in U_1^{(i')}$ by one;

$$\bar{A}_c := \bar{A}_c - 1, \quad \forall\, c \in U_1^{(i')}. \tag{4.34}$$

Also, we need to undo the bookkeping of $\mathcal{N}$-lists, (4.15). To avoid having to scan through the columns of H once again, we perform this simultaneously with

γ	m	N						
			Bit-Filling					
		MacKay	I	II	III	Av.	Att's	ρ_{max}
3	60	492	485	489	**512**	467	791	27
3	62	495	483	508	**550**	500	3615	28
3	90	998	1087	1120	**1171**	1085	2641	40
3	100	900	1339	1353	**1452**	1351	231	45
3	111	999	1636	1717	**1786**	1677	1801	50
4	222	1998	2752	1967	**3164**	2970	4531	58
4	282	4376	4821	4867	**5128**	4809	1932	83
4	300	4096	5499	5499	**5867**	5473	3552	87
4	444	3584	12360	12370	**13035**	12429	1007	128

Table 4.2: Maximising rate using EBF, compared to results of MacKay. Columns labelled 'I' and 'II' are from [1], while our results are in the rightmost subtable, starting with the column 'III.'

(4.34). In our EBF implementation we consequently *append* to the neighbour-lists as new connections are registered. By simply stripping off the final $\bar{A}_c - 1$ entries of $\mathcal{N}_c$, we remove all/any edges previously connecting c to the expired bits;

$$\mathcal{N}_c := \mathcal{N}_c \setminus \{ \text{ final } A'_c - 1 \text{ entries of } \mathcal{N}_c \}, \quad \forall c \in \mathcal{V}_C. \tag{4.35}$$

This process (4.34) - (4.35) is repeated until we reach the 'destination bit,' i'. At this point, the construction has been reset to the beginning of bit $v_{i'}$, and can proceed without any further considerations. The look-up table (4.31) is then overwritten (according to BFT) from this position i', to facilitate future 'jumping back.'

4.7 Results

The Bit-Filling algorithm (BF), and it's extended version, EBF, were successfully implemented using `C++`. As "main heuristic," we chose the "first-order homogeneity," `1-h` [1].

4.7.1 Maximizing Rate

In assessing the performance of our version–without modifications described in Ch. 4.6.4–we reproduced the results of [2], in which EBF is compared to the results of MacKay [41] in terms of maximized rate. By running EBF 5000 times, keeping the optimal (highest N) code, our results showed improvement over the original EBF results;[25] see Table 4.2.

Our data is presented in the rightmost subtable, along with results on the average blocklength, followed by the number of constructions (in 5000) before the maximum blocklength was achieved. As in the original table, all codes are

[25]Even when compared to the "complete homogeneity" heuristic, `c-h`, which we did not implement.

γ	m	N	g								
			II	III	Att.	+2	+4	IV	Att.	+2	+4
3	408	816	8	8	3	89.0% 84.9%	57.0% 54.2%	8	1	87.4% 84.9%	55.8% 54.2%
3	504	1008	8	8	3	96.0% 91.4%	61.1% 57.9%	8	1	94.0% 91.4%	60.2% 57.9%
4	544	816	8	8	12	66.2% 62.4%	43.5% 40.7%	8	1	65.3% 62.4%	42.9% 40.7%
3	272	408	10	10	2	68.9% 67.2%	56.1% 52.9%	10	15	68.1% 67.2%	54.2% 52.8%
3	544	816	10	10	1	83.3% 79.2%	65.3% 62.4%	10	1	81.1% 79.2%	64.2% 62.4%
3	1280	1920	12	12	103	72.8% 70.7%	61.7% 60.0%	12	127	71.9% 70.7%	61.5% 60.0%

Table 4.3: Maximising girth using EBF, again compared to [2] with our results in the two rightmost subtables. Column 'IV' is the results of using the extensions suggested in this thesis.

of girth 6 (i.e., no 4-cycles), and, since we were maximizing blocklength, ρ was left unrestricted. The resulting maximum row-weight is listed in the rightmost column, and these values are rather in the high-end making the high-rate codes not as sparse as desirable.

4.7.2 Maximizing Girth

The most prominent feature of the EBF algorithm is the ability to maximize girth. Again, comparing to [3] ($g = 6$), Campello *et. al* presented the results of Table 4.3 in [1]. Our results are in the subtables marked 'III' and 'IV.'

In all experiments, we bounded girth by $[\, g_{II}\,,\, 100\,]$, where g_{II} refers to the value prescribed by column 'II.' We observed the resulting (maximum) girth over a total of 5000 runs. In subtable 'III,' we easily reproduced the published EBF results using our 'basic' implementation. This implies that construction must resume 'from scratch' after every failure ($g' < g_{II}$). Using this same technique, we were unable to improve the results. Columns '+2' and '+4' are attempts at the two successive girths (e.g., if $g_{II} = 8$, we tried 10 and 12), again for 5000 runs each. The top number shows the maximum completed columns, while the bottom number is the average over 5000 runs–both as percentage fractions. However, many came close ($> 80\%$ average), so perhaps minor adjustments to the parameters–such as increasing no. runs; or, increasing ρ–would allow improvement.

The data in subtable 'IV' are the same experiments repeated using the modified EBF algorithm. Here, we use the 'jumpBack' extension described previously, with parameters set such that if construction completes $> 50\%$ of N (more than half done), then the following $1.3N$ attempts will resume from this half-way point, using this sub-construction as a 'basis' upon which to try a different set of random choices. Unfortunately, these results are not encouraging, and show that–in this experiment, at least–the EBF algorithm does not improve with the jumpBack scheme. However, it should be underlined that it would be satisfactory to try to adjust the threshold, and see whether improvement could be found; however, this was beyond what was achievable within the deadline of

this thesis.

Chapter 5

Encoding

In order to approach the theoretical limits due to Shannon, Gallager deviced extremely long, random codes that are also very sparse. This facilitates decoding, and, as known (and discussed in Ch. 6), the results are still quite impressive. However, the tradeoff limiting the application of random LDPC codes is the high encoding complexity.

Encoding contributes to the distinction between LDPC codes of practical and theoretical interest. The former group is dominated by random codes with blocklengths of the order 10^{4+}, where performance can be made arbitrarily close to the Shannon limit [12]. Among the latter, more applicable codes, are the "structured" codes discussed briefly in Ch. 4. Often, these are designed in such a way as to facilitate efficient encoding using, for instance, Linear Shift Registers (LFSR's). Elementary encoding of linear codes is achieved using the Generator matrix, which, as we saw in Ch. 2, is a $\mathbf{O}(N^2)$ operation. Structured LDPC codes are often designed to allow efficient encoding, while random codes must resort to the basic approach. However, certain efficient sparse-matrix operations do exist, allowing near-linear time encoding of LDPC codes, which would otherwise much less useful.

In this chapter, we will look at the process of encoding LDPC codes, maintaining our focus on random codes.

5.1 Matrix Encoding

In the most basic sense, any $[N, k]$ linear code, $\mathcal{C}$, may be encoded via an[1] $k \times N$ Generator matrix, G, consisting of k linearly independent rows. The corresponding $[N, N-k]$ dual code, $\mathcal{C}^{\perp}$, may be used for decoding (see Ch. 6), and the generator matrix for the null space[2] is known as a Parity Check matrix, H, of $\mathcal{C}$. Recalling that $GH^T = \vec{0} \mod 2$. In other words, any codeword, $\mathbf{x}$, of $\mathcal{C}$ is a linear combination of some subset of row vectors, $\vec{g}$, and, hence, any such product $\mathbf{x}H^T$ must equal 0. Any random k-bit information sequence may be encoded through multiplication with G. The major concern with LDPC codes

[1] As we have seen, there are several equivalent matrix representations–an ensemble–for any given code.

[2] As discussed in Ch. 2.

Definition 5 Elementary Row Operations [16] used in Gaussian Reduction.

1. (Replacement) Replace one row by the sum of itself and a multiple of another row.
2. (Interchange) Interchange (swap) two rows.
3. (Scaling) Multiply all entries in a row by a nonzero constant.

is then, simply, that the $\mathcal{O}(N^2)$ encoding complexity (matrix multiplication) is disproportional to the streamlined, linear-time SPA decoding. The schemes explored in this section all share this common bottleneck, and focus mainly on the preprocessing stage, in preparing the code for use in a system.

Again, we encounter real-life difficulties with the theoretically-poised random LDPC codes. When aiming at approaching capacity (Shannon limit) as closely as possible, one is less concerned with the details of the code that is defined by H, and perhaps more interested in proving some asymptotic behaviour of an ensemble. As we will see in Ch. 7, we do not need to know the code(space) in order to simulate Bit-Error Rate (BER) performance. The all-zero, N-bit vector is necessarily a valid codeword of any linear code (2.1), so one may skip the *encoding* process alltogether.

In the remainder of this chapter, we will have a brief look at some ways of encoding random LDPC codes. Consider a random, sparse $m \times N$ matrix, H, that is optimised for decoding. For encoding purposes, we need G', which we find via Gaussian Reduction (GR, Def. 5) on H, followed by the transformation (2.2). Any linear dependencies in H will then be 'neutralized' by GR, and moved to the bottom of H'.

Since linearly dependent rows in H may be removed entirely without changing the codespace, it is apparent that such dependencies are not part of defining the code. However, redundant protection means overdefined codes, which can be helpful in the decoding process. Consider this as added protection, in terms of Parity Check constraints. However, this preprocessing stage (GR) requires $\mathcal{O}(N^3)$. An important point is to maintain (row) equivalence between matrices at all times, otherwise the null space–and, the codespace–will change. For instance, columns of both G and H map to codeword bits, so we may not perform any column operations when reducing H' to standard form. However, this is only important if we wish to maintain the relationship (2.1) between H and G. In many cases, it is acceptable to change the codespace by performing the *identical permutations* to both H and G. This way, (2.1) is maintained.

5.1.1 Decoding in Standard-Form

Although reducing (GR) H to standard-form may seem to be (and often is) a textbook description of encoding linear codes, this approach is not well suited for use with Sum-Product (SPA) LDPC decoding. Say we were to use H' to construct our SPA decoder. From Ch. 4, we are aware of the puzzle it is to construct *good* LDPC codes; e.g., girth, density etc. Reduction to standard form (GR) is not a very 'clean' transformation, it does not take any precautions

before performing the row operations in Def. 5. Generally, the result is that the P-part of H' is quite dense, which, in turn, means lots of 4-cycles. Also, the identity-part, I_m, translates to m bit nodes of weight 1 (only connected to one Parity Check node). This means that there is a significant number of bits that are very weakly protected, meaning that the decoder is unable to produce a good bias on whether such a bit is 'correct' (or, conversely, if it should 'flip its value').[3] Fig. 5.1 show clear indications on the extremely poor results when decoding on H'; in particular, Fig. 5.1(a), where performance is worse than uncoded. Incidentally, this is the *converse* situation of the highly connected, 'elite bits' used to motivate irregular LDPC design in Ch. 4. In the systematic case, the non-systematic bits (weight > 1) may be seen as elite, and converge easier. However, in this case, they are unable to 'help' the systematic bits, which 'have no support.'

5.1.2 Appending I_m to H

One clean and efficient way to overcome the obstacles of row-reducing H is to simply alter the construction perspective. Using the EBF (or similar scheme), we construct only the P-part of H'. We may then augment the sparse, random $m \times N$ (sub)matrix to a $m \times (N + m)$ standard-form Parity Check matrix by simply appending I_m to P. Similar to the definition of G', we note how the identity part ensures that the resulting H' has full rank, $m = N - k$; regardless of the original rank of P. If desired, blocklength N may be preserved by simply designing P as an $m \times (N - m)$ matrix.

It is easily proven that the augmented matrix, H', maintains the important design characteristics of P, such as girth, density, and blocklength (as described above).

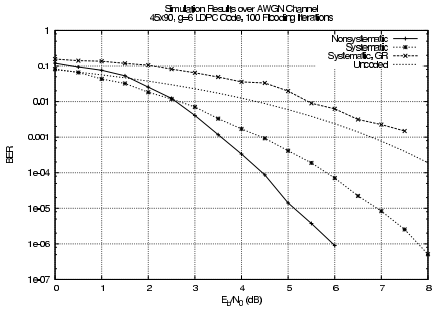
Starting with girth; consider the original, optimized construction, P, with girth $G(H)$. The 'extra' submatrix, I_m, consists of weight-1 pivot columns, each with a non-zero entry in an unique row (along the main diagonal). Considering the graph representation of H', no cycle of length $\geq G(H)$ can ever be extended by going via any systematic bit because each of these are 'dongles,' or dead-ends. In other words, no path entering a systematic bit can ever proceed further without going back along the same edge. Hence, it is not a cycle (as defined in Ch. 2), and the girth remains unchanged.

Where the density of the original matrix is approximately $\Delta_P = \frac{m\rho}{mN} = \rho/N$, the *very sparse* I_m reduces the density of the augmented matrix,

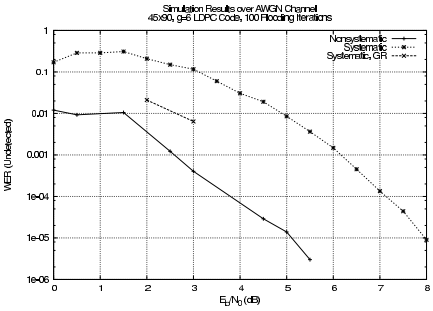
$$\Delta_{H'} = \frac{m(\rho+1)}{m(m+N)} = \frac{\rho+1}{m+N} \; < \; \Delta P. \tag{5.1}$$

Finally, there is the question of rank. Say the rank of P is $k_P < m$; i.e., P contains linear dependencies (not of full rank). As mentioned above, I_m incapacitates any linear dependencies in P, such that H' must be of full rank, even when P is singular. Changing k (to $k = m$) will redefine the code completely. However, the sparsity of P will usually result in $k_P = m$, such that this is not an issue. Otherwise, one may protect the underfull rank, k_P, by appending a 'smaller' identity matrix, I_{k_P} to P.

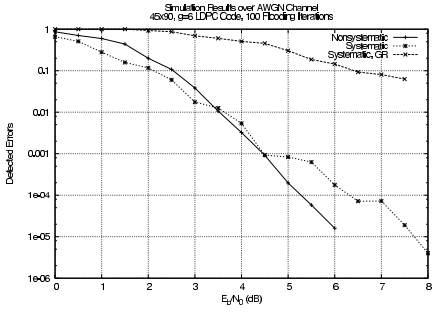
[3] See Fig. 6.1.



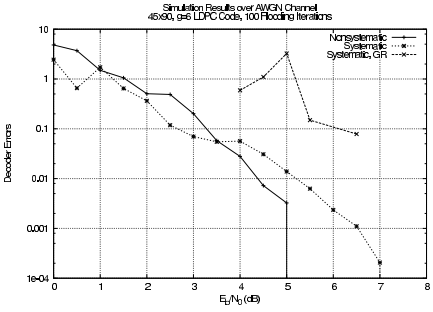
(a) Bit-Error Rate (BER).



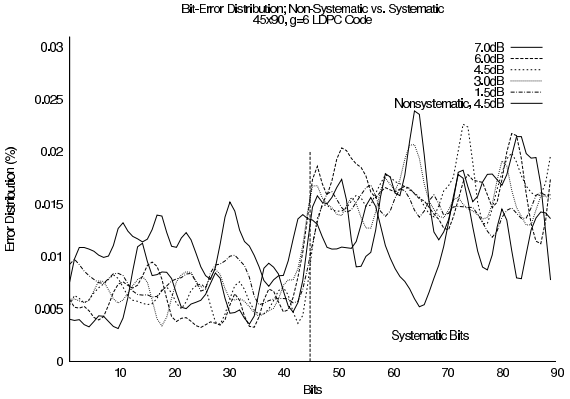
(b) Undetected Word-Error Rate (WER).



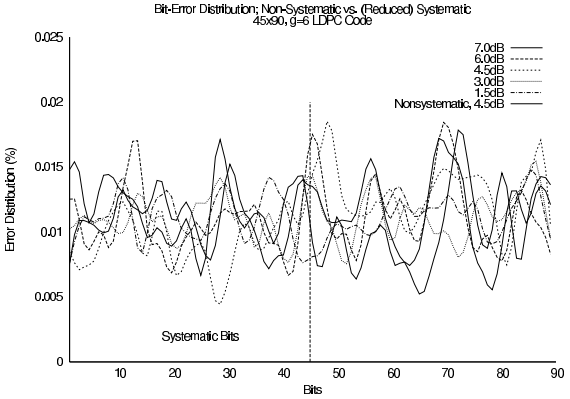
(c) Detected WER.



(d) Introduced (Decoder) Errors.



(e) Error-distribution.



(f) Error-distribution.

Figure 5.1: Systematic versus non-systematic.

Fig. 5.1 shows various simulation results comparing systematic and non-systematic versions of equivalent codes. Both $N = 90, k = m = 45$ (full rank), girth $g = 6$ codes were constructed using the EBF algorithm (Ch. 4), where the systematic code was constructed by augmenting an $m \times (N-m)$ non-systematic code with the identity matrix, I_m. The plots show an unmistakable tendency towards deterioration in the systematic case; already at BER 10^{-4}, Fig. 5.1(a) shows a *gain* of well over 1 dB. Fig. 5.1(e) reveals the decoder's incapability to correct errors in the systematic bits–note the uniform distribution in the non-systematic case. How this remainder of errors (after decoding) causes such a large gain, is evident from Fig's 5.1(b)-5.1(d). First, recall the correlation between *sparsity* and code minimum distance, $\mathrm{d_{min}}$. Fig. 5.1(b) shows an extreme increase in *undetected* word-errors; where the decoder converges to a valid codeword *other than* the correct, transmitted codeword. If $\mathrm{d_{min}}$ decreases–compressing the codespace, so to speak–it becomes more easy for the decoder convergence to be pulled into the 'gravity of nearby codewords.'[4] Fig. 5.1(c) show the detected word-error rate; where the decoder simply 'gives up' trying to converge to a valid codeword. The fact that these plots are quite similar, is actually further evidence of the change in $\mathrm{d_{min}}$, since this leaves only Fig. 5.1(b) (the undetected WER) as explanation for the gain. Finally, Fig. 5.1(d) shows the mistakes made during decoding; again, both codes seem to cause a similar amount of 'internal errors' (not caused by channel noise).

5.1.3 Standard LDPC Encoding

The basic method for encoding LDPC codes is quite similar to that described above. To protect the optimised features of the code, it is common to replace the identity matrix with a more well-defined Parity Check (sub)matrix, C_2, that is invertible[5] [32, 44]. We have,

$$H = [\, C_1 \,|\, C_2 \,], \tag{5.2}$$

where $C_1 \in GF(2)^{m \times k}$ and $C_2 \in GF(2)^{m \times m}$. Since C_2 is invertible, it must be non-singular, thus ensuring that the code has full rank, $rank(H) = N - k = m, k = N - m$. The major difference is that C_2 is not required to be the identity matrix, and that C_1 can be *any* random matrix, enabling greater freedom in constructing a code that is well suited for SPA decoding. Clever design allows us to calculate G' without altering H;

$$G' = [\, I_k \,|\, (C_2^{-1} C_1)^T \,]. \tag{5.3}$$

With (5.2) - (5.3), (2.1) holds, and we may decode on H (5.2).

If we allow *column swaps* in GR, it is much easier to produce H', and G'. Denote the ordered sequence of column-swaps performed on H (during GR) as π, producing the standard-form $\pi(H')$ which gives the corresponding standard-form $\pi(G')$. However, due to the column permutations, (2.1) no longer holds; $\pi(G')H^T \neq 0$. Bit-positions in codewords produced by encoding on $\pi(G')$

[4] As in the *sphere-packing bound*, e.g. [24].
[5] Non-singular and rectangular.

are permuted according to π, and these codewords do not satisfy the code-membership requirement (2.4). Hence, we must decode using $\pi(H')$; and not the original, well-defined H.

However, it is possible to 'undo' the column-swaps, such that the generator matrix produces codewords that belong to the original code–and H. By applying the *reversed* column-swap sequence, $\bar{\pi}$, we produce the 'near-standard-form' matrix, $\tilde{G}' = \bar{\pi}(\pi(G'))$, which 'matches' the original H (2.1);

$$\tilde{G}' H^T = \vec{0} \mod 2.$$

We may now encode using $\tilde{G}'$, and decode using the original, well-defined H. Since $\tilde{G}'$ is not in standard-form, and the k information-bits are permuted within the codeword, the decoder must store a k-bit vector identifying the systematic bits. This is similar to *interleaved* codes, and might increase the error-correction abilities when transmitting over a burst-error channel.

Consider the following example. The random matrix,

$$H = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{5.4}$$

turns out to be of full rank, but requires column-swaps to set it in standard-form. By performing the swaps $\pi = \{ (3,4), (4,5) \}$, we get $\pi(H')$, and

$$H = \begin{bmatrix} \mathbf{1} & 0 & 1 & 0 & 0 & 1 \\ 0 & \mathbf{1} & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & \mathbf{1} & 0 & 1 \\ 0 & 0 & 0 & 0 & \mathbf{1} & 0 \end{bmatrix} \rightarrow \pi(H') = \left[\begin{array}{cccc|cc} \mathbf{1} & 0 & 0 & 0 & 1 & 1 \\ 0 & \mathbf{1} & 0 & 0 & 1 & 1 \\ 0 & 0 & \mathbf{1} & 0 & 0 & 1 \\ 0 & 0 & 0 & \mathbf{1} & 0 & 0 \end{array}\right].$$

We may now produce the (permuted) Generator matrix using (2.2)

$$\pi(G') = \left[\begin{array}{cccc|cc} 1 & 1 & 0 & 0 & \mathbf{1} & 0 \\ 1 & 1 & 1 & 0 & 0 & \mathbf{1} \end{array}\right] \rightarrow \tilde{G}' = \begin{bmatrix} 1 & 1 & \mathbf{1} & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & \mathbf{1} \end{bmatrix}.$$

However, we are still left with encoding latency due to the $\mathcal{O}(N^2)$ matrix multiplication (2.3).

5.1.4 Efficient Encoding

Richardson *et al.* [45] have deviced a matrix-oriented encoding scheme which attempts to exploit the convenient sparseness of LDPC codes, and bring the complexity down to $\mathcal{O}(N)$.

In a preprocessing stage, the *approximate lower triangular form*, $\tilde{H}$ (our notation), is produced. This is defined as the partitioned H matrix (still $m \times N$ in total), where the upper-right submatrix, T, is in lower triangular form and the "gap," defined as the width of B and D, is as small as possible;

$$\tilde{H} = \left[\begin{array}{c|c|c} A & B & T \\ C & D & E \end{array}\right]. \tag{5.5}$$

By allowing row and column *permutations* only, the sparsity of $\tilde{H}$ is not degraded; unlike the case with regular GR. Without going into detail,[6] $\tilde{H}$ is processed further, such that the actual encoding process may be performed in an efficient manner. The codeword, $\mathbf{x}$, is produced as three separate vectors, where s is the k-bit information part;

$$\mathbf{x} = (s, p_1, p_2). \tag{5.6}$$

Parts p_1 and p_2 comprise the m-bit redundancy, and are now both computable using optimized, $\mathcal{O}(N + g^2)$ sparse-matrix operations described in [45]. Here, g is defined as the 'width' of the gap in (5.5).

[6] [45] contains a concise summary of the process.

Chapter 6

Sum-Product Decoding

In the Forward Error Correction (FEC) scenario, which we focus on in this thesis, we assume that there is no possibility of requesting retransmission of noisy messages. This one-way restriction is by far the greatest obstacle in achieving useful transmissions. For instance, in deep-space communications the data might be subject to only moderate disturbance, and should be expected to arrive relatively unscathed. However, the transmission time is so great that requests for retransmission might eventually reach a dead emitter. To compensate for this handicap, one typically accepts a somewhat higher latency (or delay) in FEC decoders.

In this chapter we will explore the implications of decoding from a strictly local perspective. Not only does this refer to the receiver in the one-way FEC scenario, but also to the internal structure of the decoder itself. Conventionally, due to the high complexity of decoders in general, most systems currently rely on software implementations [46]. The distributed structure of Factor Graphs permits the assembling of a Sum-Product Algorithm (SPA) decoder from local suboperations only, making LDPC decoders well suited for efficient, low power hardware implementation. This also improves on precision issues and numerical instabilities (such as buffer overflow), yet, a typical concern for on-chip design is the prohibitively intricate cross-wiring prescribed by the large high-girth H-matrix [47]. This presents a second challenge for designing good LDPC codes and decoders; how can we improve performance of practically sized codes. From chapter 4, we have already seen the design of optimized small LDPC codes using the Bit-Filling algorithm.

6.1 Maximum Likelihood Decoding

In its essence, the decoding problem amounts to finding the valid codeword which most resembles what was received at the channel output, $\mathbf{y}$. In other words, find the best estimate codeword, $\hat{\mathbf{x}}$, that satisfies the Maximum Likelihood (ML) condition;

$$\begin{aligned} \hat{\mathbf{x}} &= \max_{\mathbf{x}' \in \mathcal{C}} P(\mathbf{x}' = \mathbf{x} \,|\, \mathbf{y}) \\ &= \min_{\mathbf{x}' \in \mathcal{C}} \mathrm{d_H}(\mathbf{x}' - \mathbf{y}), \end{aligned} \tag{6.1}$$

where $\mathbf{x}$ is the original codeword, and $\mathrm{d_H}$ the Hamming distance between vectors. The ML-decoding strategy (MLD) guarantees optimal results[1], but requires exhaustive search through the vectorspace of the code. Since $|\mathcal{C}| = \mathcal{O}(2^N)$ [48], this is mainly of theoretical interest–as a benchmark against which to compare other, more practical strategies.

Another important concept is the distinction between hard and soft decoding. As mentioned in Ch. 3, it is helpful to take advantage of the extra information available in the real-valued channel output. Classical decoding will immediately quantize this information into 'hard values,' to which efficient modulo-2 operations apply (e.g, addition can be performed by simple XOR). By accepting the added complexity of handling soft information, one achieves an immediate coding gain of 2 - 3 dB.

6.2 Distributed Decoding on Factor Graphs

In the field of (very) long block codes, such as LDPC codes, one cannot construct decoders with complexity proportional to 2^N, so this renders MLD infeasible. By observing tree codes, Gallager observed that the decoding process could be split up into a network of suboperations, each performing only trivial computations. By allowing the decoding process to propagate through the tree, in one forward and one backward pass, each node (bit or check) would only need to communicate with its immediate neighbourhood. Hence, the overall complexity of decoding N bits drops from $\mathcal{O}(2^N)$ to $\mathcal{O}(N2^\rho)$, where ρ is a bound on the size of the input to suboperations. In this work, Gallager experienced optimum results with $(N, 3, 6)$-regular[2] LDPC codes; an ensemble that still remains valid today.

6.2.1 Syndrome Decoding

The prohivitively complex global problem of decoding could be factored–into a chain of manageable subproblems. One might find it helpful to view local configurations as minute subcodes, which are interconnected to form the larger, global code [7]. Valid codewords in $\mathcal{C}$ must satisfy *all* parity check constraints. The syndrome, $\mathbf{z}$, of an N-bit vector, $\hat{\mathbf{x}}$, (not necessarily a codeword) is an m-bit vector representing the *error pattern* that is most likely causing the failed decoding [49]. Using the Parity-Check matrix, H, we have

$$\mathbf{z} = H\hat{\mathbf{x}}^T, \tag{6.2}$$

where $\mathbf{z}$ may be used to locate the bits in error. Recall that we are working with *column* vectors (Ch. 2).

The all-zero syndrome indicates that a codeword is found, and–as we shall see–the Sum-Product Algorithm often realises a very good approximation of MLD (finding the codeword nearest, in Hamming distance $\mathrm{d_H}$, to the transmitted $\mathbf{x}$).

[1] On the assumption that few errors are more likely than many errors.

[2] Recall that this is shorthand notation for node degrees, γ and ρ, for bits and function, respectively.

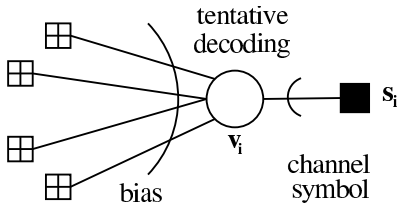


Figure 6.1: The noisy channel symbol from the input bit is adjusted by the bias of the local constraint nodes. Hence, the tentative decoding is contained in bit nodes, and its *protection* is proportional to the size of its support, $|n(v)|$.

Returning to the discussion and the example in Ch. 2, we have that the characteristic function, $\chi_{\mathcal{C}}(\mathbf{x})$, conveniently splits into the logical conjunction (or product) of the individual Parity Check constraints, $c_0, c_1, \ldots, c_{m-1}$, each working only on a subset, A_i, of the N bits [22]

$$\chi_{\mathcal{C}}(v_0, v_1, \ldots, v_{N-1}) = c_0(A_0) \wedge c_1(A_1) \wedge \cdots \wedge c_{m-1}(A_{m-1}). \tag{6.3}$$

Since rate $R = k/N = (N-m)/N \leq 1$, we have that $m < N$ and, by necessity, these subsets A_i will overlap. In a tree graph, however, this overlap is limited to any bit being part of a maximum of 2 parity check constraints. In other words, there are no back-edges or cycles in the tree. This distributed solving of interdependent subfunctions is generally referred to as "marginalized product-of-functions" (MPF), which has been found to be essential in the operation of a wide variety of algorithms. The marginalization principle is essential to the reliability of the final solution. Consider a received, noisy vector. The conservative–and quite reasonable–view is that any given bit *is* in error; or, at best, unreliable. As such, we wish to avoid bits "trusting" their current assumptions on what is their correct value, and rather have them work exclusively with the summary (product) of the information received from the *other* parts of the system–see Fig. 6.1. This extrinsic principle (Ch. 2) serves as a variant of majority logic, where individual bits conform to the influence of their local neighbourhoods. The distributed structure of a (factor) graph is well suited to store such input (and output) distributions of maginalized (extrinsic) messages on the edges connecting any pair of nodes.

Gallager's acyclic codes (i.e., tree structure) ensure that one can always find some node whose parents are already processed. Hence, all required messages to this node would be pending on the input edges, enabling this node to be processed next. By starting the iteration from leaf nodes, one can unambigously traverse the entire network in this step by step manner. After completing the message passing in both directions, the results of the local computations can be read off at the corresponding bit nodes. In the acyclic case, it can be shown that the end results are identical to the output of the global problem.

6.2.2 Cyclic Factor Graphs

As established in Ch. 4, the design of good LDPC codes is a challenging task of balancing girth against the assumption that tree codes have poor performance (low rate, low minimum distance). In any event, one is forced to deal with a nat-

Construction 4 Gallager's "Bit-Flipping" Algorithm [4].

For each bit node, v_i, count the number of *unsatisfied* Parity Check nodes to which it is connected, e_i. If all checks are satisfied, end decoding. Otherwise, flip the value of each bit that is part of more than some threshold, δ, of unsatisfied checks; i.e., where $e_i > \delta$. Repeat the process.

ural presence of cycles in the associated graph. In terms of the MPF problem, cycles represent undesirable dependencies among variables, making it impossible to marginalize "cleanly." This feedback interferes with the MPF principle of working with extrinsic information only. From the decoding perspective, this means that an erroneous bit indirectly will influence the bias from its support (via the cycle). However, the 'strength' of this feedback does become 'watered out,' as the information is moderated (within bits) underway around the cycle. This is one way of explaining why shorter cycles are more damaging than larger cycles. However, although (acyclic) tree codes have optimum performance in SPA (no feedback), the application of these codes is limitied due to the more fundamental problem of their low minimum distance.

Regardless of feedback, experience has concluded that MPF performs surprisingly well on typical, cyclic LDPC codes. In a cyclic graph (i.e., non-tree), individual codebits are checked by > 2 different Parity Checks. Such dependencies of variables (bits) were initially expected to have a corruptive effect on the accuracy of the MPF algorithm. However, as experiments concluded, the procedure is extremely robust. By only avoiding the most severe feedback (cycles of length ≤ 4), end results are found to converge to the correct value with quite acceptable precision [4].

It should also be pointed out that in decoding, as opposed to, say, artificial intelligence networks where MPF is used to perform belief propagation, one is already working in a noisy environment, and is less sensitive to imprecision in calculations. After soft, iterated decoding, a hard-decision (quantization to binary bits) of the MPF output is required to produce the decoded data block. Hence, a strong inclination ($P(e) > 50\%$, entropy < 1) towards the correct value is sufficient to decode a bit.

6.3 Sum-Product Algorithm

In [4], two strategies were introduced; one hard and one soft. The former has been dubbed "Bit-Flipping" [17] (Constr. 4), while the latter is the now popular Sum-Product Algorithm (SPA).[3] By observing the duality of soft values and probabilities, Gallager designed a "probabilistic decoder" which solves the MLD problem using the MPF algorithm discussed above.

6.3.1 Iterative Decoding

Also in the cyclic case, one still uses exactly the same SPA description, with a minor change in scheduling. Whereas 'regular' MPF will terminate naturally

[3] Like Gallager codes, this algorithm was forgotten and rediscovered several times, within various areas of research. See [23] for an excellent survey on SPA decoding of LDPC codes.

after two messages have been passed along each edge (one in each direction), we here need to repeat the process, during which a tentative decoding, $\hat{\mathbf{x}}^{(l)}$, is successively updated.

The maximum number of allowed iterations, T, is called the timeout of the decoder. The aim is to converge to a fixed, stable state (within $l < T$ iterations) in which each tentative bit, $\hat{x}_i^{(l)}$, resembles the corresponding original bit, x_i, with some confidence; $p \neq 0.5$. In this case, hard decisions on $\hat{\mathbf{x}}$ will produce the correct output, $\mathbf{x}$. If $\hat{x}^{(T)}$ is not a valid codeword, a decoder error is declared.

An important observation is that iterated decoders do not necessarily benefit from increased time, T. Soft values tend to converge to a stable (while not necessarily valid) state after relatively few iterations.[4] Only in some particular cases will the entropy fail to drop, as a result of the decoder being stuck in a repeated, oscillation between two distinct (invalid) states. In neither of these cases will the decoder improve if given more time. Such measurements are efficiently conducted using the technique of Density Evolution [31] mentioned in Ch. 4.

6.3.2 Initialization: Demodulation

The task of SPA decoding is to maximise the *a posteriori* probabilities (APP's) of individual bits having value '1' (or, conversely, '0'), given the channel output, $\mathbf{y}$. Recalling section 3.3.1, for our purposes, codebits are modulated to BPSK[5] and subject to AWGN noise before they are received at the decoder. Conveniently, the noisy symbol y_i can be viewed as a *likelihood* measure of the corresponding original bit, v_i, being '1' or '0'.

AWGN noise has the effect of offsetting symbols to within a distance proportional to the current noise-level, σ^2. This follows a normal probability distribution (PDF 'bell curve'). As illustrated in Fig. 3.3, at high noise levels (i.e., low SNR) these offsets overlap, such that the crossover probability becomes non-zero, and errors can occur.[6]

Given the channel SNR, we use (3.11) to calculate the variance, σ^2, which, in turn, determines the shape of the two PDF curves (3.10). The curves are then offset according to the modulation (BPSK), hence the medians are $\mu = \pm E_s$. Now, the APP's are simply 'read off' the opposing curves; algebraically, this is

$$\begin{aligned} p_i^{(1)} &= P(c_i = 1) = P(x_i = -1 \,|\, y_i) = f(y_i, \mu = -1), \text{ such that} \\ p_i^{(0)} &= 1 - p_i^{(1)}. \end{aligned} \tag{6.4}$$

Note that $p_i^{(0)}$ may also be calculated from the PDF by changing the median to $\mu = +E_s$. This is an important fact when working with non-binary alphabets. Finally, the probabilities are normalized by scaling each value by

$$1/(p_i^{(0)} + p_i^{(1)}), \tag{6.5}$$

which also generalizes nicely to the non-binary case.

[4] As reported by MacKay [50], and confirmed by our results in Ch. 7.

[5] BPSK mapping is $1 \rightarrow -1$ and $0 \rightarrow +1$, assuming $E_s = 1$.

[6] Obviously, one could move the code symbols further apart, thereby decreasing the chance of overlap (increased tolerance to noise), but this comes at the cost of increased energy usage per symbol, E_s. See [17, 51, 52] for a discussion on Coded Modulation.

6.3.3 Messages

From a Factor Graph point of view, each bit node, v, is connected to a set of check nodes, $n(v)$, called the support of v. Nodes (bits and checks) compute an extrinsic (SPA is MPF) outbound message for each edge, based on some local transformation on the net incoming messages. The message from a bit node to a check,

$$\mu_{v\to c} = (p_v^{(0)}, p_v^{(1)}). \tag{6.6}$$

is the current local value (or, state) at v, conveying the 'assumption of v' on its parity. Similarily, the opposite message, $\mu_{c\to v}$, is a bias on the correctness of v, as computed by c. The check node attempts to adjust the state of 'its bits,' so that it may be satisfied (XOR $= 0$).

In essence, messages are probability distributions. In this chapter (and the rest of the thesis), we will only consider the case where all variables are binary.[7]

Although it is perhaps most intuitive to work directly with these probability distributions (6.6), the Sum-Product Algorithm is typically implemented in a more 'convenient' mathematical domain. This has two beneficial effects; firstly, memory usage is cut in half by compressing variables into a one-dimensional likelihood ratio (LR),

$$\lambda(\,\mu_{u'\to u}\,) \triangleq p_{u'}^{(0)}/p_{u'}^{(1)}. \tag{6.7}$$

Second–as we will look at in the following–more efficient (in terms of CPU-usage and time) internal operations exist in other domains. However, in software, obvious numerical concerns (buffer overflow, division-by-zero) arise when dividing by values that might very well approach 0 (as $p_0 \to 1$). This weakness is overcome by clipping extreme values, as suggested in [3] by $10^{\pm 5}$–especially immediately prior to division. Another popular approach is to work with log-likelihood ratios (LLR's),

$$\Lambda(\,\lambda\,) \triangleq \ln(\lambda). \tag{6.8}$$

In the logarithmic domain, extreme values are naturally scaled by the ln function.

Before decoding, all messages in the FG (except the input messages) are initialized to neutral values which, in terms of probabilities, is $(0.5, 0.5)$. Similarily, the neutral LR is 1, and the neutral LLR is 0.

6.3.4 Function Types

Although it is conventional to describe the Sum-Product algorithm in terms of two separate update rules–one for check nodes, and one for bit nodes–we want to stress the simplicity of the algorithm by describing one, generic rule without reference to type. In the following, we will assume all variables to be binary, yet, we point out that it is straight-forward to extend the rule to variables of arbitrary dimension.

Consider a binary function, $f : GF(2)^i \mapsto GF(2)^o$, performing some mapping from i input variables to o output variables; where the total number of

[7] When, for instance, implementing the SPA variant of Turbo (Viterbi) decoding, q-ary auxiliary variables are required to link nodes containing trellis sections–see [22, 53].

Θ_4							
Input			Output		Output		
v_0	v_1	v_2	v_3	τ_{XOR}	$\vec{\phi}_{4\setminus\{v_3\}}$		
0	0	0	0	1	$0.3 \cdot 0.9 \cdot 0.2$	$\cdot 0.5$	$q_3^{(0)}$
1	0	0	0	0	0		
0	1	0	0	0	0		
1	1	0	0	1	$0.7 \cdot 0.1 \cdot 0.2$	$\cdot 0.5$	
0	0	1	0	0	0		
1	0	1	0	1	$0.7 \cdot 0.9 \cdot 0.8$	$\cdot 0.5$	
0	1	1	0	1	$0.3 \cdot 0.1 \cdot 0.8$	$\cdot 0.5$	
1	1	1	0	0	0		
0	0	0	1	0	0		$q_3^{(1)}$
1	0	0	1	1	$0.7 \cdot 0.9 \cdot 0.2$	$\cdot 0.5$	
0	1	0	1	1	$0.3 \cdot 0.1 \cdot 0.2$	$\cdot 0.5$	
1	1	0	1	0	0		
0	0	1	1	1	$0.3 \cdot 0.9 \cdot 0.8$	$\cdot 0.5$	
1	0	1	1	0	0		
0	1	1	1	0	0		
1	1	1	1	1	$0.7 \cdot 0.1 \cdot 0.8$	$\cdot 0.5$	

Table 6.1: Θ_4, $p = 4, o = 1$, truth table τ_{XOR}, and an example calculation of $\mu_{u \to v_3}$.

variables is $i + o = p$. Define the matrix Θ_p as that consisting of all 2^p binary value-assignments across the variables; listed in lexicographical ordering. The *indicator function* of f, $\mathcal{I}_f$, is defined as a mapping from p variables (input and output to f), to one Boolean value (typically, using $\{0, 1\}$) indicating the validity of the input/output combination; $\mathcal{I}_f : GF(2)^p \mapsto GF(2)$ [54]. By applying $\mathcal{I}_f$ to each individual row of Θ_p, we get the length-2^p vector, τ_f, whose non-zero positions identify all valid assignments of variables–in the same ordering as Θ_p. Where $\vec{\theta}_j$ is row j of Θ_p, we have

$$\begin{aligned} \tau_f(j) \neq 0 &\Leftrightarrow \mathcal{I}_f(\vec{\theta}_j) \neq 0 \\ &\Leftrightarrow f(\theta_{j,0}, \theta_{j,1}, \ldots, \theta_{j,i-1}) = (\theta_{j,i}, \theta_{j,i+1}, \theta_{j,i+o-1}). \end{aligned} \tag{6.9}$$

While invalid assignments are always zero (6.9) for the most general type of indicator function, mapping to the reals, the valid (non-zero) assignments do not necessarily occur with uniform probabilities. Hence, in the general case, the range of τ_f is the positive real numbers; where $\tau_f(j)$ can be thought of as the probability of the corresponding input producing the corresponding output. As the indicator vector is essentially a probability distribution in the codewords set, the vector is normalized such that $\sum_j \tau_f(j) = 1$.

Consider the XOR function on $p = 4$ variables, as shown in Table 6.1. By augmenting Θ_4 with the extra column τ_{XOR}, we get a *look-up* table describing the XOR function.[8] In the decoding setting, the code is partitioned into smaller subcodes (around each Parity Check node), which are then decoded seperately. As such, the input in Θ_4 is the space of possible codewords, where τ_{XOR} indicates

[8] XOR is defined by $1 \oplus 1 = 0$.

those that are valid according to this subcode. As part of the definition of linear codes, all codewords are equally likely to occur, which is seen by all non-zero entries of $\mathcal{I}_f$ being identical. Moreover, as the indicator is a probability distribution, $\mathcal{I}_f$ is also normalised by 1/8 (not shown).

Table 6.1 also shows an example calculation of the extrinsic output message for v_3; we will discuss this in the next subsection.

6.3.5 Generalized Update Rule

In this section, we will derive an update rule that does not depend on the local function. By expressing the mappings as an indicator function, (6.9), we may use the following rule to compute the output of any (linear) function.

Probability distributions are vectors of messages, so the input distribution to node u is

$$\vec{r}_u = (\, \mu_{v_0 \to u},\, \mu_{v_1 \to u},\, \ldots,\, \mu_{v_{(p-1)} \to u}\,), \tag{6.10}$$

where v_i is a neighbour of u. Regarding the above discussion on domain, we will initially consider working directly with probability distributions, where each message is a set of (binary) APP's (6.4)

$$\mu_{v \to u} = (\, p_v^{(0)},\, p_v^{(1)}\,).$$

$\vec{r}_u$ can be expanded into the vector $\vec{\phi}_p$, in which the input *values* (APP's or ratios) are combined to produce the compound probability of each possible input/output assignment. $\vec{\phi}_p$ is computed by taking the tensor product of the elements of $\vec{r}_u$,

$$\vec{\phi}_p = \bigotimes_{i=0}^{p-1} r_{u,i} = (\, \mu_{v_0 \to u} \otimes \mu_{v_1 \to u} \otimes \cdots \otimes \mu_{v_{(p-1)} \to u}\,). \tag{6.11}$$

When using (6.11) to produce the o output messages, we must ensure that the important extrinsic principle of SPA is obeyed. Neutralizing the contribution of a variable, v_i, on its own output message, is a matter of replacing that input, $\mu_{v_i \to u}$, with the neutral message, which–in terms of APP's–is $(\,0.5,\, 0.5\,)$. Define

$$\vec{\phi}_{p\setminus\{v_i\}} = (\, \mu_{v_0 \to u} \otimes \cdots \otimes \mu_{v_{i-1} \to u} \otimes (\,0.5,\, 0.5\,) \otimes \mu_{v_{i+1} \to u} \otimes \cdots \otimes \mu_{v_{(p-1)} \to u}\,). \tag{6.12}$$

Returning to the example in Table 6.1, we have the input distribution $\vec{r}_u = (\,(0.3, 0.7), (0.9, 0.1), (0.2, 0.8), (0.6, 0.4)\,)$. The calculations for v_3 is shown in column $\vec{\phi}_{4\setminus\{v_3\}}$.

The output for v_i is computed by summing the products in $\vec{\phi}_{p\setminus\{v_i\}}$–hence the name, *Sum-Product* Algorithm. Obviously, the return message to v_i has the same dimension as v_i itself; that is, consisting of $k = \dim(v_i)$ fields. The output column in Θ_p identifies how to produce those fields, by summing certain entries of $\vec{\phi}_{p\setminus\{v_i\}}$ (i.e., marginalizing on v_i). This means that the values in the output column, $\vec{\phi}_{p\setminus\{v_i\}}$, are marginalised according to the indicator vector, τ_f

$$q_{v_i}^{(k)} = \sum_{j\,:\,\tau_j = k} \phi_{p\setminus\{v_i\}}(j)\,, \quad \forall\, k = 0, \ldots, \dim(v_i) - 1. \tag{6.13}$$

In Table 6.1, field $q_{v_i}^{(1)}$ in the (in this case, binary) output message corresponds to the sum of all output values for which the indicator column, τ_{XOR} equals 1.

Hence, the APP becomes

$$\mu_{u\to v_i} = (\, q_{v_i}^{(0)},\, q_{v_i}^{(1)},\, \ldots, q_{v_i}^{(k-1)}\,), \quad k = \dim(v_i) \tag{6.14}$$

As a final step, when working with APP's, we must ensure that each output message (6.16) is a probability distribution in its own right (i.e., $\sum_k q_{v_i}^{(k)} = 1$). By scaling each message with a normalization factor,

$$\delta_{v_i} = (\sum_k q_{v_i}^{(k)}\,)^{-1}, \quad k = \dim(v_i) \tag{6.15}$$

we have the final output APP,

$$\mu_{u\to v_i} = (\, q_{v_i}^{(0)},\, q_{v_i}^{(1)},\, \ldots, q_{v_i}^{(k-1)}\,)\,\delta_{v_i}, \quad k = \dim(v_i) \tag{6.16}$$

The total output distribution, $\vec{q}_u$, similar to (6.10),

$$\vec{q}_u = (\, \mu_{u\to v_0},\, \mu_{u\to v_1},\, \ldots,\, \mu_{u\to v_{(p-1)}}\,),$$

is produced by repeating the above process (6.12) - (6.16) for each output variable. The local value (state) of the node, $\omega_u^{(i)}$, is the normalized product of all incoming messages,

$$\omega_u^{(i)} = \prod_{v\in n(u)} q_v^{(i)}\ , \quad \forall\, i = 0, \ldots, \dim(u) - 1. \tag{6.17}$$

An example of the calculations for v_3 is shown in Table 6.1. Table $\Phi_{4\setminus\{v_3\}}$ is not shown, but note how the summation (6.13) is done according to output column 3. The (normalized) output message is $\mu_{u\to v_3} = (\,0.596,\, 0.202\,)\cdot 2.000 = (\,0.596,\, 0.404\,)$. Also, the local value of u is $\omega_u = (\,0.3\cdot 0.9\cdot 0.2\cdot 0.4\,, 0.7\cdot 0.1\cdot 0.8\cdot 0.6\,) = (\,0.0216\,, 0.0336\,)\cdot 18.12 = (\,0.391\,, 0.609\,)$. If u is a bit node, the tentative decoding of this bit, $\hat{x}_u^{(l)}$, is updated by quantizing (hard decision) on ω_u.

For reference, the conventional SPA description [22] is

variable to local function:

$$\mu_{v\to f} = \prod_{f'\in n(v)\setminus f} \mu_{f'\to v} \qquad \text{and,} \tag{6.18}$$

local function to variable:

$$\mu_{f\to v} = \sum_{\sim\{v\}} \left(f(V) \prod_{v'\in n(f)\setminus\{v\}} \mu_{v'\to f} \right), \tag{6.19}$$

where v is variable (bit), and f is a generic function node, as defined with $V = n(f)$ as input. The notation $\sim\{v\}$ refers to the summation (marginalization) stage for output to v, (6.13). Note that the distinction between the two rules is strictly unnecessary, as (6.18) is merely a simplification of the (generic) rule (6.19); with "the unit function [22]," i.e. where $f(V) = 1$ (normalised).

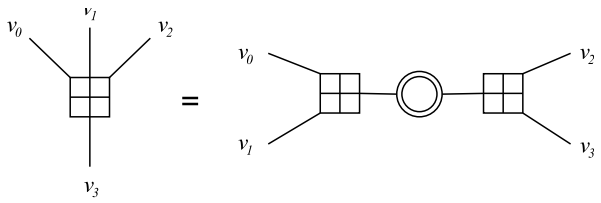


Figure 6.2: Equivalent Factor Graph representations of the XOR_4 function. The double-circled node is an auxiliary 'state-node,' containing only the end result of the chaining.

6.3.6 Optimized SPA for Decoding

The generalized update rule should be thought of primarily as a confirmation on the fact that SPA (or MPF in general) can be applied to a wide range of detection and estimation problems. If u is a function node, then (6.12) is exponential in the number of variables connected to u, and SPA is generally not viable in this "basic form." Consider, for instance, the delay introduced in decoding a medium-sized factor graph with nodes of degree ~ 20.

As implied earlier, various optimizations exist to simplify the calculations (LR's, LLR's), however the vital trick is to avoid working with the entire $\mathcal{O}(2^p)$ space of value-assignments, Φ_p.

Applying the generalised rule is simply a matter of inserting the desired local function as a truth table. In addition to the XOR function already discussed, only one other function is required when SPA is used for LDPC decoding; namely, the Equality Constraint (EQ). Keeping with the terminology of the previous sections, we describe the simple function by its truth table. Assuming $p = 4$ (binary) variables, the length 2^4 vector has only 2 non-zero entries,[9]

$$(\,\tau_{EQ}\,)^T = (\,1, 0, \ldots, 0, 1\,), \tag{6.20}$$

corresponding to value-assignments $00\ldots0$, and $11\ldots1$. Hence, $\mathcal{O}(2^p)$ resources are wasted in the process. The simple EQ function of p variables can be reduced to

$$q_{v_i}^{(k)} = \prod_p p_{v_i}^{(k)}\,, \quad \forall\, k = 0, \ldots, \dim(v_i) - 1, \tag{6.21}$$

followed by the normalization step of (6.16).

Similar observations lead to the optimization of the p-variable XOR function. The structure of Θ_p (note gray decomposition lines in Table 6.1) is extremely regular, and, as a result, highly decomposable. Consider extending Θ_p to Θ_{p+1}. Since Θ_p already exhausts all possible assignments over the previous, p variables, we simply append the first value of the new variable to the table, and repeat the process (this time, on a *copy* of Θ_p) with the next value.

The simple logic of the XOR function applied to such structure gives an obvious reduction in complexity. Rather than using $\mathcal{O}(2^p)$ calculations to compute XOR_p (recall the example previously, for $p = 4$), we may use $\mathcal{O}(2^{p-1})$

[9]This, again, is straight-forward to extend to non-binary variables.

resources to calculate XOR_{p-1}, and simply *combine* those subresults, using near-constant-time XOR_3 (with $i = 2, o = 1$). Continuing in this fashion, we have an exponential *increase* in efficiency (drop in complexity), bottoming out as a chain of $p-1$ XOR_3 operations; for a total complexity of $\mathcal{O}(p2^3) = \mathcal{O}(p)$. This chaining effect is illustrated in Fig. 6.2.

Following [22], XOR_3 essentially performs the following calculations,

$$\mathrm{XOR}(\,\mu_{v\to u}, \mu_{v'\to u}\,) = (\,p_v^{(0)}p_{v'}^{(0)} + p_v^{(1)}p_{v'}^{(1)}\,,\, p_v^{(0)}p_{v'}^{(1)} + p_v^{(1)}p_{v'}^{(0)}\,), \tag{6.22}$$

which produce the output message $\mu_{u\to v''}$. This is exactly the calculation performed by the generalized rule on $p = 3$ variables, with $(\,\tau_{\mathrm{XOR}}\,)^T = (\,1,0,0,1,0,1,1,0\,)$. This shows that the generalized update rule is still the 'main engine' underlying any optimized, "factorized" SPA implementation.

Fig. 6.2 shows how such optimizations can be coded into the structure of the Factor Graph, such that one need not modify the generalized update rule. When working with hardware realizations, this means one only needs the standard, 3-input XOR logic gate to build any decoder. Although this certainly requires a larger areal on the chip (which may have its own negative implications), the chaining simultaneously alleviates congestion in highly connected areas.

In software, however, it is more efficient to implement the FG according to the description of the original H-matrix, and rather perform 'virtual chaining,' during the update-call on nodes.[10] In example, the XOR_p ($p > 3$) update may be processed sequentially, two variables at a time (resulting in a third, auxiliary variable), using *in-order* ordering; starting with v_0 and v_1,

$$\mathrm{XOR}_p = \mathrm{XOR}_3(v_{p-1}, (\mathrm{XOR}_3(v_{p-2}, \ldots (\mathrm{XOR}_3(v_1, v_0))\ldots).$$

6.3.7 Likelihood Ratios

While the important optimization is already achieved, it is possible to gain some further improvement by replacing probability distributions (APP messages) with one-dimensional (unary) likelihood ratios (LR's) (6.7)

Considering only the unitary, 'chainable,' two-input one-output update rules (as discussed above), the LR rules become [22],

$$\mathrm{EQ}(\,\lambda_1, \lambda_2\,) = \lambda_1\lambda_2 \tag{6.23}$$

$$\mathrm{CHK}(\,\lambda_1, \lambda_2\,) = \frac{\lambda_1\lambda_2 + 1}{\lambda_1 + \lambda 2}, \tag{6.24}$$

for variable (bit) and function node, respectively.

Although it is possible to work directly with LR's, potential numerical vulnerabilities in software suggests avoiding this domain. Since $\ln(a \cdot b) = \ln(a) + \ln(b)$, switching to the log-domain means multiplication operations are replaced by addition, which is extremely suitable for software implementation. The update rules for LLR's (6.8) translate to

$$\mathrm{EQ}(\,\Lambda_1, \Lambda_2\,) = \Lambda_1 + \Lambda_2 \tag{6.25}$$

$$\begin{aligned}\mathrm{CHK}(\,\Lambda_1, \Lambda_2\,) &= \ln(\cosh((\Lambda_1 + \Lambda_2)/2)) - \ln(\cosh((\Lambda_1 - \Lambda_2)/2)) \\ &= 2\tanh^{-1}(\tanh(\Lambda_1/2)\tanh(\Lambda_2/2)).\end{aligned} \tag{6.26}$$

[10] In reality, a 16-bit look-up table (i.e., XOR_{16}) would be more convenient.

Fortunately, the somewhat complex LLR CHK rule (6.26) can be approximated[11] by the extremely efficient rule [22]

$$\begin{aligned}\mathrm{CHK}'(\,\Lambda_1,\Lambda_2\,) &\approx |(\Lambda_1+\Lambda_2)/2| - |(\Lambda_1-\Lambda_2)/2| \\ &= \mathrm{sgn}(\Lambda_1)\,\mathrm{sgn}(\Lambda_2)\,\min(|\Lambda_1|,\,|\Lambda_2|). \end{aligned} \tag{6.27}$$

The reduction in precision is acceptable since, as discussed before, we are already working in a noisy environment. Interestingly, (6.27) is the update rule of another MPF algorithm–the Min-Sum Algorithm [23, 55]–which is also used for decoding. In our simulations, in Ch. 7, we compare the efficiency of the LR and the LLR domains.

Switching from one domain to another requires only a few changes to the SPA implementation. First, during initialization, we must convert probabilities (APP's) to ratios. Since this is the most likely source of error (buffer overflow, division-by-zero), it is reassuring that this happens only once during decoding. Hence, APP's are converted to LLR's (or LR's) using (6.8) (or (6.7)). Also, the neutral messages–originally $(0.5, 0.5)$–become 0 (or 1, for LR).

Second, the procedure for quantizing (hard decision) soft values depends on domain, so we define the (one-way[12]) transformation $Q : \mathbb{R}^{\dim(u)} \mapsto GF(2)$,

$$Q(x) = \begin{cases} 1 & \text{iff} \quad x \geq 0.5 & \text{and } x \text{ is } (p_1\text{-part of) APP} \\ 1 & \text{iff} \quad x \geq 1 & \text{and } x \text{ is LR} \\ 1 & \text{iff} \quad x \geq 0 & \text{and } x \text{ is LLR} \\ 0 & \text{else} \end{cases} \tag{6.28}$$

In the remaining, we will assume an optimized implementation, such that all update calls are of constant-time complexity.

6.3.8 Scheduling

When cycles are present in the factor graph, the SPA becomes an iterated algorithm. Working with the convergence of soft messages in a multiplicative (or additive, for LLR's) procedure does not require any particular schedule on the order in which nodes are updated, and many different schemes have been explored. The scheduling used defines the work done per iteration.

Most common is the *flooding schedule*, where one iteration consists of the separate updating of each *type* of node. In the "conventional LDPC" case, with bits and checks, one iteration corresponds to first updating all checks, followed by the updating of all bits–or, *vice verca*. All nodes are fed fresh information in every iteration, in a manner which is extremely well suited for parallel implementation. However, since this schedule propagates messages at maximum rate through the graph, it is very sensitive to feedback in the form of (short) cycles. If the girth of the graph is g', the independence of messages is distorted after only $g'/2$ iterations. Using this scheduling, SPA executes $N+m$ constant-time update calls per iteration. As will be seen in Ch. 7, the

[11] Because, for $x \gg 1$, $\ln(\cosh(x)) \approx |x| - \ln(2)$.

[12] Obviously, information is lost during quantizing; this is the argument for the gain in soft decoding. In SPA, we only quantize *after* decoding.

average number of required iterations is normally bounded by $\log N$, so the total complexity of decoding is $\mathcal{O}(N)$.

In this project, we explore some novel SPA decoder schedules. These are described and tested in Ch. 7.

6.3.9 Stopping Criterion

As iterative decoding proceeds, one desires the BER to drop with each iteration. However, this improvement simultaneously flattens out, ideally towards a stable state–convergence. A cruicial feature of any iterative process is to know when the gain in proceeding drops below what is worth the effort of doing so; in other words, when to stop. This is a question which requires good insight into the algorithm, and does not always have a definite answer. Some error patterns send the decoder into continuous oscillation between two states, during which the change in entropy does not drop (it mainly changes sign). The 'decoder trajectory' can be plotted as an Extrinisic Information Transfer (EXIT) chart depicting the asymptotic behaviour of the decoder [17]. The numerical analysis bears resemblance to Density Evolution, as discussed previously.

Conventionally, the two major iterated decoders–the Viterbi-like SOVA (or SISO) Turbo decoders versus the Sum-Product algorithm–differ in the latter's ability to stop decoding early. By monitoring the state (tentative decoding) of the decoder, the process is stopped as soon as certain conditions are satisfied. This is a significant advantage of SPA, adding to the popularity of LDPC codes.

In SPA decoding, two main stopping criterions exist–one hard, and one soft decision. Normally, after each iteration, the syndrome (6.2) of the tentative decoding is computed, $\mathbf{z} = H\hat{\mathbf{x}}^T$. The all-zero syndrome indicates–with good confidence–that the Maximum-Likelihood (MLD) codeword is found. In our distributed SPA implementation, it is necessary to perform this check *locally*, within each check node, c. Since we do not correct errors in the quantized (hard decision) domain, we do not need the actual error pattern, $\vec{z}$; but only the net result–"is $\mathbf{z}$ the all-zero (no-error) pattern?" By polling the hard-decision value (6.28) of the state, $Q(\omega_u)$, of bit nodes a decision is made on whether each individual syndrome bit, z_i, is satisfied; i.e., if the sum of incoming messages to c has even parity. The final stopping decision is made as soon as all m checks are satisfied; i.e., in terms of an indicator function

$$S = \bigwedge_c [\omega_c = 0] = 1 \tag{6.29}$$

On the other hand, decoding may be stopped when the overall entropy, $H(\mathbf{x}) \triangleq -\sum_v p_v^{(1)} \log_2(p_v^{(1)})$, of the system drops to zero [56]. In this case, the information has converged to a stable state, from which it might not proceed.

6.3.10 Comments

The Sum-Product Algorithm provides the required efficiency to decode very long blocklength codes, such as LDPC codes. In this chapter, we have seen how the complex overall problem of syndrome decoding is reducible into a network (FG) of Parity Check and Equality Constraint (bit) nodes, where each local neighbourhood is responsible for processing only a subcode. This distributed approach may also be further refined, owing to the easily factorisable nature

of the XOR function, making LDPC decoders ideal for low-power, hardware implementation (cellphones, handheld devices).

SPA is already used in areas other than decoding, and, although beyond the scope of this thesis, we expect the possibility of solving other problems, in a distributed manner, by iterating on an FG representing the factorized truth table. An ambitious attempt would be to factorize (or even approximate the factorization of) the Discrete Log function– see Appendix A.

Chapter 7

Simulations and Results

In this Chapter we will look at the specifics involved in the task of assessing the BER performance of LDPC codes over a range of SNR, E_b/N_0, levels. In its most basic form, the system we wish to simulate is the transmitter-channel-receiver environment most resembling an actual employment of the code. To this aim, we require all the components previously discussed in this thesis to be linked in such a manner that it is possible to repeatedly execute the system (simulate), while monitoring several concurrent outputs.

Coding gain is defined as the reduction in SNR required to achieve a specific error probability for a coded communication system compared to an uncoded system [17]. In other words, how much more noise are we able to handle (at the same BER), by using the coding scheme. In all coding schemes, there is a coding threshold, beyond which there is nothing gained by further reducing SNR. In fact, the code loses its effectiveness at SNR below the threshold, making for a negative coding gain where the code performs worse than the uncoded transmissions.

The following is a brief summary of the simulation components, with reference to the previous chapters of this project.

7.1 Components

The transmitter consists of the encoder discussed in Ch. 5, where the generator matrix, G', is used to calculate a codeword from a random, k-bit information vector, $\mathbf{x} = \mathbf{v}G'$. In most cases, due to the $\mathcal{O}(N^2)$ cost of encoding, and the fact that the codes modelled are linear, we skip encoding by always simulating the all-zero codeword.

7.1.1 Channel

As discussed in Ch. 3, the channels are modelled simply by adding random noise to the transmission. The noisy output will be refered to as $\mathbf{y} = \mathbf{n} \oplus \mathbf{x}$, where $\mathbf{n}$ depends on noise level (SNR).

7.1.2 Receiver

By allowing the SPA decoder to run until completion (i.e., either convergence, or *timeout*), the number of bitwise discrepancies between $\mathbf{x}$ and the quantized (hard decision) decoding, $\hat{\mathbf{x}}$, is returned. This requires a global "monitoring routine," which, unlike the decoder, 'knows' the original, error-free vector. Although this may seem as a step away from the distributed design that is used in this project, it is important to point out that the simulation module is a diagnostics tool, and should not be considered part of the decoder software.

In producing performance data, there are three scenarios in which the decoder may halt. Firstly, there is the situation where $\hat{\mathbf{x}} = \mathbf{x}$. This is either a result of the decoder correcting all errors; or, that the codeword was unaffected by noise. This situation contributes nothing to the BER plot, and should be observed increasingly often as SNR increases. Secondly, the decoder may exhaust all iterations, halting in an invalid state (syndrome $\mathbf{z} \neq 0$), Such *detected errors* are most frequent in the low end of the SNR range. Finally, the decoder may halt with a valid codeword *other than* $\vec{x}$. To the decoder, which does not know $\vec{x}$, such *undetected errors*, or word errors, are impossible to avoid. These errors are indications of poor LDPC codes, and should not be frequently encountered. Nevertheless, to produce "fair" plots, they must still contribute to the BER plot.

7.2 Bit-Error Rate Simulations

By linking these components into a simulations model, various data can be produced describing the performance of the code. Such a system is called a Bit-Error Rate Tester (BERT). The purpose of simulating on a code is to produce a 'profile' of this particular code's error-correcting capabilities–its performance–over a range of channels. Recalling from Ch.3 that the AWGN channel can be thought of as a soft-output BSC, we can think of each SNR value as a seperate channel, with transition probability p.

Since the bit-errors are independent events, we may conduct this as a probabilistic experiment [57]. The most essential data are the BER points, which are calculated as

$$\mathrm{BER}_{E_b/N_0} = \frac{B}{NS}, \tag{7.1}$$

where B is the total number of errors observed over the blocklength, N. To achieve reasonable confidence that the calculations are correct, we need to repeat the experiment until we have sampled enough errors. For instance, 95% confidence, which is quite standard, requires $B = 100$ samples [58]. The complexity of producing the required information is relatively low. The $\mathcal{O}(N^2)$ encoding process can be avoided by always transmitting the all-zero vector which–in a linear code–is always a codeword; $\vec{0}H^T = \vec{0}$. After decoding, the statistics are found by counting the remaining errors, so the entire simulation can be bounded by the $\mathcal{O}(N + m)$ Sum-Product decoder.

Note that the SNR (signal quality, E_b/N_0) is not a directly expressed in (7.1), but affects the total number of bit-errors we sample. Hence, as the SNR increases, we expect a significant drop in the occurence of errors. This is seen

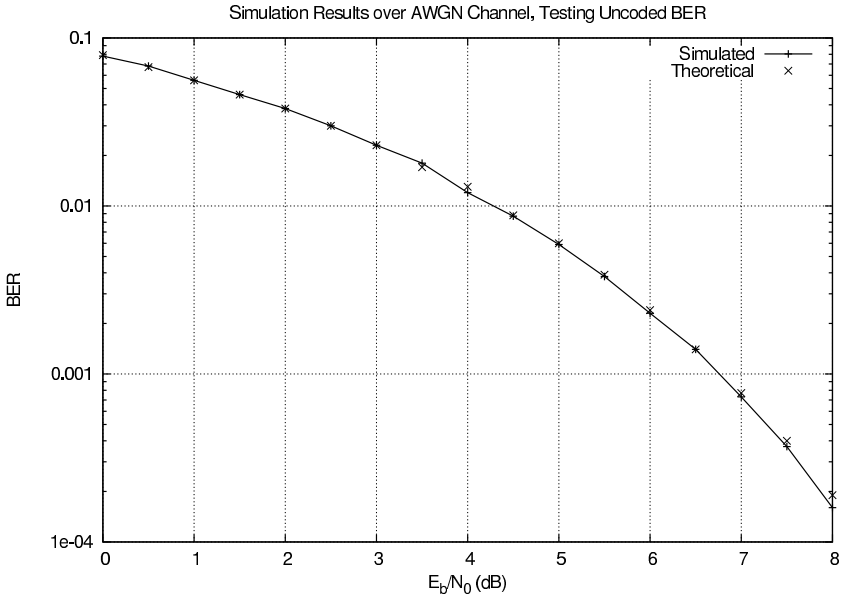


Figure 7.1: Comparison of simulated uncoded BER, and theoretical uncoded BER according to (7.3). To gather sufficient data, we simulated 5×10^3 transmissions over the interval $[0, 4\rangle$; 5×10^4 over $[4, 7\rangle$; and 10^6 over $[7, 10\rangle$.

from typical BER plots, where the range of the y-axis drops by orders of magnitude (which is why we always plot in the logarithmic scale). As errors become less frequent, it becomes increasingly difficult to maintain confidence (100 errors), and S must be allowed to grow quite large. This is a tradeoff between simulation time and confidence, meaning that, beyond a certain upper SNR, we may choose to simply stop, and truncate BER curves; or, we may accept a rougher estimate, and proceed with reduced S. Hence, it is common to supply plots with error-bars, indicating the confidence at each plot.[1]

7.2.1 Uncoded Transmissions

The BER performance of the uncoded (i.e., not decoded) transmissions is determined by modulation and SNR only, and can be approximated mathematically. Using the BPSK modulation, the probability that a bit is in error is the cumulative probability of it being demodulated (6.4) to the wrong bit. For instance, the symbol s_1 (3.1) is demodulated incorrectly if the received $\mathbf{y} < 0$;

$$P(e|s_1) = \int_{-\infty}^{0} p(y|s_1)\, dy = Q\left(\sqrt{\frac{2E_s}{N_0}}\right), \tag{7.2}$$

where N_0 is the noise density (3.3), and $Q(x)$ expresses the area under the tail (probability) of the Gaussian PDF (3.10) [29]. Since the symbols (in this case, s_1 and s_2) are transmitted with equal frequency, they are equally likely at the

[1] Our simulations are truncated at the SNR where confidence dropped below 95% (due to computer resources), hence we have not included error bars.

receiving end, and the average probability of (uncoded) error is

$$\mathrm{BER}_{\mathrm{unc}}(\mathrm{SNR}) = \frac{1}{2}P(e|s_1) + \frac{1}{2}P(e|s_2) = \frac{1}{2}\mathrm{erfc}(\sqrt{\mathrm{SNR}}), \tag{7.3}$$

where SNR is the dimesionless *ratio* of signal to noise,[2] and not given in dB. As illustrated in Fig. 7.1, this is a very valid approximation.[3]

7.2.2 Word-Error Rate

In the coded transmissions, there is another error-event called a word (or frame) error. According to the code, we may verify whether a received and decoded N-bit vector is a valid codeword. If it is not, then we have sampled a (detected) word error (this means that the decoder 'timed out').

Also, there is a second, more worrisome word error event, which is *undetectable* to the receiver. Consider the impact of channel errors is sufficient to offset the input to the decoder to such an extent that the convergence is drawn towards the gravity of a neighbouring codeword (in MLD terms). If we experience more than $\mathrm{d}_{\mathrm{min}}$ bit-errors, the received vector (decoder input) may be more similar to a different codeword than what was actually transmitted. In this case, a significant fraction of the errors will be 'viewed' as correct bits of this other codeword, and result in a successful decoding–to the wrong codeword–that is impossible to detect.[4]

The rate of undetected word errors is, arguably, the most important piece of information produced by the simulation. Especially when working with long, sparse codes (such as LDPC) for which it is difficult to calculate the minimum distance, $\mathrm{d}_{\mathrm{min}}$. The WER points may be plotted on the same scale as the BER plots, and are calculated as

$$\mathrm{WER}_{E_b/N_0} = W/S, \tag{7.4}$$

where W is the total number of word errors sampled. Three different WER plots may be produced, depending on how we count W; undetected WER, detected WER, or, total WER (the sum of the two first). In a good code, we must require $w_u \approx 0$ (undetected WER), so it is common to plot the detected (or total) WER.

7.3 Characteristic Data

In producing the samples for the BER curves (7.1), there is a great deal of valuable information that is simultaneously produced. In this section, we will briefly look at the output of the simulation software of this project.

`Sims` is the number of transmissions simulated, S, each sending an N-bit codeword. In order to achieve sufficient statistical confidence as errors become less frequent, this number will increase with every SNR step.

[2] Conversion from dB to ratio as $\mathrm{SNR} = 10^{(E_b/N_0)/10}$.

[3] Note that the discrepancy between the curves is mainly due to insufficient data at SNR above 8dB, where errors are infrequent.

[4] At least, on the bit-level. If this word is part of a larger, overall message (text or picture), then it is quite possible for a human to detect parts that are out of context.

`T.o.` counts the number of decoder failures; i.e., transmissions where the decoder did not reach a valid state within the maximum number of iterations. Output also includes the percentage of timeouts, out of the total transmissions S. This value is expected to drop quite steeply as noise levels decrease.

`Berr` is the total count of bit-errors over all simulated codewords, B. This column also outputs the percentage of bit-errors introduced by the decoder. In normal SPA decoding, error-free bits may be corrupted due to unfortunate information via their support. When testing novel decoder schemes, this is an important value to monitor.

`Werr` is the count of word (or, frame) errors encountered; i.e., simulations where the decoder reached a valid state, which was *not* the originally transmitted codeword. Such *undetected errors*[5] are very disruptive, and clear symptoms of a *bad* code. Due to the large free distance (and d_{min}) of moderate-to-large ($N > 10^3$ [37]) LDPC codes, this number should remain at 0 even at low SNR.

`OK` counts the number of successfully decoded transmissions. Note that, to produce "fair" statistics relative to the uncoded curve, this count must also include the error-free transmissions–even though these are in no way attribute to neither code nor decoder. The percentage of such "direct throughput" is indicated, and should always comprise only a negligable fraction of the total transmissions yet with a slight increase towards higher SNR.

`Av. Berr` is the average number of bit-errors per N-bit transmission. Obviously, this should be monotonically decreasing.

`Av. It` shows the average number of decoder iterations performed, and should drop quite rapidly.

`Av.DIt` gives similar information, yet it disregards the error-free transmissions, averaging more precisely the number of decoder iterations used. When evaluating novel decoder schemes, such as those described in Ch. 6, this field provides interesting information on the decoder's ability to converge, which is a good indicator on the effectiveness of the decoder. As a reference, note that SPA with flooding scheduling rapidly drops to one iteration, regardless of `max`–see Fig. 7.2.

7.3.1 Error Floors

Exhaustive BER curves of linear codes in general show an unmistakable tendency towards separating into two distinct components. Good codes initially show a swift gain (as opposed to other codes, and the uncoded transmissions) within the first few decibels. As the signal quality increases towards higher SNR, and the number of error events drop, it is expected that the code will perform significantly better. This is called the *waterfall region,* and it is the ability of LDPC codes to push this gain extremely close to the theoretical (Shannon) limit (see [12]) that has earned them the leading position among current coding schemes.

At some point, however, this gain may suddenly break off; flattening out into the error floor. Perhaps the most crucial data on the BER curve is the SNR marking this drop in gain. The corresponding BER is then what is maximally

[5] In real life, the decoder obviously can not verify the validity of a valid state, making these errors the most harmful kind.

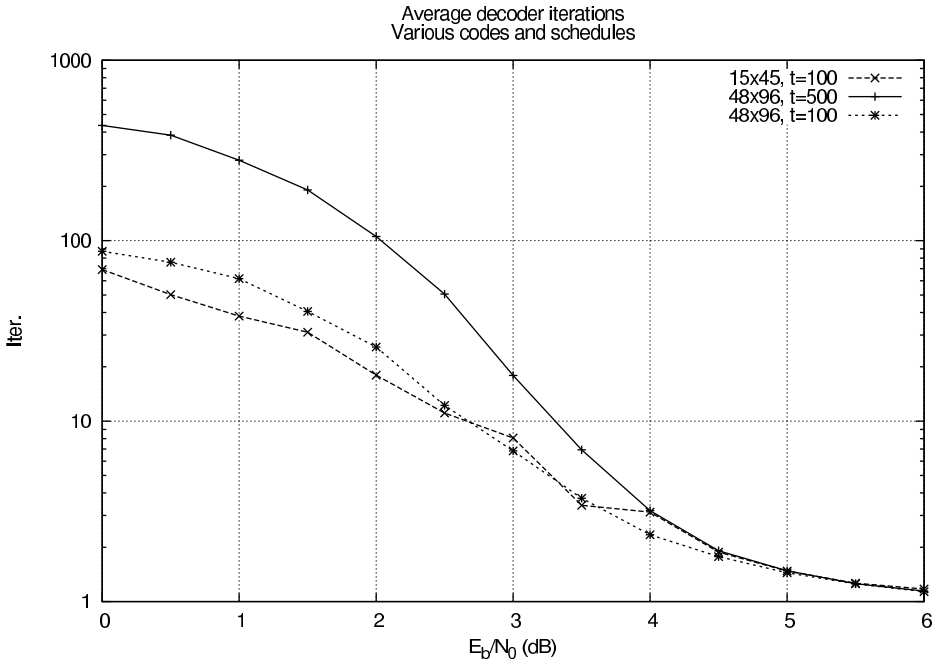


Figure 7.2: The average number of "decoder iterations" is independent of time-out, and only weakly dependent on N [3].

achievable for this code. However, there is inagreement in whether LDPC codes are affected by this phenomenon.

Conventionally, flooring effects are blamed on low-weight codewords, which would imply poor d_{min}. In simulations, this means that there is a non-negligable probability that the decoder will produce undetected (word) errors. These vectors contribute an amount of bit-errors equal to the distance between the codewords; $d_H(\mathbf{v}' \oplus \mathbf{v}) \geq d_{min}$. At high SNR, where $P(e)$ is low, this gives an unproportional bit-error count, which, at high `Werr`, will cause `Berr` to flatten out.

In the case of LDPC codes, it it assumed that the sparsity of H would generally result in high d_{min} (proportional to N [4, 37]), due to the large number of columns required in order to sum up to get zero (modulo 2) [59][6]. As stated on MacKay's website, "well designed LDPC codes do not have an error floor. If you write a bad decoder [...] then an error floor may appear" [3]. Even so, some results indicate that LDPC codes do in fact show some flooring effect [61].

In designing the simulation software, it is important to be aware of "false error floors," which are caused by insufficient confidence (too low B) and *not* poor distance measures of the code or numerical problems in the decoder. If we do not allow sufficient experiments, S, we may generate insufficient data to produce an accurate BER point. If this is not taken into consideration, the plot will flatten out towards a fixed BER (flooring), which is not dependent on

[6]Determining the minimum distance of LDPC codes is recognized as a NP-hard problem [60].

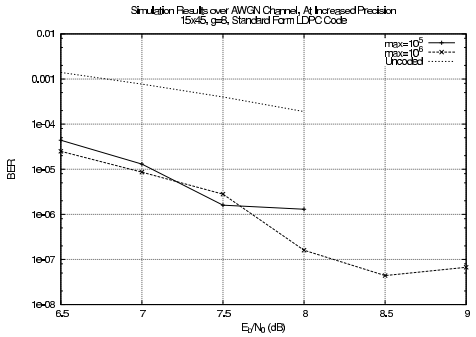


Figure 7.3: Increased precision (no flooring) as `max` is increased.

code/decoder characteristics, but rather on simulation precision;

$$\lim_{P(e)\to 0} \frac{\text{Berr}}{N \times \text{transmissions}} = \frac{1}{N \times \texttt{max}}. \tag{7.5}$$

Fig. 7.3 illustrates the increased precision as the size of the experiment is increased. Note that this is not equal to increasing the decoder timeout.

7.4 Simulations

Using the above described software, we produced performance data of the type displayed in Fig. 7.1 for various constructions. The main simulation parameter is the timeout, T; the maximum number of iterations before declaring a failed decoding. Counterintuitively, perhaps, the (minium) number of iterations *required* to converge, is largely independent of blocklength, N [50]. At high noise levels (low SNR), we typically observe that the decoder has a high timeout percentage, i.e., it exhausts all T iterations with little success. As SNR increases, the amount of error drops, and the decoder converges more and more quickly. This can be seen by noting the logarithmic drop in average iterations used. Even when disregarding error-free "throughput," we should still observe a distinct drop. Beyond an only moderate SNR, average iterations drops below $\log N$.

7.4.1 Flooding Schedule

Our simulations were performed on a standard desktop computer,[7] which restricted us to analysing the performance in the range 0 to approx. 10dB. However, it is a valuable observation that these simulations are easily parallelizable, in that the SNR range can be partitioned among several processors, with no overhead of intercommunications. To validate our software, we imported MacKay's 48×96 code [41], and simulated BER performance in the range of 0 to 6dB, using flooded scheduling with $T = 500$. Our results were plotted against the BER data obtained along with the code, and–as Fig. 7.4 shows–the curves agree

[7]Intel Pentium-4, 2.26Ghz CPU, with 503.1MB Ram.

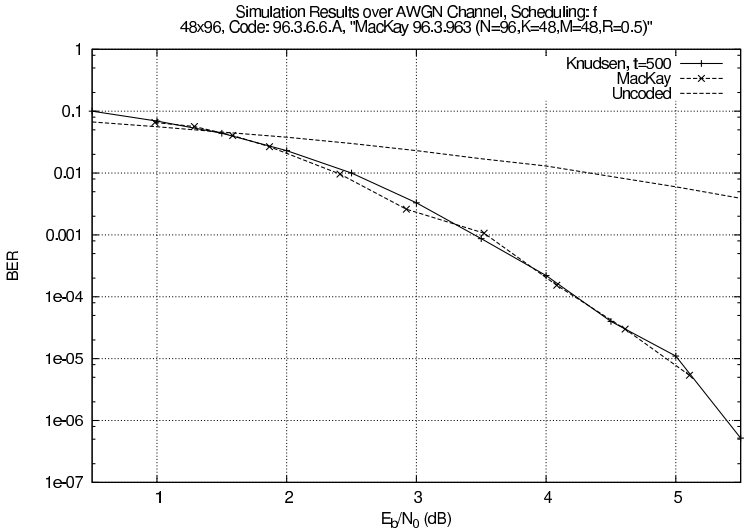


Figure 7.4: Our simulation software validated against the results of MacKay.

quite well. With the confidence that our system works,[8] we may evaluate the performance of codes generated in Ch. 4 using the flooded SPA decoder. Furthermore, similarily to the 'uncoded curve,' the flooding results will be used as an additional benchmark against which to test our alternative decoder schemes (see Ch. 6).

[8]We did not simulate the data for the 'MacKay' plot. Hence, the figure is a unbiased validation of our simulation system; the encoder, channel model, as well as the decoder.

Chapter 8

Experimental Decoding

With the proven MLD capacity, as well as the extreme simplicity, of the SPA as an LDPC decoder, it may seem as if there is no pressing need for any further developments in the area. As put by Tanner, "nowadays if you can't get close to Shannon capacity, what's wrong with you? [62]" in a comment on LDPC and Turbo codes.

Still, the most record-breaking results are always at the cost of impractically large blocklengths, where the weaknesses in the codes (e.g., cycles) become negligible. With the explosion in handheld, online devices, however, memory and power conservative, high-throughput decoders are in demand. Considering the distributed design of the previous chapter, this appears to be an important application of LDPC codes and SPA decoders, where the new question is–how can we improve the performance at the low-end?

Granted that the updating rules (Ch. 6) are seemingly optimal, we will have a look at some modifications to the scheduling–the order–of such update calls, and how this might counter some of the obstacles in decoding small to medium sized LDPC codes.

8.1 Feedback, Short Cycles

Randomized LDPC codes across the entire range of sizes, all share the fact that they are ridden with cycles of varying length. While the length of the shortest cycle–the *girth* of the code, $G(H)$–can be maintained proportional to N, observations conclude that the girth 'profile' is quite similar, regardless of blocklength. As $\mathrm{d}_{\min}$ grows with N, we expect the relatively poor performance of small LDPC codes to be due to an increase in word (undetected) errors. Most likely, these errors are caused by an overwhelming amount of self-sustaining weaknesses in the soft information; in other words, feedback caused by short (length < 10) cycles.

The girth of a node, g_v, is defined as the length of the shortest cycle in $\mathcal{G}$ passing through v [43]. As discussed in Sect. 4.6.3, this information can be collected during the normal construction of the EBF algorithm.[1] When the nodes

[1] Alternatively, the same information is accessible via a Depth-First Traversal of the graph representing the code. This is convenient for dissecting codes not constructed by EBF.

are updated in a strictly alternating backward/forward manner (i.e., flooding scheduling), g_v denotes the exact number of iterations before the 'independence' of v is violated by feedback. In other words, after exactly $g_v/2$ iterations, the message sent by node v is returned. While it certainly has been diluted by the influence (product) of other messages underway, in short cycles it should still be considered significant.

By modifying the SPA schedule, it is possible to control the flow of information, for instance, such that cycles are traversed less often, reducing the feedback. Also, some 'dynamic' schemes that change the graph during decoding may introduce feedback on nodes that do not appear to be part of any explicit loop of edges. As a result, the fixed, local values g_v no longer apply.

8.2 Detecting Cycles

Before discussing various experimental implementations of the Sum-Product Algorithm (SPA), we suggest a diagnostical tool for monitoring the amount of feedback in the decoder. Define the *effective girth* of a node v, $\tilde{g}_v$ as 2 times the average number of iterations before it is affected by feedback. An interesting parameter of the decoding scheme is the effective girth of $\mathcal{G}$, defined simply as the minium of all $\tilde{g}_v$.

8.2.1 Using the SPA

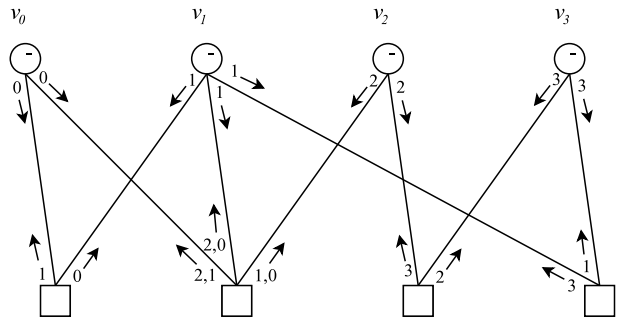
The extrinsic principle of SPA is meant to guarantee the integrity of input messages to node v, in the sense that they are not affected by the local value at v. Restricting ourselves to SPA decoding, define a communications *setting* as the desired combination of code and SPA scheduling (e.g., flooding; or more sophisticated schedulings). By reconfiguring elements of the Factor Graph corresponding to the code, we may run a 'SPA-like' MPF algorithm which will detect the presence of feedback in the original setting. Constr. 5 sketches the outline of the scheme. The following update rule is adapted directly from SPA (6.19), and should seem quite familiar. Note that, for brevity, we denote ID/age tuples simply by ID, where 'v' is the ID of node v,

$$\vec{\mu}_{v\to f} = \left(\bigcup_{f'\in\mathcal{N}(v)\setminus\{f\}} \vec{\mu}_{f'\to v} \setminus \{v\} \right) [\cup\{v\}], \tag{8.1}$$

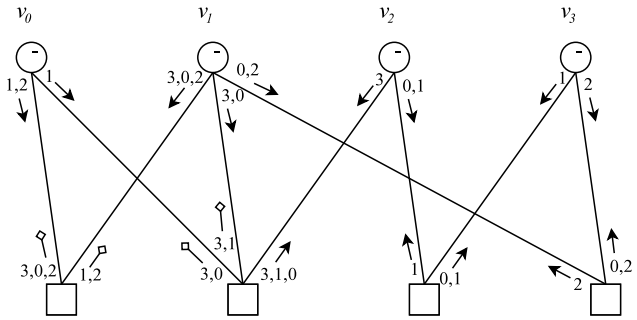
where the final argument (in brackets) indicates that whether or not local ID, v, is appended to outgoing messages, is dependant on the SPA schedule being tested. The summary part of (6.19) would translate to removing any duplicate ID's from outbound messages. Note that this is done implicitly by the *union* operation (e.g., $\{a,b,b\}\cup\{a,c,d\} = \{a,b,c,d\}$), and is not expressed in (8.1).

The converse rule is, again, a simplification of the generic rule (8.1);

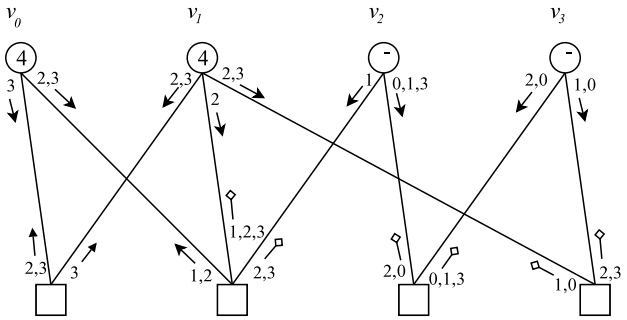
$$\vec{\mu}_{f\to v} = \bigcup_{v'\in\mathcal{N}(f)\setminus\{v\}} \vec{\mu}_{v'\to f}. \tag{8.2}$$



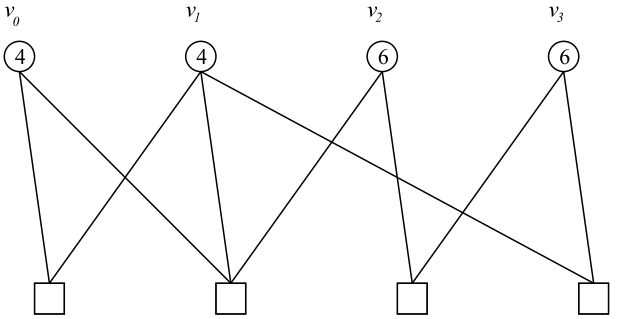
(a) Iteration 1.



(b) Iteration 2.



(c) Iteration 3: 4-cycles detected.



(d) Iteration 4: 6-cycles detected; all bits contaminated. Done.

Figure 8.1: Girth Monitor on a small LDPC code. Within 4 flooding iterations, all bits have determined their effective girth (which, in this case, equals local girth). Age fields of messages are not shown.

Construction 5 Construction for monitoring effective girth using SPA.

Assign a unique ID to each bit node. Messages are defined as vectors of tuples, (`nodeID`, `age`), where age counts the number of *edges* the message traverses. Nodes (both bit and function) produce the *extrinsic union* (8.1) of input messages, and increment age counters. When there is overlap (i.e., duplicates of an ID), only the ID with the lowest age is kept. Depending on the SPA schedule that is tested, bits append their ID to outgoing messages (with age 1). Bit nodes also perform a membership test with their local ID on the set of input vectors, to determine feedback. Stopping criterion is when all bits have determined their local girth.

8.2.2 Flooding Scheduling

The standard extrinsic principle of SPA is obeyed, so the bit nodes' membership test detect *violations* of this key principle, caused by cycles in $\mathcal{G}$. As an example, consider the flooding schedule on the small LDPC code of Fig. 8.1. The (known) girth of this graph is 4, yet it contains larger cycles (of length 6). Messages containing feedback are indicated with square arrows. In the flooded case, one iteration consist of updating all bits, followed by all functions[2], and bits append their ID's *only* to initialize messages in the first iteration. When bit v receives its first 'contaminated' message, the corresponding age-field will necessarily contain the length (in edges) of the minimum cycle connected to v, such that $g_v = \text{age}/2$. Hence, this bit is 'done', and will filter out the presence of its own ID from any future outbound messages. This way, it may still aid other bits in determining their local girth. The stopping criterion for flooding SPA is whether all bits have determined their local girth.

8.2.3 Implicit Feedback

An important observation is that also 'dongles,' i.e. bits of degree 1 (not counting input edges) will–perhaps counterintuitively–experience feedback, despite the fact that they are not part of any cycle. As long as the graph is not acyclic, their independence will be compromised by the–perhaps distant–cycles in the graph. This supports the 'Local Girth Detection' of Sect. 4.6.3, which upper bounds local girth by $g_{\max}$ Consider, for instance, dongles connected to either end of the graph of Fig. 8.1. By termination, the ID of the 'left dongle' would have traversed the adjacent 4-cycle giving local girth $4 + 2 = 6$, while, on a similar argument, the girth of the 'right dongle' would be 6.

8.2.4 Practical Comments

Firstly, there is the concern of memory usage. The vectors of tuples (i.e., messages) will increase with $\rho - 1 + \gamma - 1 = \rho + \gamma - 2$ entries per iteration, where ρ and γ are the degrees of, respectively, the function and bit nodes it passes through. However, in the extreme case where all messages are stamped with local ID, (8.1) shows that the union of such messages can never exceed the number of unique ID's in the graph, N; regardless of schedule tested.

[2]Or, conversely, bits *then* functions.

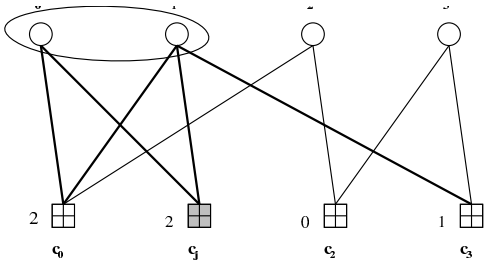


Figure 8.2: An example showing the iteration updating c_j.

As mentioned, this is a diagnostics tool, which should be used to assess average performance of a communications setting (code/scheduling pair), and not be part of an actual decoder implementation.

In the straight-forward case of flooded SPA, information flow is strictly deterministic (update all bits, followed by all functions), so the effective girth of any bit will neccessarily equal the local girth of that bit. Hence, this information should equal the results of preprocessing techniques of Sect. 4.6.3, or, simpler still, a regular DFT (depth-first traversal) of the graph.

In the following, we will describe some experimental SPA schedules in which the flow of information (on edges) is more complex–and, in some cases, even non-deterministic. Being able to monitor girth, is a valid tool for assessing the performance and stability of novel decoding schemes.

8.3 Avoiding Cycles

Perhaps the most intuitive way of increasing performance, is by simply avoiding the cycles in the graph. By altering the way nodes are updated, control is gained over the propagation of messages through the graph. From the perspective of particular bits, independence fails at different times, and this information can be used to dynamically (either deterministically, or probabilistically) decide which bits to update in the next iteration. Xiao *et. al* [43] showed interesting results, particularly in the high SNR areas where error floors were lowered, albeit at some increase in complexity. Similarily, if we stop updating[3] bit v after $g_v/2$ iterations, we might counter some of the effect of feedback in the decoder.

8.3.1 Delaying 4-Cycles

Since a 4-cycle is only causing feedback if the nodes comprising it are updated, we suggest a simple scheme for avoiding certain nodes that, if updated, would cause feedback at this particular iteration.

Consider a scheme in which iteration j consists of updating one check node, c_j, followed by the updating of all this node's adjacent bits $v_i \in n(c_j)$. As these

[3]Note that 'stopped bits' still contribute to the overall convergence, by forwarding their current, "locally converged" states.

bits are updated, they forward their information towards the check nodes in their support, $n(v)$. The iteration ends with the selection of the check, c_{j+1}, for the next iteration.

Say c_{j+1} is in the support of two or more bits in $n(c_j)$, then updating c_{j+1} would mean feeding this information back towards the bits in $n(c_j)$, via the 4-cycle.

Consider placing a counter within each check node in $\mathcal{G}$, which is reset to 0 at the beginning of each iteration. As each bit $v_i \in n(c_j)$ produces an output message onto the edge towards some check c', it simultaneously increments the counter in this check. As the next check, c_{j+1}, is selected, we select randomly among those with counter value *less than* 2, thereby effectively avoiding 'closing cycles' of length 4.

Obviously, in some subsequent iteration, $j' > j + 2$, we may select one of these checks (whose counters are now 1 or 0 due to some perhaps distant bit). As we begin this iteration by updating $c_{j'}$, we will cause the independence of this bit to be violated. However, the idea is that, by now, other updates may have 'diluted' the effect of this information.

8.4 Dynamic Decoding

As discussed in Ch. 4, a code based on a Parity Check matrix, H, uses only one in a potentionally large ensemble of equivalent matrices for the same code. Following the lead of [63], decoding performance is improved by increasing the row-space of the Parity Check matrix, H; thereby extending it to $H^\star$. With N^2 equations (rows), MLD was achieved for a Hamming code over the erasure channel, at the cost of increased complexity.

Our suggestion is that a similar gain may be achieved at a lower cost by dynamically changing the rowspace of H during decoding, in such a way that maintains equivalence with the original code at all times. Most importantly, this means that the dimension, $m \times N$, and the *null space* remains the same–otherwise, we would not be working with the original code, which makes no sense from a decoding point of view. Interestingly, the operation of Pivot (see Ch. 2) does just that; it takes us from one matrix to another–always within the same ensemble. Just as in Gaussian Elimination (where pivot is used for the simultaneous solving of a multi-variable equation set, defined by a matrix), the solution (null) space remains invariant. At each iteration, $H^{(i)}$ consists of some (perhaps arbitrary) selection of m check-equations (rows) of $H^\star$, in such a way that we may ultimately pass through a large fraction of the rows in $H^\star$. These new rows are linear combinations of the original m rows of $H^{(0)}$.

The (reversable) transformation from one matrix to another, is called a *rotation* of H' to H''. By repeating the process, we enumerate a set of rotations, which, since pivot is reversable, eventually closes back on itself. This finite, yet potentionally extremely large, set of rotations is called the *pivot-orbit* of the original matrix, H_0, and is a subset of the entire ensemble of matrices from which H_0 is selected.

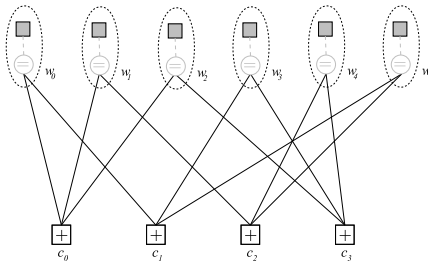


Figure 8.3: The simplified graph, $\mathcal{G}^\star$, suitable for pivot.

8.4.1 Rotating H using Pivot

A pivot is defined as "the first non-zero value of each row of a matrix after the matrix has been converted to row echelon form [64]." The operation of pivoting on a matrix element (i, j) (an edge in the FG), means transforming the matrix such that column j becomes the *identity vector*, $\vec{e}_j$, which is non-zero only in position j. For our decoding purposes, a weight-1 column of H is called a *systematic bit*, of which the standard form of H contains m. Although these bits are very poorly protected (minimal support, $|n(v)| = 1$), they will not disrupt the equivalence of the code (matrix).[4]

The conventional, 'Forney-style' Factor Graphs used in SPA decoding contain auxillary objects–the 'input nodes'–connected to each bit. These nodes (and their edges) are not part of the FG, as defined by using the Parity-Check matrix, H, as an adjacency matrix. As illustrated in Fig. 6.1, these ensure the continuous input of the data that is to be decoded–namely, the channel symbols–and must not be moved or reordered in any way during decoding. This presents a challenge when we wish to rotate H (and, implicitly, $\mathcal{G}$) via pivot. Define $\mathcal{G}^\star$ as the simplified graph resulting from grouping each bit node and its adjacent input node to one composite node, $w_i = \{v_i\} \cup \{s_i\}$. Fig. 8.3 shows an example. We may now pivot on any edge of $\mathcal{G}^\star$, and construct the corresponding FG by decomposing nodes w_i.

The graph-based pivot operation of Fig. 2.1 serves our purpose in an efficient manner. Also, it is is based solely on decisions local to the two nodes involved, which agrees well with our overall distributed approach.

To gain confidence that pivot would, in fact, maintain equivalence, we ran a test checking that the rowspace of $H^{(i)}$ did not change. The original m rows of $H^{(0)}$ constitute a *basis* of the corresponding 2^m rowspace, $\mathcal{C}^\perp$.[5] By expanding and storing this in memory, we verified that any rotated basis (due to pivot) would still expand to the *same* rowspace. Within reasonable dimension m, we checked that all m rows of any rotation $H^{(i)}$ were listed in $\mathcal{C}^\perp$; otherwise, it would not be a basis for the same space. All observations were positive.

[4]Recall that pivot is used to reduce H to standard form, from which the generator matrix *of the same code* is found.

[5]The rowspace of H is the set of codewords for the *dual code*, $\mathcal{C}^\perp$, for which H is a generator matrix.

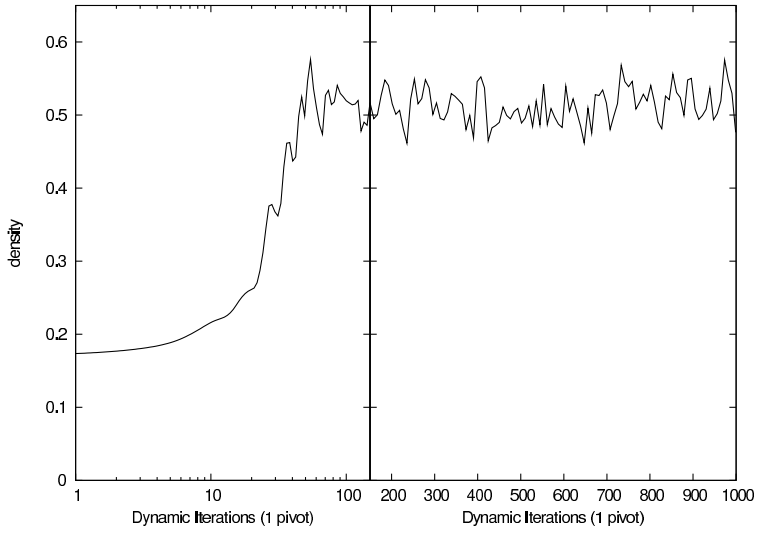


Figure 8.4: Density of 48×96 LDPC Code (MacKay) over 1000 random pivots. The code is $(3, 6)$-regular, which gives $\Delta_0 = 3/48 = 1/16$.

8.4.2 Breaking Oscillation

In addition to expanding the support of the code, we expect pivoting to be able to "shake loose" the decoder from points where its stuck in oscillating states [53].

8.4.3 Consequences of Pivoting

Although we are now confident that pivoting does allow us to update the basis (rows) used for decoding (without redefining the code), there are certain unfortunate side-effects to the procedure. As a general concern, LDPC code design features such as girth; density; and, weight-distribution (or regularity), are disrupted by pivoting. As discussed in Ch. 4, good LDPC codes are usually carefully designed so as to optimize these (and other) features.

First there is the concern of girth. Consider a standard LDPC code, with "no 4-cycles;" say, girth is $g = 6$. In terms of local neighbourhoods, it is easy to see that this means that the overlap of edges (between $n(u)$ and $n(v)$–see Ch. 2) is $\mathcal{O}^E_{u,v} = \emptyset$, since these are precisely the edges that *would* otherwise close 4-cycles. Hence, pivoting (on *any* edge of $\mathcal{G}^\star$) means creating (2.6) $\mathcal{E}^\star = (|n(u)|)(|n(v)| - 1)$ edges which *all close cycles of length* 4 (in $\mathcal{G}^\star$, and in $\mathcal{G}$). Conversely, if $\mathcal{G}$ does contain 4-cycles, pivoting will *remove* $\mathcal{E}^\dagger = \mathcal{O}^E_{u,v}$ of these cycles, but, in a sparse graph, the number of edges created (2.6) is likely to be larger than the number of edges removed (2.7). Unless the graph is highly dense, the net result of pivoting will be an overall increase in both 4-cycles, and density.

This leads us to the greatest concern; LDPC codes are defined as *low-density* codes, and–obviously–this is corrupted (very quickly, in fact) by pivoting. Recall that, in a (γ, ρ)-regular, bipartite graph (i.e., a typical LDPC construction), the original number of edges is (4.3) $|\mathcal{E}_0| = N\gamma = m\rho$. Also, initially, $|n(u)| = \rho$

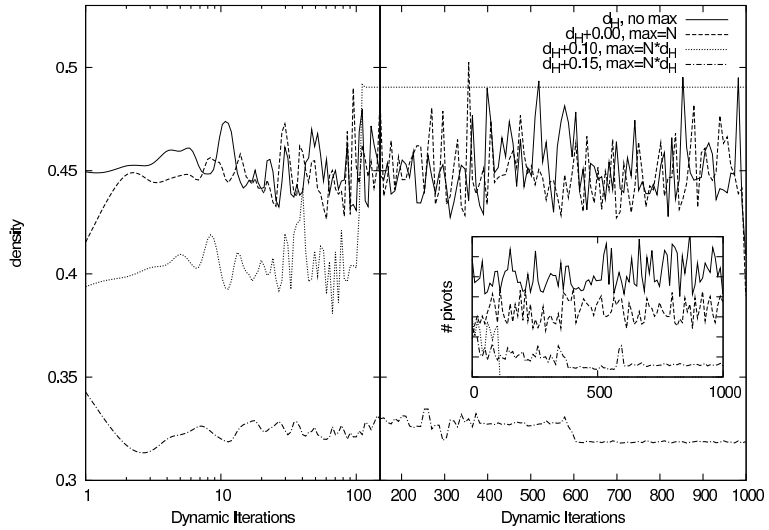


Figure 8.5: By restricting the application of pivot, we are able to control the increase in density, while still rotating the rowspace of H. The code is the same as in Fig. 8.4.

and $|n(v)| = \gamma$, such that the 'original' density' is $\Delta_0 = \gamma/m = \rho/N \leq 1$. Using this information, we see that the first pivot will increase the number of edges by $\mathcal{E}^\star = \rho(\gamma - 1)$, which is an increase (in percent) of

$$\begin{gathered} \mathcal{E}_0(1 + x/100) = \mathcal{E}_0 + \mathcal{E}^\star \\ \Downarrow \\ x = (\gamma - 1)/\rho \cdot 100 \end{gathered} \tag{8.3}$$

As seen from Fig. 8.4, the density of the code has a significant jump within the first 100 pivots, after which the number of created and deleted edges neutralize each other; $\mathcal{E}^\star \approx \mathcal{E}^\dagger$, for an average density of ~ 0.5.

8.4.4 Maintaining Sparsity

As we have seen, random pivoting has a tendency to create a sharp increase in density, beyond which the density stabilises around 0.5. However, it is possible to control pivot such that we ensure that we maintain stable density.

The net effect of one pivot operation is inversely proportional to the connectivity of the local neighbourhood (2.7). If the current density, Δ, is low, we want to allow a few pivot operations, such that we may still 'update' the rowspace of H. As Δ grows, we increase the usage of pivot–on certain edges–to simultaneously *reduce* the overall density, while updating rowspace.

Consider some candidate edge, (u, v). In a distributed design, a node can only see its immediate neighbours. A way for v to determine the consequence of pivoting (on that edge), is given in Def. 6 The size of the overlap is $\mathcal{E}^\dagger$ (**??**), the number of edges that will be removed, which reveals the effect this pivot would have on density. The density of the local neighbourhood is then

$$d_E = \mathcal{E}^\dagger / (|n(u)|(|n(v)| - 1)). \tag{8.4}$$

Definition 6 Simple protocol for determining the overlap of two local neighbourhoods.

1. Node v tells u to raise the flags of its local neighbourhood, $n(u)$.
2. Node v asks each node in $v' \in n(v) \setminus \{u\}$ for the total number of flags each node v' can 'see.'
3. The overlap equals the total of these sums returned to v.

We then define the pivot-threshold, $\varepsilon = \Delta$, as the minimum density, d_E, required for pivot. Note that this threshold is dependant on the current density of H, which changes when we pivot; as discussed above.

This information can be used to determine where to pivot (and where not to), such that we may 'rotate' the rows of H without having the density grow out of control. Fig. 8.5 shows the effects of various weightings of this scheme, as compared to random pivoting (Fig. 8.4). Here, one iteration consists of traversing all N bit nodes, in permuted order.[6] Each node is allowed *one* pivot operation to apply to one of its edges; if it finds a valid edge.

Initially, the sparse MacKay code is submit to one random pivot, which creates densely connected neighbourhoods. Otherwise, our procedure would have no edges for which pivot would *reduce* density. This can be thought of as an initial 'shake' to the matrix (or graph), such that we have some disorder to try and fix. By pivoting on edges for which $d_E > \varepsilon = \Delta$, we observe that the density still fluctuates around 0.5, but at an increased amplitude (higher peaks and dips). This is a consequence of one pivot operation trying to counteract the results of the previous, causing the density to flip-flop.

This effect can be calmed significantly by further restricting the application of pivot. Using the intuition of using fewer pivots when the density is low, and, similarily, more pivots when density is high, we stabilise the density at a reduced level. Within one iteration, we limit the number of pivots to $N\Delta$. Also, we try incrementing the pivot threshold slightly,

$$\varepsilon = \Delta + \alpha, \qquad 0 < \alpha < 1 \tag{8.5}$$

we see a lower density, however, at the cost of fewer pivot operations. The lower line in Fig. 8.5 performs approximately one pivot per iteration. Also, the figure shows the change in density as a function of the iteration number.[7]

8.4.5 Protecting Soft Information

A cornerstone of the Sum-Product algorithm lies in storing extrinsic message-distributions on edges. This way, nodes prepare specific output messages for each individual neighbour, so as to minimize feedback. When an edge is deleted–during pivot–we risk losing soft information along with it. Also, as pivot creates

[6] Since pivot is a reversable transformation, this would otherwise cause the scheme to enter a cyclic pattern of pivoting on a small set of edges repeatedly.

[7] Not to be confused with Density Evolution, described in Ch. 4, which tracks the density of *bit-error probability,* as a function of iteration number [11].

new edges, there is no information prepared for this new edge, resulting in its initialization with the neutral message. This means that each pivot operation causes loss of information in the decoder.

Our experience suggests that the convergence of the decoder is sensitive to such information loss. There are two different countermeasures to overcome this. The first idea is the more complex one, adding somewhat to the complexity of the decoding schedule. Before removing an edge (u, v), we check whether it contains a 'fresh' message pending for either u or v. By this, we mean a message that the intended node has not yet received and processed. This can be implemented by adding a simple flag to each message, which is set to 'false' by the receiving node as it reads the message. Hence, we may check for 'true' flags, and ensure that such messages are not lost by updating the corresponding node, which thereby must process that message. Consider we find that we must update v to *evacuate* pending message (from u), $\mu_{v\to u}$. As we update v, we produce a return message for the edge about to be disconnected. Note, however, that we do not need to protect this message, as it is identical to the one already pending for u; $\mu_{u\to v}$. The same is done for u, with the same reasoning. In sum, we can now safely disconnect (and discard) the edge without losing any information.

Similarily, as we create a new edge, (u', v'), we may *populate* this edge by simply updating the nodes u' and v'. At insertion, the new edge contains the neutral message–in both directions. By updating u', we produce an extrinsic output message for each adjacent edge. Hence, the new edge will be populated with the information corresponding to the contents of the messages pending for u'–*except* the edge's own value.[8]

On the other hand, we have the more unconvetional approach of storing information in vertices, instead of edges. To validate this idea, consider the situation immediately prior to decoding. The received vector, $\mathbf{y}$, is then attached to the graph, at the input-nodes. Within the first iteration, this information is 'pulled' onto the edges of the graph, from which it propagates–via edges–from iteration to iteration. Then, after the final iteration, the decoder output is produced by polling the *vertices*–not the edges–to produce the final decoder state. This concept can readily be extended to apply to all iterations, by simply considering each iteration as an independent (partial) decoding.

In other words, at some loss of extrinsic information, we may end each decoder iteration (regardless of schedule) by producing the local value of each bit node, $\omega_u^{(i)}$ (6.17). This is the (unquantized) state of bit node u after decoder iteration i. Now, all information in the graph is stored safely in the vertices, and we may freely disconnect and create by pivoting. The following decoder iteration is initialised by each bit node returning its state onto its adjacent edges.

The validity of this approach is self-evident, by realising that this is precisely the situation at the beginning and end of a normal SPA decoding. However, the drawback with this approach is the loss of extrinsic information in (6.17), which is seen as a node initialises *all* its edges with the identical value (no distribution).

[8]Since edges are initialised with the neutral message, the contribution of the new edge's message will not count (it is neutral), so the extrinsic principle is actually redundant (yet, harmless) here.

8.5 Alternative Scheduling

As discussed earlier in this thesis, the update calls of the iterated Sum-Product Algorithm (SPA) decoder are independent operations. This allows experimenting with alternative schedules, in attempt to improve bit-error-rate (BER) performance.

Proposing that the flooding schedule may be redundant in updating all nodes in every iteration, we try updating only an arbitrary subset of nodes per iteration. Also, we attempt varying the amount (and type) of updates that are done in one iteration. Perhaps some form of alternating scheme has positive effects on the decoder's ability to converge. As a further dimension, we suggest adding randomness to the mix, such that the decoder becomes non-deterministic in space and time (i.e., *which nodes* are updated at any given iteration).

8.5.1 Thresholding

In doing this, we desire a more wide-spread propagation of messages through the graph, thereby effectively reducing the overall amount of feedback. Even with the complete loss of control on the girth, as $\mathcal{G}$ changes, it takes a minimum of 2 successive iterations to complete a cycle. Using the protocol of Def. 6 to maintain sparsity, we simultaneously minimise the inevitable increase in the number of 4-cycles.

Consider the operation of pivot on the edge (u, v), as described in Ch. 2. The edges involved–either disconnected or created–are *all* edges in cycles of length 4. By definition, these edges are those interconnecting the local neighbourhoods of u and v. We may step from $n(u)$ to u (necessarily); from u to v (via the pivot edge); and, from v to $n(v)$ (necessarily). Hence, each edge from $n(u)$ to $n(v) \setminus \{u\}$ closes a 4-cycle in $\mathcal{G}$.

Conversely, the egdes disconnected in pivot, each correspond to the removal of a 4-cycle. Hence, in determining the optimum location (edge) for pivoting, we should consider the size of the overlap, $\mathcal{O}^E_{u,v}$, between the two local neighbourhoods (2.6). If the 'local density' between u and v is greater than 0.5, then we know we will remove more 4-cycles by pivoting on this edge, than the number we create.

With some probability, the subsequent pivot operation will delete one or several of the edges comprising the cycle, such that the intrinsic information is never fed back to the originating bit. The proposed decoding schedule is described in Def. 7. Note that the pivot of step 3 may remove some of the edge(s) that were updated in step 2, thereby wasting computations. Keeping things simple, we acknowledge the room for optimization, yet do not adjust this in this project. After pivot, the change in edges connected to c changes the check equation into a different equation, in accordance with the ideas of [63]. Since we have propagated the information on these edges further into the graph in step 2, we do not lose any information along with the deleted edges.

Several variations on the scheme are explored, where the first two differ in the way the next check c' is selected, in step 4. Firstly, the "Chained" schedule attempts to propagate the updated information as a connected wave through the graph, by always selecting c' among the checks adjacent to one of the bits updated in step 2. However, it is important to take into account the

Definition 7 The Dynamic Decoding Schedule.

1. Perform k *flooding* iterations, where each iteration consists of updating all check nodes of degree $< T_f$, followed by all bit nodes of degree $< T_v$.
2. Protect information by producing the state, ω_u, of each bit node, u.
3. Perform l *pivot* iterations, where we pivot only if $\Delta > \varepsilon$ (8.5).
4. Initialise next iteration by moving ω_u back onto edges adjacent to bit u.

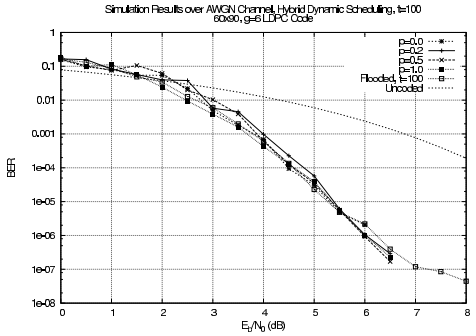


Figure 8.6: Hybrid Scheduling, which consists of regular Flooding iterations, interspersed with one pivot operation (with probability p).

edges removed in step 3–and this approach can get stuck in situations where no suitable c' exists after pivot. The second suggestion is the "Random" schedule, in which c' is selected randomly among $\mathcal{V}_C \setminus c$.

To minimize the variable updates per iteration, it is valuable to try to avoid pivoting on edges resulting in graphs which are not low density. However, to avoid preprocessing and comparing the ρ pivot-options in every iteration, we observe the more convenient approach of simply escaping from such undesirable graphs, by pivoting again. Hence, in step 4, simply choose one of the low-density checks as c', and the congested check(s) should be expected to be alleviated in the subsequent pivot operations.

8.6 Hybrid Decoding

Finally, we have experimented with a 'hybrid' scheme, in which regular flooding iterations are intermixed with one (or several) pivot operations. It is shown [53] that certain error patterns sends the SPA decoder into a oscillation between two states, where neither is a valid codeword. With flooding scheduling, the decoder can not escape this loop and will simply time out. Our hypothesis is that it would be possible to dislodge the system when it's stuck, by using pivot. This can be viewed as a means of 'shaking' the system to furhter disperse the information. The success of such a scheme is expected to reveal itself as a noticable reduction in the number of timed out decodings.

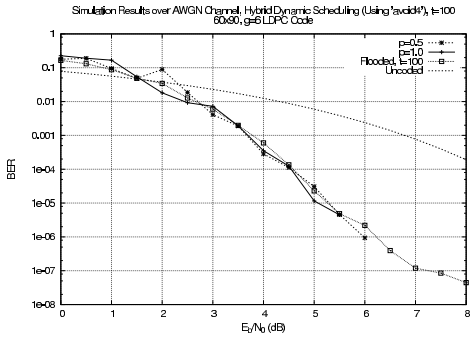


Figure 8.7: Hybrid Scheduling, but with 'avoid4' scheduling instead of Flooding.

8.7 Comments

All schemes have been simulated for BER performance, and compared against conventional flooding scheduling on the same code. Returning to the discussion on distributed implementation, it is essential that the transformations discussed–LC and Pivot–are local operations. Nodes may execute this using only the information readily available in their local neighbourhood, without any need for global assistance or any form of message passing.

The work described in Ch. 6 has been subject to thorough testing in order to verify the effectiveness of the scheme. As a benchmark, we have used the flooding schedule over the same code, with a fair timeout. One decoder (SPA) iteration consists of c check-node updates, followed by b bit-node updates. Hence, the information in the system is updated by an amount proportional to $W = c + b$, per iteration. As such, one flooding iteration can be described as $W_f = m + N$, whereas one dynamic iteration is only $W_d = 1 + \rho$. To compare fairly, we should ensure that both schemes have equal influence; $TW_f = DW_d$. As an example, given $T = 100$ flooding iterations, we need $D = 100W_f/W_d \approx 261$ dynamic iterations.

It is important to point out that this is a weighting of information propagation *only*, and that we are not concerned with discrepancies in workload, or decoder latency. For instance, we ignore the complexity of repeated $\mathcal{O}(\gamma\rho)$ pivot operations since these are mainly design concerns. Also, pivoting on $\mathcal{G}$ will change the set of edges, $\mathcal{E}$. Hence, $\mathcal{E}$ must be *reset* in between decoder applications. Confident that all graphs in the pivot orbit are equivalent, we could simply initiate the next decoding with $\mathcal{E}$ in the final state of the previous. This would be a quite elegant simplification, especially considering an implementation in hardware. However, since the density of $\mathcal{E}$ varies considerably, we produce more fair statistics by always initiating the decoder according to $H^{(0)}$. Again, in keeping with our distributed view, we do not require the separate storing of H (or any rotation of it), but rather store the original adjacency list within each bit node–as a reference.

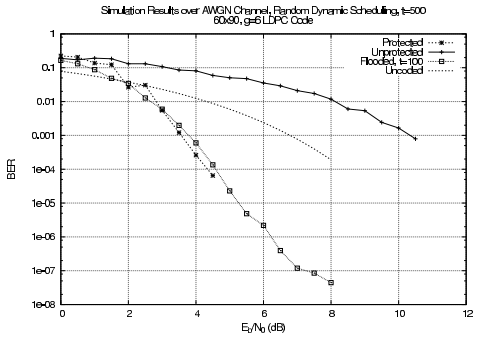


Figure 8.8: Dynamic decoding, using pivot.

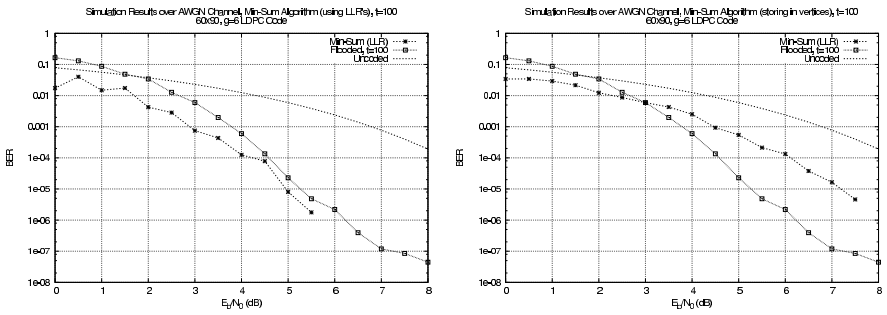
8.7.1 Unfinished Results

Owing to limited time for this thesis, we did not have time to fully explore the ideas suggested in this chapter. However, some preliminary plots are presented here.

Fig. 8.8 show a simulation on the dynamic scheduling, where thresholds are set such that regular flooded iterations are interspersed with *one* pivot-operation on a random edge. Unfortunately, time did not permit experimenting further with the thresholds. The 'unprotected' curve shows the performance when information is *evacuated* from egdes (not stored in vertices) before pivoting. The 'unprotected' curve shows the effect of not protecting information at all. Apparently, although we can not completely rule out software errors, the idea of protecting information is valid and important. Also, to gain confidence in the pivot operation, we did confirm that the *codespace* does not change–that we stay within the orbit, as discussed previously.

This curve also shows that the idea of dynamic decoding may have potential. Although this plot does not show a significant gain (assuming, for now, that the data is in fact valid), it does not show a *worsening* of the convergence. This is an interesting result in itself (if it is correct), because it adds to the conviction that it is possible to decode using SPA on a network that dynamically changes. In other words, if pivots happen beyond out control (not as a decoder scheme), we can still converge.

Also, we experienced an unexplained decoder gain at low SNR when experimenting with decoder schedules involving storing messages in vertices in between regular flooding iterations (no pivoting)–Fig. 8.9(b). A similar gain occurred while running regular flooding in the Log-Likelihood domain (LLR's)–Fig. 8.9(a). We again emphasize our lack of confidence in the results, due to limited time.



(a) Min-Sum (LLR) (b) Storing in vertices

Figure 8.9: An unexplained gain at low SNR.

Chapter 9

Concluding Remarks

In this thesis we have looked at the practical and theoretical issues involved in working with Low-Density Parity Check codes. In particular, we have focused on the construction and decoding problems. In both cases, we have selected one particular algorithm (the Extended Bit-Filling, and the Sum-Product Algorithm, respectively) as focus for our work. In addition, we have experimented with various novel ideas and extensions to the algorithms. Using our simulations software, we were able to test the validity of these ideas, to a certain extent

9.1 Open Problems

9.1.1 Ant Traversal Decoding

The problem of "ant patrolling" [65] on a graph is the exploration of a network by a "decentralised group of simple memoryless robotic agents." By using an extremely simple local rule, these agents cover the edges of the graph in a most uniform manner, such that each edge is visited equally frequently. This appears to be ideal for use as a SPA schedule, where the desire is precisely to cover the edges in a uniform manner, thereby avoiding (or, postponing) the traversal of cycles for as long as possible.

9.1.2 Avoid Going Round Cycles

Use local girth to stop decoding bits when their independence fails.

9.1.3 Strong Subcodes

The technique of Fig. 4.2 may be used to identify strongly protected bits in the code; note the *dips* in the plot. By viewing these positions as the bits of a subcode embedded within the larger code, and decoding on the larger code, we would expect very good bit-error performance (convergence) of these bits.

However, it remains to verify that this particular topology (for this particular example) is significant for this code. In other words, we would like to test that the strong bits do not vary according to the *codeword* transmitted (which, in this example, is the all-zero codeword). Unfortunately, time-restrictions prevented did not permit this experiment.

9.1.4 Graph-Based Encoding

Sparse Matrix operations are already the standard method for encoding LDPC codes in linear time. In this respect, we would like to attempt devicing a simple algorithm similar to the SPA decoder, which encodes using the Parity-Check matrix.

Gaussian Reduction reveals the location of the k information bits, which are likely to be distributed around H. By attaching the information to the bit nodes corresponding to these rows, and filling the remaining positions with neutral values, it might be possible to infer the remaining redundancy symbols by using a SPA-like algorithm. This technique is quite similar to that of [11], who describe an encoding method based on the decoding algorithm for the Binary Erasure Channel (BEC).

Our very limited attempt at this seemed to face difficulties with the majority ($m = N - k > k, R < 1$) of neutral symbols, and could not converge.

Appendix A

Approximated Discrete Log

The binary Discrete Logarithm problem, on 5 bits (i.e., $p = 10$ variables; $i = 5$ input, and $o = 5$ output) modulo 29 is characterized by the following look-up table 6. The table is written in compact notation, showing only the valid input/output combinations.

$$f(x) = \log_2(x) mod 29.$$

Note that the domain of the function is naturally abridged to $2^5 - 3$, due to the modular constraint. This naturally follows for the range of the function, which, due to the log function, is also undefined for input 0.

	Input					Output					τ_{DL}
1–32	0	0	0	0	0	-	-	-			-
33–64	0	0	0	0	1	0	0	0	0	0	33
65–96	0	0	0	1	0	0	0	0	0	1	66
97–128	0	0	0	1	1	0	0	1	0	1	102
129–160	0	0	1	0	0	0	0	0	1	0	131
161–192	0	0	1	0	1	1	0	1	1	0	183
193–224	0	0	1	1	0	0	0	1	1	0	199
225–256	0	0	1	1	1	0	1	1	0	0	237
257–288	0	1	0	0	0	0	0	0	1	1	260
289–320	0	1	0	0	1	0	1	0	1	0	299
321–352	0	1	0	1	0	1	0	1	1	1	344
353–384	0	1	0	1	1	1	1	0	0	1	378
385–416	0	1	1	0	0	0	1	0	0	1	394
417–448	0	1	1	0	1	1	0	0	1	0	437
449–480	0	1	1	1	0	0	1	1	0	1	462
481–512	0	1	1	1	1	1	1	0	1	1	508
513–544	1	0	0	0	0	0	0	1	0	0	517
545–576	1	0	0	0	1	1	0	1	0	1	566
577–608	1	0	0	1	0	0	1	0	1	1	588
609–640	1	0	0	1	1	0	1	0	0	1	618
641–672	1	0	1	0	0	1	1	0	0	0	665
673–704	1	0	1	0	1	1	0	0	0	1	690
705–736	1	0	1	1	0	1	1	0	1	0	731
737–768	1	0	1	1	1	1	0	1	0	0	757
769–800	1	1	0	0	0	0	1	0	0	0	777
801–832	1	1	0	0	1	1	0	0	0	0	817
833–864	1	1	0	1	0	1	0	0	1	1	852
865–896	1	1	0	1	1	0	1	1	1	1	880
897–928	1	1	1	0	0	0	1	1	1	0	911

Table A.1: Θ_{10} (abridged), $p = 10, o = 5$, truth table τ_{DL}.

Appendix B

Tools

All tools developed for this thesis share a common (text-based) user interface, with a prompt-based menu system. The prompt for the setting 'show' allows the user to select the amount of printout generated to the screen during execution. In normal use, the choice would be 0, for minimum output. From this, increasing numbers 1 to 3 are defined, where 3 indicates debugging mode (full output). At certain points, if show > 0, the program halts, and waits for the user to view the output before proceeding (and clearing the screen). At such points, the program is resumed by entering a small integer (e.g., 1).

Also, please note that the program will produce output which does not 'break' neatly over several lines. Hence, for correct result, it is important to maximise the window in which the program is running.

B.1 1: Augmented EBF

Our implementation of the Extended Bit-Filling Algorithm is described in detail in Ch. 4. The use of the program is relatively straight-forward, and assisted by understandable input-prompts.

First, one is asked for the main optimization parameter; girth or rank. The former attempts to construct a code of specific dimensions N and m (assuming full rank), such that the girth remains above a user-defined bound, g. The latter is quite similar, in that it also attempts to maintain girth above the minimum bound, but the process is aimed at maximising the number of columns added. **Note:** g is a *lower bound* on girth, which means that the software attempts to *avoid* cycles of length $\geq g$.

The software is designed to take as input a column-weight sequence, defining (in bits, not fractions) the weight of each column (or, bit). By declining the prompt, the uniform column weight is set to $\gamma = 3$. Otherwise, one is asked for this sequence, and the input is expected in groups; such that one specifies the size of a successive group of bits and their common weight. This way, it is possible to specify any irregular LDPC code.

As discussed, 'jumpBack' is one of the extensions to the EBF algorithm suggested in this thesis. By disabling this tool, the construction process restarts from the beginning immediately after a failure. Otherwise, by accepting it,

there are two thresholds to be set,[1] the minimum required columns successfully constructed to qualify as a 'basis' for further constructions; the distance the procedure jumps back before resuming (column i'); and, finally, the maximum number of times the algorithm tries to jump back before restarting from the beginning. Currently, the first two thresholds are both set to 50% of N, while the number of resumes is set to 30% of N.

Next, the user is prompted for the maximum number of 'attempts,' which is the total (including resumes using jumpBack) number of times the algorithm tries to construct the code. The final inputs are the lower and upper girth bounds, g and $\bar{g}$, respectively, and the maximum (fixed) row-weight, $\rho_{\max}$.[2]

Codefiles are stored in the `incomplete` directory by default, and must be moved to the `matrices` directory before they are accessible in the Code Library (se below). Also, by convention, the EBF algorithm is designed to optimise on the design parameter, so it may produce more than one file. Each output file is then named according to the naming scheme below, but with $N = 0$ (in the filename), so that files of the same optimisation process can be determined.[3]

Recommended output level for code optimisation construction is 0.

B.1.1 Shortcuts

To demonstrate the software, the toolset comes with a set of construction 'shortcuts,' as given by design parameters in Table 4.2. These constructions do not prompt the user for any design parameters, and is a quick way of viewing the EBF process.

Recommended output level for code construction is also 0, but can be changed to 1 to follow the details of the process.

B.1.2 No Optimisation

In the event that the user wishes to produce a code of specific dimensions *and* girth, and do not want the software to attempt optimising further, the workaround is, as suggested in [1], to set $g = \bar{g}$.

B.2 2: Code Library

Before proceeding to any other tools, it is important that a code is loaded into memory, from the 'code library' (the `matrices` directory). The program supports the `Alist` format,[4] and expects matrices (codes) to be stored in files which have the following naming scheme;

$$N\,.\,\gamma\,.\,\rho\,.\,g \quad .\mathrm{A}$$

[1] These are specified within the code, not at the prompt.

[2] The construction of irregular codes is limited to the column-weight sequence, yet this would not require a major change to the code to extend to include the row-weight sequence.

[3] This is well-defined for optimising on rank, where N is expected to vary, but perhaps less intuitive for girth, where N is fixed. Time did not permit ajdusting this feature.

[4] http://www.inference.phy.cam.ac.uk/mackay/codes/alist.html

B.3 4: SPA Decoder

The SPA decoder begins with the desired infoword. Due to restrictions in time, we have not completed the encoder, meaning that the program may only be used with the all-zero codeword (assuming a linear code), or a user-defined codeword.

The various schedules explored in this thesis are available for testing, and are selected according to the following system.

f for Flooding;

d for Dynamic (with further prompts for 'Random'; or 'Chained');

h for Hybrid (flooding interspersed with pivots);

a for Avoid 4-cycles;

z for Storing messages in vertices;

g for Tresholding (also, storing messages in vertices);

Next, the user is prompted for decoder timeout (max number of iterations), and SNR (in dB). Beyond this, the channel is modelled, adding noise to the codeword, which is then decoded. Recommended output level for single-codeword decoding is 1.

As a compile-time setting, the user can choose the 'message domain' (LR or LLR) by setting a parameter in the `Graph` class.

B.4 11: Channel Simulator

The simulation software will prompt for many of the same parameters described in the SPA decoder above. Essentially, the user selects a codeword (usually, the all-zero codeword), and sets the decoder settings (see above). Next, the simulator attempts to generate BER points with 95% confidence (i.e., sampling 100 bit-error events) over the range of 0 to 10.5 dB, with increments of 0.5 dB.[5]

The simulator produces the output both on screen and to file (if desired). These datafiles are placed in the `curves` directory, and follow the following naming scheme, where `filename` is defined above

```
filename - schedule timeout - seed _ id .dat
```

'Schedule' and 'timeout' are defined above, while 'seed' is set as

```
srand(static_cast<unsigned>(time(0)))
```

(which is denoted in the filename by 0). The seed can be changed at compile-time, within the code. Finally, the 'id' ensures that datafiles are never overwritten, in case of name-conflicts.

The final prompt, is whether to display the output from `ps u -p [pid]` on-screen, during simulation. This can be interesting to follow, especially while processing large codes which take long to decode. The data produced by the simulations software (to screen, and file) is described in Ch. 7.

[5]The software has a maximum of 10^7 messages simulated, to avoid infinite looping.

B.5 21: Check Girth

Using a exhaustive Depth-First traversal of the graph, the minimum girth is identified. For large codes, this may take a while to complete.

B.6 22: Draw Graph

Using the `dot` format, the software can produce a script-file for `neato` of the GraphViz package. The output is forced into standard LDPC bipartite form, but also a 'free' script is produced, which allows `neato` to determine the location of nodes. Both files are named by filename (see above), with the `.dot` suffix, and are stored in the `bitfill` directory.

```
> neato -Tps file.dot -o file.ps
```

B.7 Etcetera

In addition to the above software, a small set of applications were written, mainly to handle the codefiles.

B.7.1 Convert Maple - Alist

The `alist` format, defined by MacKay *et al.*, is a compact notation for representing sparse matrices. Rather than storing the entire $m \times N$ matrix in binary, only the non-zero entries are recorded. Also, for queries, the format contains a header, with the characteristic parameters: blocklength, N; redundancy, m; row- and column-weight sequences, $\rho(x)$ and $\gamma(x)$, respectively.

In this thesis, we have implemented this format, in which all codes constructed are stored. This way, the software and code-library is more easily accessible for further use. Also, obviously, our software may import codes from the comprehensive Encyclopedia of Sparse Graph Codes [41].

Some matrix operations–such as invertion, and Gaussian Reduction–are better left to professional software, such as Maple or Matlab, who do not support the `alist` format. To bridge this gap, we wrote two small tools to convert between `alist` and 'Maple matrix format.'[6]

`A_to_Maple.pl` and `Maple_to_A.pl` are both run via `Perl`, with the filename as input. Both scripts are to be run from the main directory, and expect to find the given file in the `matrices` or `mapleMatrices` directories, respectively. Below, is an example which converts the `alist` file to `Maple` format (for pasting into `Maple`),

```
> perl A_to_Maple.pl 816.3.6.6.A
```

[6]Matrices are represented as lists of lists, delimited by square brackets, and commas.

Bibliography

[1] J. Campello and D. Modha. Extended bit-filling and ldpc code design. *IEEE Trans. Inform. Theory*, 1:985–989, 2001.

[2] J. Campello, D. S. Modha, and S. Rajagopalan. Designing ldpc codes using bit-filling. *Proc. Int. Conf. Communications (ICC), Helsinki, Finland*, 2001.

[3] D. C. J. MacKay. David MacKay's Research group, Cavendish Laboratory, 2003. http://beta.metafaq.com/action/answer?aref=318654&id=MKCTHOUBRP5J1BOCB%01PLI077I.

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

[5] C. E. Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27:379–423, 623–656, 1948.

[6] C. Berroux, A. Glavieux, and P. Thitimajshima. Near shannon limit error-correcting coding and decoding: Turbo codes. *Proc. IEEE Intl. Conf. Commun. (ICC 93)*, pages 1064–1070, 1993.

[7] R. M. Tanner. A recursive approach to low complexity codes. *IEEE Trans. Inform. Theory*, 27:533–547, 1981.

[8] D. J. C. MacKay and R. M. Neal. Good codes based on very sparse matrices. *Cryptography and Coding 5th IMA Conf.*, pages 100–111, 1995.

[9] M. Sipser and D. A. Spielman. Expander codes. *IEEE Trans. Inform. Theory*, 42:1710–1722, 1996.

[10] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Spielman, and V. Stemann. Practical loss-resilient codes. *Proc. 29th Symp. on Theory of Computing*, pages 150–159, 1997.

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

[12] S. Chung, G. D. Forney, T. J. Richardson, and R. Urbanke. On the design of low-density parity-check codes within 0.0045 db of the shannon limit. *IEEE Comm. Letters*, 2:58–60, 2001.

[13] T. Nozawa. Ldpc adopted for use in comms, broadcasting, hdds. Nikkei Electonics Asia, 2005. http://neasia.nikkeibp.com/neasia/000828.

[14] B. Vasic and I. B. Djordjevic. Low-density parity check codes for long-haul optical communications systems. *IEEE Photonics Tech. Letters*, 14:1208–1210, 2002.

[15] C. Neumann, V. Roca, A. Francillon, and D. Furodet. Impacts of packet scheduling and packet loss distribution on fec performances: Observations and recommendations, 2005. http://www.inrialpes.fr/planete/people/roca/mcl/ldpc_infos.html.

[16] *Linear Algebra and its Applications*. Addison-Wesley Publishing Company, 2000.

[17] Shu Lin and Jr. Daniel J. Costello. *Error Control Coding*. Pearson, Prentice Hall, 2004.

[18] M. G. Parker C. Riera. On pivot orbits of boolean functions. 2005.

[19] A. Bouchet. Isotropic systems. *Eur. J. Comb.*, 8:231–244, 1987.

[20] L. E. Danielsen. On self-dual quantum codes, graphs, and boolean functions. Master's thesis, UiB, 2005.

[21] H. Loeliger. An introduction to factor graphs. *IEEE Signal Proc. Magazine*, 21:28–41, 2004.

[22] F. R. Kschischang, B. J. Frey, and H. Loeliger. Factor graphs and the sum-product algorithm. *IEEE. Trans. Inform. Theory*, 47:498–519, 2001.

[23] W. Ryan. An introduction to low-density parity-check codes. 2001. http://www.csee.wvu.edu/wcrl/papers/ldpc.pdf.

[24] R. Hill. *A First Course in Coding Theory*. Oxford University Press, 1986.

[25] Channel (communications). From Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Channel_%28communications%29.

[26] Electromagnetic spectrum. From Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Electromagnetic_spectrum.

[27] B. J. Frey and D. J. C. MacKay. A revolution: Belief propagation in graphs with cycles. *NIPS*, 10, 1998. http://www.cs.toronto.edu/~mackay/rev.ps.gz.

[28] Eric W. Weisstein. Gaussian function. From *MathWorld*–A Wolfram Web Resource. http://mathworld.wolfram.com/GaussianFunction.html.

[29] J. G. Proakis. *Digital Communications*. McGraw-Hill, 2000.

[30] M. Potužník and P. Hinow. Deterministic patterns in pseudorandom point sets. 1997. http://math.vanderbilt.edu/~hinopw/workshop_97.pdf.

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

[32] D. J. C. MacKay. Good error-correcting codes based on very sparse matrices. *IEEE Trans. Inform. Theory*, 45:399–431, 1999.

[33] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. A. Spielman. Improved low-density parity-check codes using irregular graphs. *IEEE Trans. Inform. Theory*, 47:585–598, 2001.

[34] J. Kim, U. N. Peled, I. Perepelitsa, V. Pless, and S. Friedland. Explicit constructions of ldpc codes with girth at least six. *IEEE Trans. Inform. Theory*, 50:2378–2388, 2004.

[35] V. Stolpman, C. Zhang, and N. Vanwaes. Irregular structured ldpc codes. 2004. http:www.ieee802.org/16/tge/contrib/C80216e-04_264.pdf.

[36] A. Prabhakar and K. Narayanan. Pseudo-random construction of low-density parity check codes using linear congruential sequences. Web page, 2002. http://www.ee.tamu.edu/~krn/PAPERS/lcldpc.pdf.

[37] D. J. C. MacKay, S. T. Wilson, and M. C. Davey. Comparisons of constructions of irregular gallager codes. *IEEE Trans. on Commun.*, 47:1449–1415, 1999.

[38] R. Urbanke. Lthc: Ldpcopt. Web application, 2005. http://lthcwww.epfl.ch/research/ldpcopt/.

[39] S. Chung. Density evolution applet. Web application (appears offline), 2003. http://lids.mit.edu/~sychung/de.html.

[40] Greedy algorithm. From Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Greedy_algorithm.

[41] D. C. J. MacKay. Encyclopedia of sparse graph codes, 2005. http://www.inference.phy.cam.ac.uk/mackay/codes/data.html.

[42] K. V. Rhein. Algorithms and complexity. http://www-1g.cs.luc.edu/~van/cs460/lecture8/.

[43] H. Xiao and A. Banihashemi. Graph-based message-passing schedules for decoding ldpc codes. *IEEE. Trans. on Commun.*, 52:2098–2105, 2004.

[44] S. Ikeda, T. Tanaka, and S. Amari. Information geometry of turbo and low-density parity-check codes. *IEEE Trans. Inform. Theory*, 50:1097–1114, 2004.

[45] T. J. Richardson and R. Urbanke. Efficient encoding of low-density parity-check codes. *IEEE Trans. on Inform. Theory*, 47:638–656, 2001.

[46] C. P. Shelton. Lecture Notes, 1999. http://www.ece.cmu.edu/~koopman/des_s99/coding.

[47] V. Nagarajan. Lecture Notes. http://ece-www.colorado.edu/~milenkov/class.ppt.

[48] X. Wu, H. R. Sadjadpour, and Z. Tian. A new adaptive two-stage maximum-likelihood decoding algorithm for linear block codes. *IEEE. Trans. on Commun.*, 53:909–913, 2005.

[49] Decoding methods. From Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Syndrome_decoding.

[50] D. C. J. MacKay. David MacKay's Research group, Cavendish Laboratory, 2003. http://beta.metafaq.com/action/answer?aref=318380&id=MKCTHOUBRP5J1BOCB%01PLI077I.

[51] A. E. Pusane, M. Lentmaier, T. E. Fuja, K. S. Zigangirov, and D. J. Costello. Multilevel coding/modulation using ldpc convolution codes. *Intl. Symp. on Inf. Theory and Its Appl., ISITA2004*, 2004.

[52] H. S. Cronie. Signal constellations for multilevel coded modulation with sparse graph codes. http://www.sas.el.utwente.nl/publications/download/165.pdf.

[53] S. M. Moser. Investigation of algebraic codes of small block length using factor graphs. Master's thesis, ETH, 1999.

[54] Indicator function. From Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Indicator_function.

[55] P. Vontobel and R. Koetter. On the relationship between linear programming decoding and min-sum lagorithm decoding, 2004. http://citeseer.ist.psu.edu/vontobel104relationship.html.

[56] Information entropy. From Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Information_entropy.

[57] A. Baldman. Bit error ratio testing: How many bits are enough? Web page, 2003. http://www.iol.unh.edu/training/ethernet/BER-How_Many_Bits_18Mar2003.p%df.

[58] J. E. Gilley. Bit-error-rate simulation using matlab. Web page, 2003. http://www.transcrypt.com/download?id=7550.

[59] T. Summers. Ldpc: Another key step towards shannon. Web page. http://www.commsdesign.com/design_corner/showArticle.jhtml?articleID=4%9901136.

[60] X. Hu and M. P. C. Fossorier. On the computation of the minimum distance of low-density parity-check codes. *IEEE Intl. Conf. on Comm*, 2004.

[61] Error floors of ldpc codes. *Proc. 41st Allerton Conf. Comm., Contr. and Comp.*, 2003.

[62] E. Guizzo. Closing in on the perfect code. *IEEE Spectrum*, 2004.

[63] J. H. Weber and K. A. S. Abdel-Ghaffar. Stopping set analysis for hamming codes. *IEEE Proc. of IEEEISOC ITW2005 on Coding and Complexity*, pages 244–247, 2005.

[64] Pivot. From Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Pivot.

[65] V. Yanovski, I. A. Wagner, and A. M. bruckstein. A distributed ant algorithm for efficiently patrolling a network. *Algorithmica*, 37:165–186, 2003.