Analysis and Design of Protograph based LDPC Codes and Ensembles

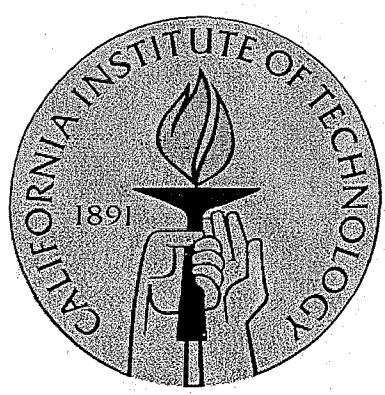
Thesis by

Jeremy Thorpe

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy



California Institute of Technology

Pasadena, California

2005

(Submitted May 25, 2005)

To my family

Acknowledgements

I wish to express my sincere thanks to many people for making this thesis possible. First, I would like to thank my advisor, Dr. Robert McEliece, who has supported my research in so many ways since the first summer that I arrived at Caltech. Throughout my graduate studies, Dr. McEliece has patiently helped me to express and clarify my ideas. In so many ways, he has helped me to put my research into larger contexts, by drawing out similarities and contrasts to other work, and simply through his knowledge of which problems are important, and where ideas can be applied. In addition, he has been extremely generous in helping me to have opportunities to work outside of Caltech. During the summers, I had opportunities to work at JPL, Sony research labs and Microsoft research, none of which would have been possible without his support and encouragement to apply. I am grateful beyond words for his enduring support.

At JPL, where much of the work contained in this thesis was done, there are many that I would like to thank. Fabrizio Pollara and Jon Hamkins, my former and current supervisor, have always given my work enthusiastic support. Ken Andrews and Sam Dolinar have collaborated with me since my first summer, sharing their ideas and patiently listening to mine. I would especially like to thank Sam for the huge amount of support he gave me in writing chapter 2 of this thesis. More recently, I have had the privilege of collaborating with Dariush Divsalar on code design. I have also enjoyed many conversations directly and indirectly related to my research with Matt Klimesh, Aaron Kiely, Bruice Moison, Chris Jones, Shervin Shambayati, Jason Lee, and others.

I would like to sincerely thank my supervisor Masayuki Hattori for generously inviting me to spend a summer at Sony Information & Network Technologies Lab. While there, I had the opportunity to interact with and learn from Toshiyuki Miyauchi, Kouhei Yamamoto, and many others.

Spending a summer at Microsoft Research Redmond has also broadened the scope of my research. I had the opportunity to do with Dimitris Achlioptas, my mentor during an internship in summer 2003. Dimitris has helped me to realize the value of clear scientific writing, which I hope has carried over into other areas of my own writing.

I would like to thank my lab-mates and the many friends I have made at Caltech for the many conversations about math and science, as well as for sharing the laughs and enjoyment that made my experience at Caltech more pleasant. I have valued the conversations with my lab-mates Hui

Jin, Aamod Kandekar, Ravi Palanki, Cedric Florens, Jonathan Harel, Radhika Gowaikar, Masoud Sharif, Amir Farajidana, Yindi Jing, as much as any other aspect of being at Caltech.

Finally, I would like to thank my family for their steadfast support. My dad has enthusiastically allowed me to explain my ideas, even when my explanations were inadequate, as he always has. His readiness to always take on new challenges is an inspiration to me. My mom will finish her Ph.D. this year, virtually simultaneously with me, and talking to her about our theses has given me renewed determination to finish mine more than once.

Abstract

Channel coding is a key component of artificial communication systems, allowing reliable communication using unreliable channels. In the last decade, iteratively decoded channel codes have become or clearly will become standards in a wide range of applications where large amounts of information must be communicated using unreliable media. Of the class of iteratively decoded codes, Low-density parity check (LDPC) codes are arguably the simplest class to describe, and indeed were described more than four decades ago in 1963 by Robert Gallager.

The current understanding of LDPC codes has progressed in several significant ways beyond what had been expressed by 1963 by Gallager. Importantly, irregular LDPC codes, whose parity check matrices do not have constant row and column sums, have been shown to significantly outperform their regular counterparts explicitly considered by Gallager.

By 1999, researchers had defined a class of irregular ensembles, each characterized by a pair of polynomials. Along with this new class of ensemble, they defined an analytical technique, density evolution, that accurately predicted the channel coding performance of a typical code under iterative message-passing decoding. The pair of polynomials could be effectively be designed by optimizing the coefficients of the polynomial for density evolution threshold threshold.

This thesis concerns a different class of ensembles, namely protograph ensembles. Protograph ensembles are characterized by a template graph called, intuitively, a protograph. The Tanner-graph representation of a code in the ensemble is a random lift of the protograph.

Protograph-based codes have significant advantages over unstructured irregular codes in regard to implementation of their encoders and decoders. In the decoder, this structure can be used in at least two distinct ways to organize the computations defined by any message-passing algorithm. If, in addition to the protograph structure, circulant structure is imposed on each "section" of the matrix then a quasi-cyclic code results, bestowing even mores advantages, especially in the possible implementation of the encoder.

A central difficulty in using protograph ensembles is finding a suitable protograph. Since graphs are discrete objects, there is no obvious correspondence to any optimization model using vectors of real numbers. Instead, the technique of simulated annealing has been applied with a remarkable degree of success. For example, on the AWGN channel, given a constraint on the node degrees,

protograph ensembles can be found that achieve a threshold only half as far (measured in dB) from the Shannon limit as unstructured irregular ensembles. This simultaneously illustrates an inherent performance advantage of protograph codes over unstructured codes as well as the efficacy of simulated annealing as an optimization technique.

A persistent problem which appears to be common in all codes optimized for density evolution threshold is that of error floors. On a superficial level, this is explained by the maxim that "There's no such thing as a free lunch." In some contexts, such as in codes designed for the erasure channel, the phenomenon can be explained on a much deeper level, though it is not clear why the phenomenon should persist so universally.

Still, even without a detailed understanding the cause of this problem, there are techniques that can mitigate error floors. An important tool toward this end is weight enumerators, which are discussed in chapter 3. Codeword and stopping set enumerators can be efficiently computed if a certain (non-concave) function can be efficiently maximized. Protographs that are selected on the basis of their enumerators have shown some success in reducing error floors.

Contents

List of Figures

List of Tables

Chapter 1

Introduction to LDPC Codes

The fundamental problem of communication os that of reproducing at one point ... a message selected at another point.

-Claude E. Shannon

1.1 Natural and Artificial Communication Systems

Nature has endowed animals with the ability to *communicate* with each other in a plethora of ways. Bees dance to indicate the location of a new food source. Plants use chemical signals to communicate danger sensed in the environment. Woodpeckers bang on trees to communicate unknown messages with distant friends. Humans transmit even more complex thoughts to each other via vibrations produced in the throat.

It is not surprising then that one of the great challenges of our day is engineering systems capable of communication. While perhaps not as diverse as natural biological communication systems, artificial communication systems have already achieved their own remarkable success. As I am writing this paragraph, a network of copper and glass threads carries my thoughts halfway around the world. Browsing the web, I see that one of two twin rovers has just sent back a panoramic Mars-scape from 300,000,000 kilometers away, through empty space.

Communication, biological or artificial, involves agreements between the two communicating agents. These agreements can be simple or complex. In simple systems, such those used by some plants, only a few different messages or even just a single message. To an insect, a particular chemical may mean "danger" perhaps, or "let's mate." In this case, the system may be thought of as having only one level of agreement, a direct mapping from a physical signal to a meaning.

More complex systems are capable of communicating many more messages, and consequently use more levels of agreement. Spoken human languages are, at a low level, each based on an agreement to use a certain set of phonemes that the speaker can produce with his throat, and listener can can distinguish with his ears. At a higher level, a shared vocabulary maps combinations of phonemes to

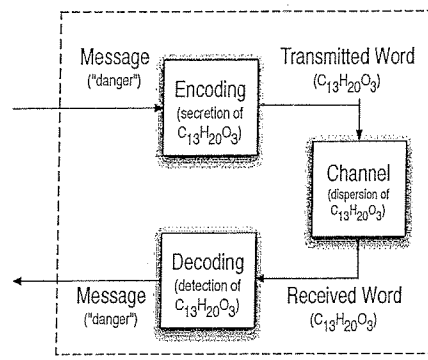


Figure 1.1: An abstraction of a communication system found in tobacco plants

words identifying concise concepts. A shared grammar maps complex thoughts to strings of words.

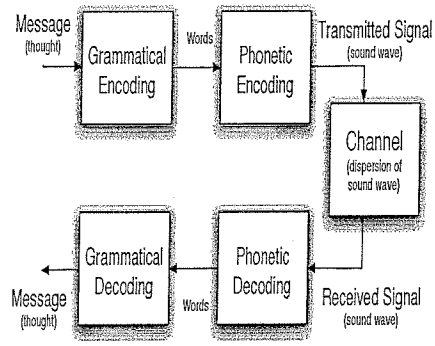


Figure 1.2: A Human Communication System

This use of abstraction, in which multiple levels of agreement are used, also allows one component to be replaced with another. Phonemes can be represented by written symbols instead of pressure waves, making use of the same vocabulary and grammar as in the spoken system. This is the case in Spanish and other phonetically written languages. Other languages, such as Chinese represent words directly, rather than phonemes. In either case, one part of the communication system has been replaced by another.

In Artificial systems, abstraction is useful because it allows the many problems faced by engineers to be solved one by one. The Open Systems Interconnect (OSI) framework is a model for a system to enable communication between any two computers located almost any place on the Earth. Such a system must make use of a diverse set of physical resources, and solve a host of problems related to unreliable physical channels, unreliable hardware, unknown location of the recipient, and unknown demand on the system, and other issues.

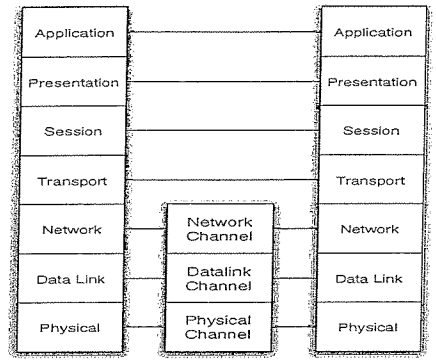


Figure 1.3: The OSI Stack Abstraction

1.2 The Role of Channel Coding

In the framework of channel coding, communication is accomplished via a stochastic system called a channel. The channel allows *transmission* by taking input from a sender and providing output, depending stochastically on the input, to a receiver. The goal of channel coding is to provide a more reliable channel by using the less reliable one.

In the OSI model, channel coding is often used at the physical layer, the lowest layer defined in that model. At this level, uncertainty can be introduced by physical processes such as thermal noise, by parameter mis-estimation, or by deliberate jamming [1].

However, channel coding can be applied not only at the physical layer, but also at higher levels to deal with uncertainty caused by lower layers of the protocol stack. Codes designed to efficiently handle erasures [2] have been applied at the Network layer to mitigate the effect of dropped packets.

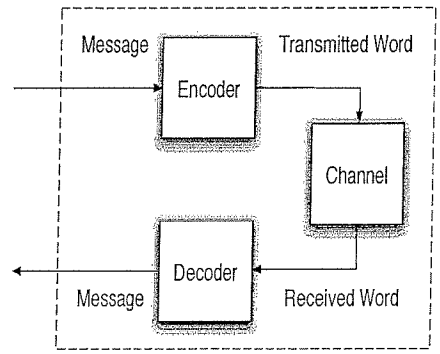


Figure 1.4: An abstract communication system

A channel-coding system, which itself satisfies the definition of a channel, is comprised of an inner channel, and two systems called an encoder and a decoder, connected as in figure 1.2. Usually, the inner channel is considered to be given, and the encoder and decoder should be designed. A successful

design is such that the decoder can *reliably* reconstruct any message that might be provided to the encoder.

In general, a channel can be any system whose output depends stochastically on its input. In fact, this definition is broad enough to include any possible system useful for communication. Two further restrictions on this definition add a great deal of tractability: time-invariance and memoryless-ness.

Memoryless-ness implies that the channel can be used at discrete times and that the output y_i at time i depends stochastically only on the input x_i at the same time, and is independent of all other inputs and outputs of the system. Formally, this means that there is a conditional probability $P(y \mid x)$ that can be expressed:

$$P(y \mid x) = \prod_i P(y_i \mid x_i) \tag{1.1}$$

Given that a channel is memoryless, time-invariance means that the output y_i at time i depends stochastically on x_i in a way that is independent of the time i. This dependence can be characterized by a stochastic matrix $A = (a_{x,y})$ such that:

$$P(Y_i = y | X_i = x) = a_{x,y} \tag{1.2}$$

In reality, most channels are neither memoryless nor time-invariant. Still, there are techniques that can make a large class of channels fit, or nearly fit, this model. Parameter estimation techniques such as Kalman filtering can drastically reduce the uncertainty in the receiver's knowledge in the parameters of a time-varying channel, provided that the parameters change slowly enough. Signal processing techniques such as pre- and post- filtering can turn an inter-symbol-interference channel into an essentially memoryless one. Such practices are the subjects of much study, but from the perspective of this thesis, we will henceforth ignore them and consider only the narrow class of channels.

Shannon's famous Channel Coding theorem [3] gives an upper bound on the amount of information (which can be measured in bits) that can be reliably transmitted making use of any time-invariant memoryless channel n times. This bound is $C(A) \cdot n$, where $C(A)$ is a number called the capacity that depends on the channel. Shannon also gives the converse assurance that it is possible to transmit information at arbitrarily close to this rate, and with arbitrarily small probability of error. A coding system which can transmit a fraction arbitrarily close to 1 of $C(A)$ bits per channel use with arbitrarily low is said to achieve the capacity of the channel.

1.3 Linear Codes

To define the communication problem fully, we must have a model of the information that should be transmitted. A standard model is that there are k symbols chosen at random, equiprobably from among the symbols of a particular alphabet.

In practice, messages are much more diverse. Messages representing images, sounds, and text all have inherent redundancy which reduce the amount of information (or entropy) in the message. In principle, it is possible that a channel-coding system could use this redundancy to decrease the number of times the channel must be used.

However, in general the problem can be split into two processes. The first process that occurs in the encoding is called source coding, in which the redundancy of the message is essentially removed, and a shorter representation is derived. This process yields a string of symbols which approximately conforms to our model of k equiprobable symbols. Shannon's Separation Theorem tells us that this is possible in general.

A further simplifying assumption is that the alphabet from which our source is chosen is of arbitrary size. A very common convention is to let the alphabet be the binary alphabet $\{0, 1\}$, so that the transmitted messages m is in $\{0, 1\}^k$.

In the framework of linear codes, we identify the input alphabet with a finite field F. In the binary case, it is possible to let the field $F = F_2$ be the finite field with 2 elements. We identify the string $m \in \{0, 1\}^k$ with a corresponding vector F_2^k.

For the purpose of linear codes, even the class of memoryless time-invariant channels is not sufficiently restricted. We need the further restriction that the channel input can be mapped to the same finite-field to which we map the messages. This implies that the channel input alphabet be the same size as the message alphabet and to some finite field.

Such a channel can be created from a large class of memoryless channels by the use of a modulation scheme. For example, various modulation schemes such as phase-amplitude modulation can provide a binary input channel by using a complex-input additive white gaussian noise (AWGN) channel. Channel modulation represents a large field of study, but from this point, we will assume that the channel has an input alphabet of a size that is convenient for us, and can be mapped to, and identified with, the finite field of our choice.

Given the channel which takes as its input elements of a finite field, a common technique is to use the channel some fixed number n times.

A code $\mathbb{C}$ defines a set of valid vector channel inputs. A linear code is a subspace of the vector space F^n with dimension k. Under the theory of linear algebra, this implies that there exist matrices H, of dimension $n \times r$, and G, of dimension $n \times k$, such that $k + r = n$ and

$$\underset{\mathbf{c}\in\mathbb{C}}{\forall}\{H\cdot\mathbf{c}=\mathbf{0}\} \tag{1.3}$$

The matrix G defines a mapping from message vectors m to channel input vectors given by $\mathbf{x}=\mathbf{m}G^T$.

$$\mathbb{C}=\{\mathbf{m}G^T\} \tag{1.4}$$

Since the (scalar) channel is used n times to transmit k symbols of information, the system can transmit $\frac{k}{n}$ symbols of information per use of the channel, and this quantity is called the code rate R.

1.4 Low Density Parity Check Codes

Low-density parity check (LDPC) Codes are a type of linear code in which there is at least one low-density parity check matrix H. The term low-density is loosely defined, but it usually indicates that there are a number of non-zero terms is roughly proportional to the number of rows in the matrix. By contrast, a random code of rate R where $0 < R < 1$ has lowest weight parity check matrix whose number of non-zero entries grows as $O(n^2)$. Formally, the concept of low-weight can be defined in terms of a growing ensemble of codes, which will be introduced in chapter 2.

1.5 LDPC Codes as Graphs

For many applications, it is useful to define a structure related to LDPC codes called a Tanner graph[4] which we will call $\mathbb{G}$. This structure represents a particular parity check matrix H.

$\mathbb{G}=(V,C,E)$ is constructed as a bipartite graph with a set of nodes V representing each variable in the code.

And LDPC code is characterized by a bipartite graph $\mathbb{G}=(C,V,E)$, where $V,|V|=r$ is a set of variable nodes $C,|C|=n$ is a set of check nodes, and E is a set of edges that have one endpoint in C and one endpoint in V. A vector $X\in F(2)^n$, indexed by the elements of V, is a codeword if and only if$\sum_{c\backslash v} X_v=0\forall c\in C$. The notation $v|c$ means any $v\in V$ connected to c via an edge $e\in E$.

The importance of $\mathbb{G}$ cannot be underestimated. Indeed, there is a fair amount of doubt over whether an LDPC code ought to be considered equivalent to its the linear subspace it defines, or to its graph. The traditional view, and the view that most authors are careful to adhere to, is that it is equivalent to the subspace.

Indeed, this view is sufficient to characterize many important aspects such as its minimum distance, codeword weight enumerator, and performance under maximum likelihood decoding.

However, many fundamental properties associated with LDPC codes are not defined by its subspace. Message passing algorithms including belief propagation (BP) algorithm (section 1.8), can be defined only in terms of $\mathbb{G}$ or H and not in terms of subspaces. Stopping sets, which are related to message-passing decoding algorithms, can only be defined in terms of $\mathbb{G}$ or H. The analytical tool of density evolution (section 2.5) is defined with respect to ensembles of parity-check matrices, or ensembles of graphs, but not ensembles of subspaces.

Thus, from the perspective of this thesis, we will always take the definition of an LDPC code to be its graph $\mathbb{G}$ or, equivalently, a particular parity check matrix H and not just the subspace defined by H.

1.6 Encoding of LDPC Codes

Since LDPC codes are defined by their parity check matrices, it is not always a trivial to encode them, nor is there always a particularly efficient way. The encoding procedure defined by Richardson and Urbanke [5] works quite well for optimized distributions, in which case the complexity is nearly linear, but not as well for comparatively high-density check matrices (for example, minimum variable node degree 3 or more). In the case of circulant based codes 4.3, certain techniques can drastically reduce hardware complexity.

This section describes what is perhaps the simplest algorithm for encoding LDPC codes, constructing an explicit generator matrix. The technique is to first construct a systematic parity check matrix of the form:

$$H_{systematic} = [H_{dense} \mid I_r] \tag{1.5}$$

using Gaussian elimination. The elementary operations are to add one row to another, swap rows, and swap columns. Since the order of columns is meaningful, if columns are swapped during this stage, they must be swapped in the original matrix as well. Redundant rows of H must also be removed.

Given $H_{systematic}$ a systematic encoding matrix $G_{systematic}$ can be defined:

$$G_{systematic} = [I_k \mid H_{dense}^T] \tag{1.6}$$

This systematic matrix defines a systematic encoding given by:

$$x = G_{systematic}^T \cdot m \tag{1.7}$$

1.7 Decoding Framework

This section presents a standard decoding framework, which formalizes the assumptions made by the belief propagation decoder and defines terms used therein. This framework consists of several component systems, signals which flow between the systems, and a set of assumptions on the component systems and nominal values on some of the signals. The component systems are an encoder, a channel, a channel demodulator, a probabilistic decoder, and a de-encoder. All of the systems are abstract, and each has some degree of freedom, though the flexibility in the code, channel, and decoder account for the range of possible system performance.

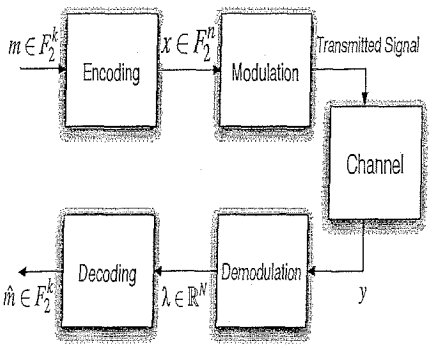


Figure 1.5: Generalized Decoding Framework

Figure 1.7 shows the flow chart defined by this decoding framework. The single input to the system, a message denoted m, is supposed to come from a uniform source over F_2^k. The encoder is an arbitrary $1:1$ linear mapping from F_2^k to $\mathbb{C}$. Since the dimension of the code is k, the same as that of the message, at least one such mapping always exists. Given that m is uniformly random, it is immediate that the decoder output is also uniformly random.

The codeword x is input to, or *transmitted over* the channel. The channel is assumed to be time-invariant and memoryless. Additionally, it is assumed to take binary input. It is not assumed at this point that the channel is symmetric, though this assumption will be made later for the purpose of density evolution analysis (section 2.5). The channel output is $y \in \mathbb{Y}^n$.

The demodulator provides an interface from the channel to the probabilistic decoder, observing the channel output and providing a representation suitable for the probabilistic decoder.

$$\lambda_i = \frac{P(y_i|X_i = 1)}{P(y_i|X_i = 0)} \tag{1.8}$$

The demodulator is assumed to know the channel statistics, as it must in order to evaluate 1.8. The presence of the demodulator allows the decoder to function without needing to know anything about the channel itself.

The decoder observes the likelihood ratios λ and produces an output $\hat{x} \in F_2^n$ whose nominal value is x, the output of the encoder.

Finally, the de-encoder inverts the operation of the encoder, producing an estimated message $\hat{m} \in F_2^k$. If $\hat{x} = x$, then it may be assumed that $\hat{m} = m$. Any pseudo-inverse of the encoding mapping has this property and thus is suitable to function as the de-encoder. In the case of a systematic encoder, the preferred de-encoder is the linear mapping which simply truncates the non-systematic portion of the codeword.

1.8 Belief Propagation Decoding

This section describes the most standard message-passing decoding algorithm, a specific instance of a much more general algorithm known as Belief Propagation (BP). This algorithm is due, in its present generality, to Pearl [6]. Another view of this algorithm, written in terms of a Generalized Distributive Law, is provided by Aji et. al [7]. However, the derivation of this specific form of BP from any generalized framework is beyond the scope of this section and this thesis.

The belief-propagation (BP) algorithm estimates X given λ. The algorithm functions by passing messages along the edges of $\mathbb{G}$for a series of iterations $i \in \{0, 1, 2, ..., i_{max}\}$. Each iteration is broken into two parts. In the first part of each iteration i and for each edge $e \in E$, a message λ_e^i is computed and passed along e from variable to check. Each such message represents an estimate of a certain conditional probability on x_e. x_e is the variable associated with e.

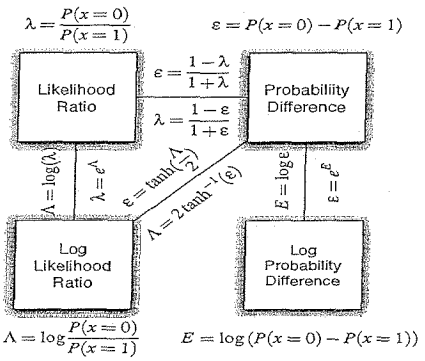


Figure 1.6: Four representations of the distribution of a binary random variable

1.9 Design Criteria

The primary design criteria for any channel coding system is its effectiveness in providing a reliable communication channel. The first, and most often cited, measure of this criteria is simply the

probability of error at the decoder output. A second measure, also of high importance in many applications, is probability of an *undetected* error at the decoder output.

It is customary to make direct comparisons only between codes of the same rate and length. This is because, for a fixed channel, it is generally easier to design codes that have low probability of error when the code has longer length (given a fixed rate), or lower rate (given a fixed length).

The channel coding theorem and *** theorem make this precise by bounding output error probability of a channel coding system as a function of rate and length. Shannon's channel coding theorem gives a lower bound on the probability of decoder error as a function of the code rate and the channel which is asymptotically independent of the length, though this bound is trivial (equal to 0) when the code rate is below the channel capacity. The error exponent of a channel implies an asymptotic bound on the probability of error that is exponentially small in the code length.

Since we may use the same coding system on different channels, it is often reasonable to compare performance on a family of channels. An example of such a family is the additive white Gaussian noise (AWGN) channel, sometimes referred to as the "deep-space channel" since almost any other channel is approximated comparatively badly by the model. The AWGN channel is parameterized by its signal-to-noise ratio (SNR), and thus a very common way to characterize the channel coding performance is to plot probability of error at the decoder output versus SNR. Most of the time, signal to noise ratio is normalized such that the signal power is measured per symbol of information rather than per channel use, i.e. to characterize SNR by E_b/N_0.

Chapter 2

LDPC Codes Constructed From Protographs

in which we define the class of protograph-based LDPC code ensembles

We introduce a new class of LDPC codes constructed from a template graph called a protograph. This construction allows the creation of LDPC codes of any natural multiple of the protograph size. In section 2.1, we define this construction in both its deterministic and randomized form.

The randomized construction defines an ensemble of codes of any particular length. In section 2.3, we discuss the properties of these ensembles. We give an equivalence relation on sequences of ensembles. We show that under this equivalence relation, there are protograph ensembles that are equivalent to any unstructured regular ensemble, but none that are equivalent to any unstructured irregular ensemble.

Some aspects of typical protograph LDPC codes can be inferred from the protograph. In section 2.5 we see that the performance under certain message-passing decoding algorithms can be predicted by analyzing the protograph via Density Evolution, an indispensable tool developed by Richardson and Urbanke. Density evolution gives a lower bound on the fidelity of the channel necessary for reliable communication. Indeed this bound is achievable provided that one is willing to use a large enough code.

In chapter 3, we see how the asymptotic codeword weight enumerator and asymptotic stopping-set weight enumerator can be computed for any protograph. Weight enumerators and stopping set enumerators give bounds on performance under both maximum likelihood decoding and, in certain cases, under message passing decoding.

Density evolution threshold and asymptotic enumerators reveal a great deal about the suitability of a typical code (of given size) in the ensemble to an application. In chapter 4, we see one way to use these properties to select among several possible protographs. The problem of searching for protographs is more difficult, but simulated annealing can be used to find protographs in a large class.

2.1 Introduction

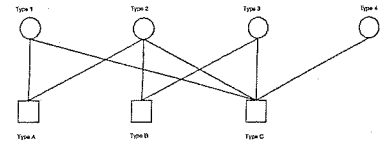
Recently, more and more sophisticated classes of LDPC codes have been forwarded by members of the research community, each offering advances in one area or another.

We have seen in chapter 1 that an LDPC code is described by its (sparse) parity-check matrix. Such matrices can be efficiently represented by a bipartite (Tanner) graph. The standard iterative decoding algorithm, known as Belief Propagation (BP) passes messages along the edges of this graph. Much research has gone into understanding the properties required of a Tanner graph to produce an LDPC Code that performs well under this decoding algorithm.

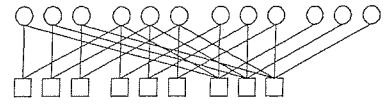
In this paper, we introduce a new class of LDPC codes constructed from a template called a protograph. The protograph serves as a blueprint for constructing LDPC codes of arbitrary size whose performance can predicted by analyzing the protograph. We apply standard Density Evolution techniques to predict the performance of large protograph codes. Finally, we use a randomized search algorithm to find good protographs.

2.2 Protographs and Protograph Codes

A protograph can be any Tanner graph, typically one with a relatively small number of nodes. A protograph $G = (V, C, E)$ consists of a set of variable nodes V, a set of check nodes C, and a set of edges E. Each edge $e = (v, c) \in E$ connects a variable node $v \in V$ to a check node $c \in C$

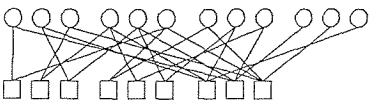


As a simple example, we consider the protograph shown in figure 1. This graph consists of $|V| = 4$ variable nodes and $|C| = 3$ check nodes, connected by $|E| = 8$ edges. By itself, this graph may be recognized as the Tanner graph of a $(n = 4, k = 1)$ LDPC code, though not necessarily an interesting one.



caption

We can obtain a larger graph by a "copy and permute" operation, illustrated in figure 2 and 3. In figure 2, the protograph has been copied 3 times.



caption

In figure 3, the endpoints of the edges corresponding to *each edge in the protograph* have been permuted. We call this the *derived graph*, which is the Tanner graph of an $(n = 12, k = 3)$ LDPC code.

In general, we can apply the "copy and permute" operation to any protograph, to obtain derived graphs of different sizes. This operation consists of making T copies of each variable and check node in the protograph and, for each edge in the protograph, adding T edges between corresponding nodes.

Definition 2.2.1 *e=mc2*

Definition 2.2.2 *A protograph code is an LDPC code whose Tanner graph is a derived graph.*

The usual mapping of Tanner graphs to LDPC codes makes the implicit assumption that each variable defined in the code will be transmitted over a channel. However, a useful refinement[8] is to allow the variable node set V to contain untransmitted variable nodes. Under this refinement, each variable $v \in V$ may be designated a *transmitted* node or an *untransmitted* node. The number of transmitted nodes is denoted n, and the number of untransmitted nodes is denoted u, thus $n + u = |V|$. The number of check nodes is denoted $r = |C|$. The dimension k of a code with untransmitted variables is $k = n+u-r$, and the rate is $R = \frac{n+u-r}{n}$. A variable node (v, t) in G' has the same transmitted/untransmitted designation as v. The derived graph contains nT transmitted and uT untransmitted variable nodes, as well as rT check nodes. Thus, any derived graph has the same rate R as its protograph.

Untransmitted variables can improve the performance of protograph codes. These variables are decoded by the decoder given the channel had yielded no information about, as in an erasure.

2.3 Equivalence among Ensembles

LDPC code ensembles can be partly characterized by their distribution on local neighborhoods. In this section, we define an equivalence relation on based on this characterization.

After i iterations of the decoding algorithm,

Definition 2.3.1

$$T_d = \{\} \tag{2.1}$$

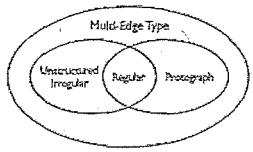


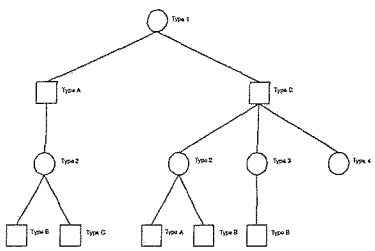
Figure 2.1: Classes of ensemble characterized by local neighborhood distribution

$$\sum_{t \in T_d} P(t_v = t) = 1 \tag{2.2}$$

Members of the same class have the same density evolution threshold.

2.4 Deterministic Neighborhoods

A property of protograph codes not shared by other classes of irregular codes is that the local neighborhood of a node (v, t) in G is completely determined by G. The local neighborhood to depth d consists of all nodes and edges connected to (v, t) by a path of length d or less. This neighborhood is a tree if there is at most one path of length d or less to any other node. In this tree, each node is adjacent to the same node types as in the protograph. To illustrate, we expand the neighborhood of a variable v of type 1 in figure 1 to depth $D = 3$



graphs optimized primarily for loopiness (left) and wire-length (right)

The local neighborhood tree can be constructed uniquely from the root downward. This leads to the following property of local neighborhood of derived graphs:

Property if a neighborhood of depth d of a variable node (v, t) in the derived graph $G'$is tree-like, its structure is determined by the adjacencies in the protograph G. Note that v need not be tree-like within depth d in G.

We can expect that the performance of belief propagation on derived graphs should be relatively insensitive to the choice of $\{\pi_e\}$ because of the structure imposed by G. By contrast, in a Tanner

graph generated from a given irregular degree profile, the structure of such a neighborhood is random, and the performance of Belief Propagation may depend more on the particular graph.

2.5 Density Evolution Analysis of Protograph Code Ensembles

We have defined an arbitrary protograph code by applying a "copy and permute" operation to a protograph G. We now define an ensemble of protograph codes by specifying a probability distribution over the permutations $\{\pi_e, e \in E\}$. The ensemble that we shall consider is that obtained by generating $\pi_e$uniformly over all permutations of length T, and independently for each $e \in E$. For this ensemble, it can be shown that as $T \rightarrow \infty$, the probability that the neighborhood of fixed depth d of any node (v, t) is tree-like goes to 1, and thus the ensemble meets the criteria for analysis by Density Evolution.

Density evolution (DE) analysis[9] can be carried out on a protograph to determine whether or not decoding will yield arbitrarily small bit error probability on a large derived graph. In this technique, the messages which are defined in the message passing algorithm are treated as random variables, and their distributions are computed.

For the ensemble of codes constructed from a given protograph G, and for a given symmetric channel C, we compute two distributions $\overrightarrow{q}_e^i$ and $\overleftarrow{q}_e^i$ on the messages passed in either direction along edge e at iteration i. Because of the deterministic neighborhood property, the message distributions $\overrightarrow{q}_e^i$ and $\overleftarrow{q}_e^i$ computed for the protograph edge e in G are valid for *any* corresponding edge (e, t) in the derived graph G' as long as the neighborhood of depth $2i$ is tree-like.

DE predicts a probability of decoder error $P_v^{(i)}$ for variable nodes of each type v at each iteration i, based on the distributions $\overrightarrow{q}_e^i$, $\overleftarrow{q}_e^i$, and the channel C. If message densities evolve such that $P_v^{(i)} \rightarrow 0$ as $i \rightarrow \infty$ for all v, then DE predicts that decoding will be successful, given T large enough. For a channel C_θ, whose fidelity increases with the parameter θ , the DE threshold θ^* for protograph G is the infimum of θ such that DE predicts successful decoding for G and C_θ.

2.6 The Reciprocal-Channel Approximation to Density Evolution

Our algorithm essentially relies on a very fast and accurate approximation to density evolution originally proposed in chapter 7 of S.Y. Chung's thesis [10] for regular LDPC codes called the reciprocal-channel approximation (RCA). Like other approximations to density evolution, the reciprocal channel approximation makes use of a single (real valued) parameter which approximately

characterizes the message distribution. We may suppose that this parameter represents a quantity which is additive at the check nodes (meaning that the parameter describing an outgoing message is the sum of those describing the incoming messages).

For the Gaussian channel, the statistic is the signal energy E normalized so that $N_0 = 1$. Let $C(E)$ denote the capacity of the binary-input AWGN channel of with input energy E, and C^{-1} its inverse. We define a reciprocal energy function $\Phi : E \longmapsto C^{-1}(1 - C(E))$, which is self-inverse. The dual domain statistic $E' = \Phi(E)$, which we call "reciprocal energy" is additive at each check node in the same way that energy is additive at the variable nodes.

In density evolution on a protograph G, we recursively compute the quantity $E^i_{\overrightarrow{e}}$ and $E'^i_{\overleftarrow{e}}$ for each edge e in the protograph (representing a class of edges in the derived graph). The quantity $E^i_{\overrightarrow{e}}$ represents the "energy" corresponding to the message from variable to check along edge e at iteration i of decoding. The analogous quantity $E'^i_{\overleftarrow{e}}$ represents the "reciprocal energy" corresponding to the message from check to variable along edge e at iteration i.

$$E^i_{\overrightarrow{e}} = E_{ch(e)} + \sum_{e' \neq e} \Phi\left(E'^i_{\overleftarrow{e'}}\right)$$

where the sum is taken over edges $e' \neq e$ where e and e' are connected to the same check node. $E_{ch(e)}$ is the channel signal power given to the variable node of e. On the 0^{th} iteration, the energy $\Phi\left(E'^0_{\overleftarrow{e'}}\right) = 0$. If the protograph includes untransmitted variables and the variable adjacent to e is untransmitted, then $E_{ch(e)} = 0$, otherwise $E_{ch(e)} = E_{ch}$, the signal energy of the channel.

$$E'^{i+1}_{\overleftarrow{e}} = \sum_{e' \neq e} \Phi\left(E^i_{\overleftarrow{e'}}\right)$$

If, for particular values of E_{ch}, the energy $E^i_{\overrightarrow{e}} \to \infty$ as $i \to \infty$, then the RCA predicts that message-passing decoding for large codes based on G will be successful for the channel given by E_{ch}. The minimum value of E_{ch} such that $E^i_{\overrightarrow{e}} \to \infty$ is called the RCA threshold of G. It is reported in [10], and confirmed by our own observations that the RCA threshold is typically within 0.01 dB of the true density evolution threshold.

Chapter 3

Protograph Weight Enumerators

3.1 Introduction

LDPC codes are becoming a standard in today's error correcting systems. However, even as the number of codes investigated by researchers has swelled, it remains difficult to find codes achieving "near zero" error probability at rates close to Shannon capacity. Instead, codes which are designed to behave well close to the capacity limit typically exhibit an Error floor.

Error floors are generally attributed to small sets of variables such as low-weight codewords, low-weight stopping sets [11], pseudocodewords [12] of small pseudo-weight, or, in the case of quantized decoders, small trapping sets [13]. Often, these sets are discovered only after specific codes have been designed and simulated. However, it is desirable to be able to predict the existence and frequency of such sets for entire ensembles of codes.

Speaking more formally, we are interested in certain asymptotic weight enumerators of LDPC code ensembles. Gallager was able to compute asymptotic codeword weight enumerators for regular LDPC codes at least as early as 1963[14]. Litsyn and Shevelev[15] extended this result to include unstructured irregular ensembles. More recently, Di[16] has computed weight enumerators and stopping set enumerators also for unstructured irregular ensembles (in both average and typical case).

In this paper, we consider the problem of finding average enumerators for the class of protograph ensembles, which are related in a certain way to quasi-cyclic codes. Our methods, which are necessarily different from those used to compute enumerators for unstructured irregular ensembles, can be applied to both codeword and stopping set weight enumerators, based on their simple combinatorial characterizations.

In section 3.2, we define the quantity $A(\Theta, \mathbb{G})$ which is the number of vectors of fractional weight Θ having a certain relationship to the graph $\mathbb{G}$ (e.g. being a codeword or a stopping set). The expectation of this quantity with respect to an ensemble is $\overline{A_N(\Theta)}$. This quantity typically grows exponentially with N and the enumerator exponent $E(\Theta)$ is:

$$E(\Theta) = \lim_{N\to\infty} \frac{1}{N} \ln \overline{A_N(\Theta)} \tag{3.1}$$

We show that

$$E(\Theta) = \max_{\langle\theta\rangle=\Theta} E(\theta) \tag{3.2}$$

for a certain function $E(\theta)$ where θ is a vector fractional weight or partial weight. In section 3.4, we show how to compute the value of $E(\theta)$. In section 3.5, we show that $E(\theta)$ is in general not convex, and thus is difficult to optimize. Nonetheless, we apply steepest ascent to solve the maximization, and show that this method gives results that are reasonable.

In section 3.10, we outline some future research directions

3.2 Weight Enumerators Defined

Protograph ensembles are defined and characterized by a bipartite graph $P = (V, C, E)$, where $V = \{v\}$ is a set of variable nodes, $C = \{c\}$ is a set of check nodes, and $E = \{e\}$ is a set of edges each adjacent to one element $v(e) \in V$ and one element $c(e) \in C$. Formally, a protograph is equivalent to a Tanner graph, except that multiple edges are allowed.

A protograph P is semantically equivalent to an $r \times n$ protomatrix $H = (H_{c,v})$ where $H_{c,v}$ is the number of edges in E adjacent to c and v.

P can be lifted by a factor of N to generate a graph GÀn N-lift of P, which we denote $P^N = (V^N, C^N, E^N)$, is constructed from a set of permutations $\{\pi_e\}_{e\in E}$ of length N. We let $V^N = V \times [n]$, $C^N = C \times [n]$, and $E^N = E \times [n]$, where (e, i) is adjacent to (c, i) and $(v, \pi_e(i))$. In the lifted graph, we refer to v as the *type* of node (v, i), and to i as its *index*, and use a similar convention for check nodes (c, i) and edges (e, i).

We define the probability measure $P(\cdot)$ to be the uniform measure over N-lifts of P, that is where $\{\pi_e\}_{e\in E}$ are independent and uniform over all length N permutations. We sometimes refer to this probability measure as an ensemble of graphs.

The codewords $x \in \mathbb{G}$ are the assignments of $(0, 1)$ to each $v^N \in V^N$ such that each $(c, i) \in C^N$ is adjacent an even number of times to variable nodes assigned the value 1. Similarly, the stopping sets s of $\mathbb{G}$ are assignments such that each $(c, i) \in C^N$ is adjacent 0 times or at least 2 times to variable nodes assigned the value 1.

We are now ready to define a set Ω which generalizes the notions of codeword and stopping set. For each check $c \in C$ in the protograph define a set of allowed vectors $\Omega_c \subset (0, 1)^{\{e:\, c(e)=c\}}$.

For a particular word x and check node $(c, i) \in C^N$, define $\omega_{(c,i)}(x)$ to be the vector of variables connected to (c, i):

$$\omega_{(c,i)}(x) = (x_{v(e)})_{e:c(e)=(c,i)} \tag{3.3}$$

Then the set Ω is the set of x such that every vector $\omega_{(c,i)}(x)$ is in the corresponding allowed set Ω_c, formally:

$$\Omega = \{x\colon \omega_{(c,i)}(x) \in \Omega_c,\ c \in C,\ i \in [n]\} \tag{3.4}$$

is a set of words x with a certain combinatorial property.

By choosing an appropriate definition of Ω_c, it is possible make Ω the set of codewords or the set of stopping sets. If, for each $c \in C$, Ω_c is the set of vectors of even weight, then Ω is the set of codewords in $\mathbb{G}$. If Ω_c is the set of vectors of weight not equal to 1, then Ω is the set of stopping sets.

We are interested in the number of words in Ω of fractional weight $\Theta(x)$, defined to be the number of 1's in x divided by the word length $N \cdot n$. For a given graph $\mathbb{G}$ the number of such words is denoted $A(\Theta, \mathbb{G})$, and the expectation with respect to the ensemble of graphs of length N is denoted $\overline{A_N(\Theta)}$This expectation typically grows exponentially with N, and the exponent $E(\Theta)$ is defined by equation (3.1).

3.3 Approach

Our approach to computing $E(\Theta)$ is based on the method of types [17]. For a particular word x (not necessarily in Ω), denote its type (or partial weight) by $t(x) = (\theta_v)_{v \in V}$, where θ_v denotes the fraction of times that x assigns 1 to variables of type v. The following lemma says that the probability that $x \in \Omega$ depends only the type θ.

Lemma 3.3.1 *if x and y are assignments of $(0,1)$ to each $v \in V^N$ such that $t(x) = t(y)$ then $P(x \in \Omega) = P(y \in \Omega)$*

Proof. $\theta_v(x) = \theta_v(y)$ implies that there exists a vector of permutations $(\pi_v)_{v \in V}$ such that for each v, $x_v = \pi_v(y_v)$, where x_v is the value x assigns to $\{v, i\}$.

The permutations $(\pi_v)_{v \in V}$ define a bijection on elements of the ensemble defined by $f((\pi_e)) = (\pi_e \cdot \pi_{(v(e))})$ such that $y \in \Omega_{(f((\pi_e)))}$ if and only if $x \in \Omega_{(\pi_e)}$. Since all lifts in the ensemble are equiprobable, the conclusion holds. ∎

Thus the expected number of words of type θ, which we denote $\overline{A_N(\theta)}$ is just the number of words of type θ times the probability that any word of that type is in Ω:

$$\overline{A_N(\theta)} = |\{x : t(x) = \theta\}| \cdot P(x \in \Omega | t(x) = \theta) \tag{3.5}$$

It is straightforward to see that the number of words of type θ can be approximated as:

$$|\{x : t(x) = \theta\}| = e^{N \sum_v H(\theta_v)} \tag{3.6}$$

Define the indicator function that x satisfies all of the constraints

Definition 3.3.2

$$f_{\Omega_c}(x, \mathbb{G}) = \begin{cases} 1, \textit{ if } \omega_{(c,i)}(x) \in \Omega_c \forall i \in [n] \\ 0, \textit{ otherwise} \end{cases} \tag{3.7}$$

The following lemma says that for any x, the probability of satisfying each type of constraint Ω_c is independent over $c \in C$.

Lemma 3.3.3

$$P(x \in \Omega) = \prod_{c \in C} P(f_{\Omega_c}(x, \mathbb{G}) = 1) \tag{3.8}$$

Proof. for a particular x, $f_{\Omega_c}(x, \mathbb{G})$ is a function only of the set of permutations $\{\pi_e\}_{e:c(e)=c}$. The permutations $\{\pi_e\}$ are mutually independent, and thus independent with respect to the partitioning $\left\{\{\pi_e\}_{e:c(e)=c}\right\}_{c \in C}$. The result follows since functions of independent variables are independent. ■

We define the asymptotic exponent of the probability that all $\omega_{(c,i)} \in \Omega_c$ for all checks of type c:

$$\Phi_c(\theta_c) = \lim_{N \to \infty} \ln(P(f_{\Omega_c}(x, \mathbb{G}) = 1))/N \tag{3.9}$$

For a particular word x, this probability depends only on the vector of weights θ_c associated with the variables adjacent to check c in the protograph:

$$\theta_c = (\theta_{v(e)})_{e:v(e)} \tag{3.10}$$

From a computational point of view, it is unfortunate that independence does not factor further. Although $f_{\Omega_c}(x, \mathbb{G})$is independent from type to type, $\omega_{(c,i)}(x)$ are generally dependent among values of i. Nonetheless, we can apply large deviation theory and Sanov's theorem to obtain the following theorem, which shows in principle how to compute the asymptotic probability exponent $\Phi_c(\theta_c)$.

Theorem 3.3.4

$$\Phi_c(\theta_c) = \max_{p \in \mathbb{P}^{\Omega R}} H(p) - \sum_{e:c(e)=c} H(\theta_{v(e)}) \tag{3.11}$$

where $\mathbb{P}^{\Omega R}$ is the set of distributions P over satisfying the marginal constrains:

$$\sum_{\omega \in [0,1]^{\deg(c)}} \omega P(\omega) = \theta_c \tag{3.12}$$

and having support only on Ω_c, thus satisfying

$$P(\omega) = 0, \; \forall \; \omega \notin \Omega_c \tag{3.13}$$

Proof. From the definition of the protograph ensemble, the check (c, i) is adjacent to the vector of variables $((v(e), \pi_e(i)))_{e \in E: c(e)=c}$. Consider the matrix whose columns are the set of such vectors, namely

$$X^c = \left(X^c_{e,i}\right), \; X^c_{e,i} = x(v(e), \pi_e(i)) \tag{3.14}$$

Because (π_e) is a set of independent uniform random permutations X^N_c is a random matrix with uniform probability over all matrices with row sums $(N\theta_c)$. Denote this set $R^N_{\theta_c}$. Furthermore, $x \in \Omega^N_c$ if and only if each column of $X^c_i \in \Omega_c$.

Define a uniform probability measure Q over all $m \times N$ $[0, 1]$ matrices. Now

$$P(X^N_c \in \Omega^N_c) = Q(\Omega^N_c | R^N_{\theta_c}) \tag{3.15}$$

Applying the definition of conditional probability,

$$Q(\Omega^N_c | R^N_{\theta_c}) = \frac{Q(\Omega^N_c \bigcap R^N_{\theta_c})}{Q(R^N_{\theta_c})} \tag{3.16}$$

Since the event $\Omega^N_c \bigcap R^N_{\theta_c}$ depends only on the empirical distribution on columns of X^N_c, and Q is column-wise independent [explanation?], we can apply Sanov's theorem [17] directly to obtain

$$\lim_{N \to \infty} \frac{1}{N} \ln Q(\Omega^N_c \bigcap R^N_{\theta_c}) = \min_{P \in \mathbb{P}^{\Omega R}} D(P||U_c) \tag{3.17}$$

where U_c is the uniform distribution over vectors of length $\deg(c)$ and $\mathbb{P}^{\Omega R}$ is the set of distributions satisfying both (3.13) and (3.12).

$$\min_{P \in \mathbb{P}^{\Omega R}} D(P||U_c) = H(U_c) - \max_{P \in \mathbb{P}^{\Omega R}} H(P) \tag{3.18}$$

The denominator of the right hand side of (3.16) can be asymptotically estimated in a similar way

$$\lim_{N\to\infty} \frac{1}{N} \ln Q(R_{\theta_c}^N) = \min_{P\in\mathbb{P}^R} D(P||U_c) \tag{3.19}$$

where $\mathbb{P}^R$ is the set of distributions satisfying (3.12)

It can easily be seen that the P which minimizes $D(P||U_c)$ is the component-wise independent distribution with marginals equal to θ_c, and thus

$$\min_{P\in\mathbb{P}^R} D(P||U_c) = H(U_c) - H(\theta_c) \tag{3.20}$$

where $H(\theta_c) = \sum_{e:c(e)=c} H(\theta_{v(e)})$.

Applying the asymptotic expressions in equations (3.18) and (3.20), we obtain

$$\lim_{N\to\infty} \frac{1}{N} \ln Q(\Omega_c^N|R_{\theta_c}^N) = \max_{P\in\mathbb{P}^{\Omega R}} H(P) - H(\theta_c) \tag{3.21}$$

■

We now apply theorem 3.3.4, to computing $E(\theta)$. Taking the log of equation (3.5), and substituting equation (3.6) and (3.9), we have:

$$E(\theta) = \sum_{v\in V} H(\theta_v) + \sum_{c\in C} \Phi_c(\theta_c) \tag{3.22}$$

In the following section, we show how to numerically compute the value of $\Phi_c(\theta_c)$ and thus how to compute $E(\theta)$.

3.4 Numerical Methods for computing $E(\theta)$

In the previous section, we have seen that computing each function $\Phi_c(\theta_c)$ requires solving a constrained entropy maximization problem.

In this section, we describe the computational mathematics used to calculate $E(\Theta)$, for a given protograph described by an $r \times n$ matrix $\mathbf{H}$.

Let m_c be the degree of c. We have seen that Ω_c is a set of m_c-vectors. We seek $\Phi_c(\theta_c) = \max H(p) - \sum_e H(\theta_v)$ where $p(\omega)$ is the set of all probability mass functions satisfying equation (3.12)

Applying Euler-Lagrange theory, we will see how this constrained optimization problem can be transformed into a non-linear system of equations. The Lagrangian corresponding to our constrained optimization problem can be written

$$\mathcal{L}(p) = -\sum_{\omega\in\Omega} p(\omega)(\log(p(\omega)) - \mathbf{s}\cdot\omega) \tag{3.23}$$

The constrained optimum must satisfy $\frac{\partial \mathcal{L}}{\partial p} = 0$, and this condition implies a Boltzmann distribution on ω given by

$$p^*(\mathbf{s}, \omega) = \frac{1}{z(\mathbf{s})} e^{-\mathbf{s}\cdot\omega} \tag{3.24}$$

The normalizing constant $z(\mathbf{s})$ takes the value that ensures p is a probability distribution, appropriately summing to 1.

$$z(\mathbf{s}) = \sum_{\omega \in \Omega} e^{-\mathbf{s}\cdot\omega}, \tag{3.25}$$

The Helmholtz free energy can be written in terms of $Z(\mathbf{s})$:

$$F(\mathbf{s}) = -\log(Z(\mathbf{s})) \tag{3.26}$$

and has the property that its gradient with respect to $\mathbf{s}$ is equal to the l.h.s of equation (3.12).

$$\begin{aligned} \nabla F(\mathbf{s}) &= \frac{1}{Z(\mathbf{s})} \sum_{\omega \in \Omega} \omega e^{-\mathbf{s}\cdot\omega} && (3.27) \\ &= \sum_{\omega \in \Omega} \omega p(\omega). && (3.28) \end{aligned}$$

Thus, if we find $\mathbf{s}^*$ which solves

$$\nabla F(\mathbf{s}^*) = \theta_c \tag{3.29}$$

then the probability density that leads to the maximum entropy is given by $p^*(\mathbf{s}^*, \omega)$ and $\Phi_c(\theta_c)$ can be expressed

$$\Phi_c(\theta_c) = -F(\mathbf{s}^*) + \mathbf{s}^* \cdot \nabla F(\mathbf{s}^*) \tag{3.30}$$

The domain of θ_c, usually denoted K, is the convex hull of all $\omega \in \Omega_c$. Outside of this domain, no distribution p can satisfy (3.12). As a side remark, we note that if each Ω_c is the set of even weighted vectors, then the feasible region is formally equivalent to the pseudocodeword fundamental polytope.

The domain of $\mathbf{s}$ is the entire space R^n of real vectors. It is shown in a concurrently submitted paper by Aji et. al[18] that there is a one-to-one correspondence between these two domains. That result follows from considering the Helmholtz free energy and its Legendre conjugate.

Once we have this non-linear system of equations (3.29), we can numerically solve for $\mathbf{s}^*$ using either Broyden's or Newton's Method. We found Broyden's Method [19] to be far faster, and thus use it whenever possible. However, since Broyden's Method uses only an approximation to the Jacobian, it is not always able to find a solution. In practice, this typically happens near the boundary of the

feasible set(where calculations of gradients and Jacobians become more difficult by any method). The values of θ_c on the boundary of K correspond to values of s with infinite norm, and care must be taken to avoid numerical problems in this region.

In the following section, we will show how to optimize the function $E(\theta)$ over θ to obtain $E(\Theta)$.

3.5 Maximization of $E(\theta)$

Since there are only polynomially many types, each having an exponential number of elements, it is a standard result that the sum is dominated by a single type, as expressed in equation (3.2).

In general, the function $E(\theta)$ is not convex. For protographs with no check nodes of high degree, it may be possible to essentially search the whole space $\{\theta : \langle\theta\rangle = \Theta$ for the global maximum of (3.2), but this is impractical for protographs with any large check nodes. A second approach is to use a gradient following method such as steepest descent. Unfortunately, this is not guaranteed to converge to the global minimum. A third approach is to use steepest descent starting from a number of different starting locations. In practice, this approach is sufficient to compute curves that appear continuous for protographs that have been investigated.

Still, it is an artifact of certain protographs that there are critical values of Θ at which the global minimum of $E(\theta)$ jumps from one place to another, which is reflected in a discontinuity in the first derivative of $E(\Theta)$.

Figure 3.5 shows our evaluation of the weight enumerator for a rate 1/3 protograph defined by the matrix in equation (3.31). The zoomed section, shown in figure 3.5 shows a discontinuity at approximately $\Theta = 0.13$ in which the global maximum of (3.2) jumps from one value of θ to another. The dashed lines indicate other local maxima.

$$H = \begin{pmatrix} 3 & 0 & 3 \\ 0 & 3 & 4 \end{pmatrix} \tag{3.31}$$

For a given matrix, $\mathbf{H}$, and a value of Θ perhaps only some vectors θ have a solution to the entropy maximization problems for each row of $\mathbf{H}$, as for some problems there are no probability mass functions which satisfy the constraints. A simple algorithm can be used to determine whether a vector θ is feasible. In the steepest ascent code, if we find ourself stepping outside the feasible set, we just take a smaller step size until either we remain in the feasible set or the step size becomes effectively zero.

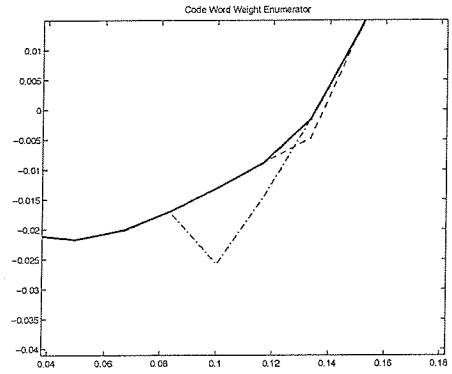


Figure 3.1: This weight enumerator has an elbow

3.6 Domain of $E(\theta)$

It is important both from an analytical and a computational point of view to know the domain in which $E(\theta)$ is well-defined. In this section, we give a general characterization of the domain. We further show that for both stopping set enumerators and weight enumerators, determination of whether a specific vector θ is in the domain can be done by efficient algorithms.

The domain of $E(\theta)$ is in general a subset of $[0,1]^V$, since by definition the fraction of 1's of any type v is by definition in $[0,1]$. Referring to equation(3.22), the term $H(\theta_v)$ is well defined in this set and thus does not restrict the domain. However, the term $\Phi_c(\theta_c)$ is defined only where there is at least one distribution satisfying equation (3.12), which is exactly when θ_c is in the convex hull of Ω_c. If all such constraints are satisfied, then $E(\theta)$ is well-defined. Thus, in the general case, the domain of $E(\theta)$ can be expressed:

$$Domain(E(\theta)) = \{\theta : \theta_c \in K(\Omega_c)\} \tag{3.32}$$

Computationally this is feasible to test as long as the degree of the check nodes is small. However, a direct application of this criterion implies at least an enumeration of all of the elements of Ω_c for each c. For codewords and stopping sets however, the characterization is even simpler.

3.6.1 Continuity of $E(\Theta)$

It is shown by Aji et al. [18] that $\Phi_c(\theta_c)$ is a smooth function of

3.7 Behavior of $E(\Theta)$ Near Zero

An important feature of average weight enumerators is their behavior near $\Theta = 0$. For some protographs, there is a region $(0, \Theta^*)$ for which the codeword weight enumerator $E(\Theta) < 0$. For

such ensembles, we will see that the majority of the codes have minimum distance growing like $N \cdot \Theta^*$.

3.7.1 Types of Zero-crossings

Informally speaking, there are three essential ways in which $E(\Theta)$ can behave near zero. We have seen that for stopping set enumerators and codeword enumerators $E(0) = 0$ for every protograph. Above zero, $E(\Theta)$ may become strictly negative. Another way is for $E(\Theta)$ to be exactly zero for some finite stretch, and the third is to become immediately positive. In this section, we will formalize these notions, and prove constructively that each case is possible.

For a given enumerator $E(\Theta)$, define:

$$\begin{aligned} \Theta^1 &= \inf_{\Theta>0} E(\Theta) \geq 0 \\ \Theta^2 &= \inf_{\Theta>0} E(\Theta) > 0 \end{aligned}$$

Clearly $\Theta^2 > \Theta^1$ since $\{\Theta : E(\Theta) > 0\} \subset \{\Theta : E(\Theta) \geq 0\}$. Having made this observation, we can divide

If $\Theta^1 = \Theta^2 = 0$, then

From a point of view of performance of the code under ML decoding, the most desirable way is the first, in which $E(\Theta)$ becomes strictly negative. In this case it is possible to argue that with high probability the minimum distance of the code grows linearly with N. In the second case, it is impossible to directly argue whether the minimum distance does or does not grow linearly with N. Spielman ***

3.8 Algorithmic Determination of Class of P

It is desirable to be able to quickly determine the class of a protograph. It is possible to determine the class of P

3.9 Protograph Ensembles as Multi-Edge-Type Ensembles

Multi-edge-type ensembles [13]

3.10 Discussion

A primary motivation for computing enumerators has been to use them to design ensembles of codes with low error floors. The general idea is to use a combination of enumerator properties, such as the asymptotic expected minimum weight, and density evolution threshold. Preliminary experiments in which such codes have been designed and simulated have suggested that this approach can be effective.

3.10.1 Quasi-Cyclic Ensembles

Quasi-cyclic codes [4] have become a popular choice for many applications for many reasons including a relatively efficient encoding algorithm. It would therefore also be desirable to know something about their average weight enumerators.

Ensembles of quasi-cyclic codes can defined for protograph P by altering the definition so that the permutations $\{\pi_e\}_{e\in E}$ are chosen among only cyclic permutations. Unfortunately, the approach of this paper cannot easily be extended to compute enumerators for this kind of ensemble. One reason for this is that lemma (3.3.3) does not hold for this ensemble.

The conjecture that average weight enumerators of quasi-cyclic ensembles are the same as those of full ensembles is false. One graph in the quasi-cyclic ensemble is that in which each permutation being the identity permutation. This graph is disconnected into N pieces, each identical to the protograph. As long as each one of these pieces admits at least one non-zero assignment, the whole code has exponentially many codewords of any weight $N \cdot \Theta$, where $0 < \Theta < 1$. Since the quasi-cyclic ensemble has only a polynomial number $N^{|E|}$ of graphs in the ensemble, this implies that the average weight enumerator $E(\Theta)$ is positive. This contrasts with our results for specific protographs whose full-ensemble codeword weight enumerators we have shown to be negative for certain values of Θ.

Chapter 4

Protograph Optimization

In this chapter, we look at some basic ways in which protograph ensembles can be optimized. Since a protograph is characterized by its

A method is presented for constructing LDPC codes with excellent performance, simple hardware implementation, low encoder complexity, and which can be concisely documented. The simple code structure is achieved by using a base graph, expanded with circulants. The base graph is chosen by computer search using simulated annealing, driven by density evolution's decoding threshold as determined by the reciprocal channel approximation. To build a full parity check matrix, each edge of the base graph is replaced by a circulant, chosen to maximize loop length by using a Viterbi-like algorithm. One hardware decoder implementation performs belief propagation sequentially on copies of the base graph, and in parallel on the edges within the base graph.

4.1 Introduction

Since the rediscovery of low-density parity-check (LDPC) codes in the 1990's, many researchers have discovered ways to improve the decoding performance of Gallager's codes. In a landmark 2001 paper [9] Richardson and Urbanke defined a class of ensembles of irregular LDPC codes, and listed an extensive collection of optimized degree distributions.

More recent research has been directed toward designing good LDPC codes with additional structural constraints imposed. One such construction that appears to have been developed concurrently by several different researchers is to generate a Tanner graph based on many copies of a smaller graph. This is described in [20] in terms of a protograph, and is closely related to the base graph of Lin [21], the seed graph of Tanner [4], and the more general Multi-Edge-Type construction of Richardson [8] [13].

This construction offers several advantages. Significantly, it is possible to lower the theoretical iterative decoding threshold compared to that of randomly connected codes, subject to the same constraint on maximum variable node degree. The imposed structure can also facilitate analysis and

design of the code, as well as offer a convenient way to document and store the code. Finally, this structure can be exploited in the creation of simple and efficient hardware decoder designs.

The construction of [20] begins with a small (n, r) base graph G called a *protograph*, containing n variable nodes and r check nodes, no more than a few dozen in total. The protograph is then expanded by making T copies, at which point each edge becomes a bundle of T parallel edges. Then, the endpoints of the bundle corresponding to edge e is rearranged by a permutation π_e. We call the the new T times larger graph the derived graph.

The derived graph inherits many important properties from the protograph including the degree distribution. If T is large enough, and each permutation π_e is chosen at random, then the performance of the resulting LDPC code can be determined by density evolution on G, as discussed in section 4.2. On the other hand, if T is not very large or the π_e's are highly constrained, then important graph characteristics like loop lengths and stopping set sizes can depend more strongly on the choice of π_e's. These issues can sharply affect code performance.

One approach to generating the π_e's is to use an algorithm such as Progressive Edge Growth (PEG)[22] which places one edge at a time, attempting at each step to avoid creating short loops. However, there is in general no efficient way to document and store the output of such an algorithm, which is necessary since both the encoder and decoder must be aware of the specific values of π_e's. Random selection of π_e's suffers from the same problem.

A good alternative which has been pioneered by Tanner [4] and Lin [21] is to restrict π_e's to a much smaller class of permutations such as *circulant* permutations I_{ϕ_e}, the $T \times T$ identity matrix right circularly shifted by ϕ_e. We contribute in this area in Section 4.3 by adapting the PEG algorithm to the specific task of circulant selection.

4.2 Optimization via Simulated Annealing

A technique that seems to be popular is to "hand pick" a protograph (or base graph or base matrix) to approximately match published optimal irregular degree distributions, or by simple trial and error. This technique can be used to approximately match the performance achievable with homogeneous (single edge-type) irregular LDPC codes. However, using an exploratory algorithm, we can design protograph based LDPC codes which achieve performance superior to what is achievable by single edge-type codes under certain constraints.

We make use of the RCA threshold to search for good protographs that perform well under BP on the AWGN channel. This search uses a randomized iterative algorithm called Simulated Annealing (SA), and is directed by our estimate of how the DE threshold changes as an underlying protograph is perturbed.

In our application, we search over protographs G with fixed numbers n of transmitted nodes, u

of untransmitted nodes and r of check nodes. Since the code rate is determined by these parameters, it is constant throughout the optimization space, and we can design good codes for any desired rate by our choice of n, u and r.

We define an energy $E(G)$ associated with graph G, to be minimized in SA. A natural quantity to choose is the SNR corresponding to the RCA threshold. However, we define it to be the channel SNR at which the average bit error probability is less than ε after I decoder iterations.

At each stage j of the SA algorithm, we select one of three types of perturbations on G_j: removing an edge, adding an edge, and swapping the endpoints of two edges in G_j to generate a new graph G'_j. If the energy $E(G'_j) < E(G_j)$ then $G_{j+1} = G'_j$. Otherwise, $G_{j+1} = G'_j$ with probability $\exp\left(\frac{E(s_j)-E(s'_j)}{t_j}\right)$ and G_j with the remaining probability, where t_j is the temperature at time j. The algorithm returns the solution G_J at the final J^{th} iteration.

If ε is chosen small enough and I is chosen large enough, then this corresponds well with RCA threshold. On the other hand, choosing smaller values of I can improve the performance of the resultant codes under small numbers of iterations. More surprisingly, preliminary experiments indicate that optimizing graphs under smaller values of I may effectively lower the error floor for small block length codes ($n < 10000$).

4.2.1 Optimization Results

An interesting comparison that can be made is between the thresholds of optimized protographs with a certain maximum degree and optimized irregular codes. A protograph has been found with Density Evolution threshold within 0.08 dB of channel capacity with maximum degree 8. To achieve such performance with irregular codes published in [23], a maximum degree of 50 is required.

4.3 Permutation Selection

Consider a protograph with check nodes $c \in C$ and variable nodes $v \in V$, connected by edges $e \in E$. Given a protograph and an expansion factor T (the dimension of each permutation matrix), the algorithm's task is to choose a circulant phase ϕ_e, $0 \leq \phi_e \leq T-1$, for each edge e in the protograph, such that small loops are avoided in the derived graph. Our protograph-and-circulant graph construction method implies two obvious relationships between loops in the protograph and corresponding loops in the derived graph.

Theorem 4.3.1

Theorem 4.3.2

While Theorem 4.3.1 holds for any set of edge permutations $\{\pi_e\}$, not necessarily circulants, Theorem 4.3.2 gives a simple criterion for selecting sets of circulant permutations to avoid small

loops in the derived graph. Note that the signs of the terms in this summation alternate for edges traversed from variable node to check node, and for edges traversed in the reverse direction.

Our algorithm begins with a blank slate, and builds one edge of the protograph at a time. If this edge e does not complete a loop in the protograph, it is assigned a circulant phase ϕ_e at random. If it does, then the phase ϕ_e that maximizes the minimum loop length in the derived graph is selected. Computationally, this involves trying each of the T possible phases in turn, and finding the smallest loop that includes this edge, and satisfies the condition in Theorem 4.3.2.

Finding the smallest loop in a graph containing a particular edge is readily done with a Viterbi-like algorithm. When the edges are labeled with circulant phases, this problem becomes modestly more difficult, but can still be done in a Viterbi-like way. We convert the protograph into a directed graph by duplicating each edge, and labeling the edges leaving variable nodes with ϕ, and those entering variable nodes with $-\phi$. This graph does not eliminate the possibility of retracing an edge, but this can be resolved by transforming this State Transition Graph (STG) to a Finite State Machine [24], and deleting edges corresponding to direction reversals. Finally, we convert this back to an STG by labeling the outgoing edges with the phases, rather than the nodes they come from. This transformation is illustrated for a small protograph in Figure 4.3. We run a Viterbi-like algorithm on this graph.

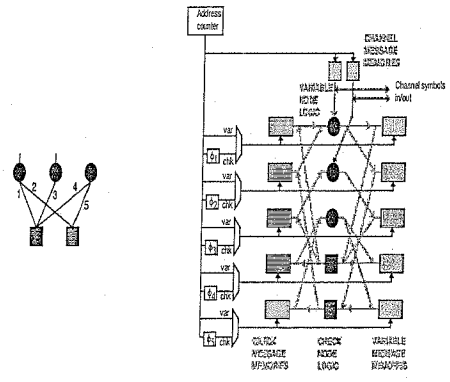


Figure 4.1: change this figure

Each edge in the graph (or trellis) is labeled with the monomial x^{ϕ}, and the Viterbi-like algorithm is initialized by putting 1 at the node corresponding to the circulant of interest, and 0's at all other nodes. The Viterbi algorithm proceeds with an unusual commutative semiring [7]: polynomials are multiplied and added in the usual way, but coefficient addition is replaced by the logical OR operation, and polynomials are reduced according to $x^T = x^0 = 1$. In this way, the non-zero terms in the polynomial assigned to a node indicate the accumulated phases of each path to that node. We halt the Viterbi iteration when the node corresponding to the circulant of interest is assigned a polynomial with nonzero constant term, thus closing a loop satisfying the condition in Theorem 4.3.2.

A corollary of Theorem 4.3.2 can be used to identify some types of protographs whose expansions by circulants are necessarily doomed to contain small loops.

Corollary 4.3.3 *If the protograph contains three parallel edges* $\{e_1, e_2, e_3\}$*, and the derived graph is constructed from circulants, then the shortest loop in the derived graph cannot be made longer than 6.*

This result is proved by traversing the edges in the order $e_1, e_2, e_3, e_1, e_2, e_3$: each edge is traversed once in each direction, so the circulant phases always sum to zero. A generalization of this corollary applies whenever there are three disjoint paths between any two nodes in the protograph.

4.4 Hardware Implementation of the Decoder

The protograph-and-circulant structure of the derived graph is particularly suitable to belief propagation (BP) decoder implementation in hardware. A brief summary is given here; a more complete development is given in [25]. Conceptually, a BP decoder simultaneously performs a simple computation at each variable node, generating one message (a number quantized to a few bits) for each outgoing edge, from the messages arriving on the incoming edges. Then another simple computation is performed simultaneously at each check node, returning updated messages to the variable nodes. The process is repeated until some stopping condition is satisfied.

For codes of interest and practical ASIC or FPGA sizes, one typically cannot implement a fully parallel decoder, with independent logic for every variable node and every check node. However, one can typically implement enough logic for several variable nodes and several check nodes. For arbitrary derived graphs, message interconnection and scheduling become intractable problems when more than a few nodes are implemented.

For derived graphs built with the protograph-and-circulant construction, it is natural to implement a decoder that can simultaneously process all the variable nodes, or all the check nodes, in the protograph, and which processes copies of the protograph serially. To do this, two memories of size T messages (times some small number of bits per message) are implemented for each edge of the protograph, one for variable-to-check messages, and one for check-to-variable messages. When performing variable node computations, in each clock cycle, one message is read from each variable-to-check memory, and one message is written to each check-to-variable memory, and the memories are addressed sequentially. When performing check node computations, the opposite memories are read and written. To capture the circulant permutations, the memory addresses for each bank are offset (modulo T) by the appropriate circulant phase. Figure 4.4 shows a small protograph and the block diagram of a corresponding hardware decoder.

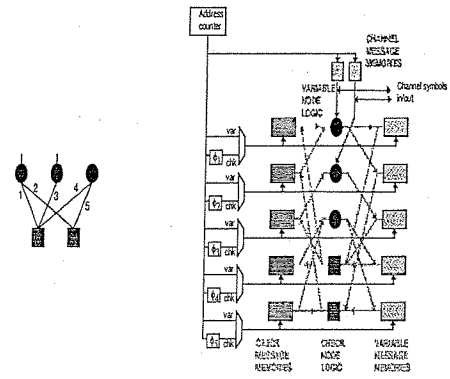


Figure 4.2: A protograph, and a corresponding hardware decoder block diagram

4.5 Performance Comparisons

Using the protograph optimization tools described in Section 4.2, we have consistently been able to find protograph designs that yield (asymptotic) iterative decoding thresholds within approximately 0.1 dB of the capacity limit. This indicates that the protograph design constraint does not limit achievable code performance to any appreciable degree; Figure 4.5(a) shows a good code. As shown in the example of Figure 5 of [20], however, a high error floor can result when a near-optimum protograph is expanded to finite size. In that example, the protograph was expanded using ordinary PEG, but a similar error floor results when circulant-constrained PEG is used. It is an open question whether the close approach to the capacity limit necessarily entails a sacrifice in error floor performance, or whether both the asymptotic threshold and the error floor can be lowered by more careful joint design of the protograph and edge permutations.

In this paper we confine our attention to a simpler question: Given the protograph construction method, does our use of circulant permutations instead of unconstrained permutations cause any significant degradations in achievable performance? As an illustration, we constructed a series of small rate-1/2 codes, and the simulated decoder performance for a few of these are shown in Figure 4.5(b). This figure compares four code constructions of size $(n = 612, k = 306)$. Two codes are regular (3,6) constructions, with an asymptotic threshold of 1.10 dB. Two codes are derived from an irregular protograph with three variable nodes of degree 2, two of degree 3, and one of degree 9, and three check nodes of degree 7. These codes have a better asymptotic threshold of 0.65 dB. All three protographs were expanded using ordinary PEG and circulant PEG.

We see from Figure 4.5 that the codes are differentiated from each other roughly by the differences in their asymptotic decoding thresholds down to a codeword error rate (WER) of roughly 10^{-3}. Below that point the error floor starts to rear its ugly head, less notably for the codes with the poorer asymptotic threshold. However, we see that, for both of the protographs, the code built from

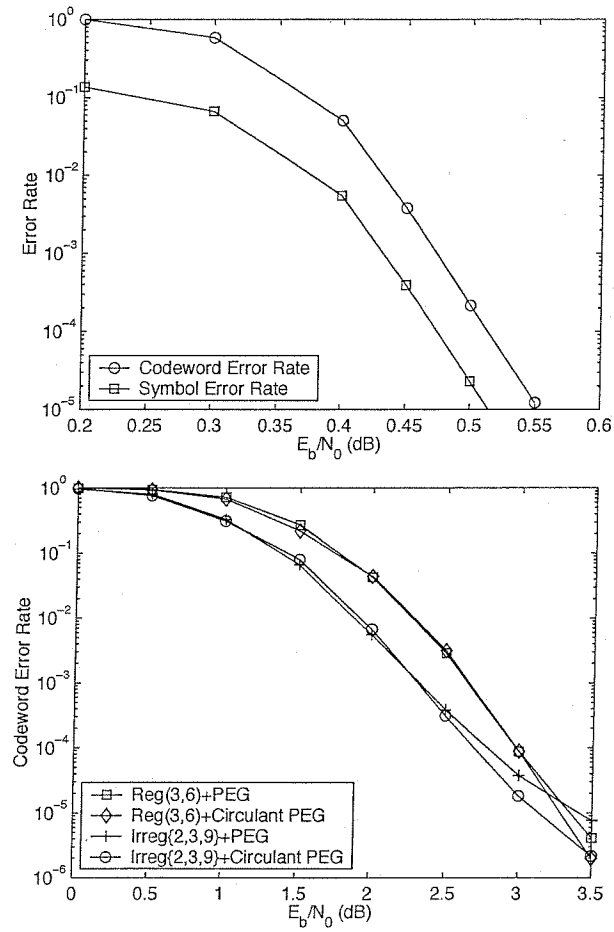


Figure 4.3: a) Performance of a $(n = 64800, k = 32400)$ Protograph-and-Circulant code; b) Performance of a $(n = 64800, k = 32400)$ Protograph-and-Circulant code

its protograph by circulant-constrained PEG achieved the lower WER at the highest tested value of E_b/N_0. This gives a preliminary indication that our method of selecting circulant permutations does not inherently cause higher error floors.

4.6 Finding Good Protograph Codes

We have devised an algorithm that searches for protograph codes that perform well under BP on the AWGN channel. This search uses a randomized iterative algorithm called Simulated Annealing (SA), and is directed by our estimate of how the the DE threshold changes as an underlying protograph is perturbed.

SA approximately minimizes over a search space S an energy function $E : S \rightarrow \mathbb{R}$. It requires a random perturbation function $p : S \rightarrow S$, and a decreasing temperature profile $Temp(j)$. It begins with an arbitrary solution $s_0 \in S$, and at each iteration j, applies the random perturbation p to generate a new solution $s'_j = p(s_j)$. If $E(s'_j) < E(s_j)$ then $s_{j+1} = s'_j$. If $E(s'_j) \geq E(s_j)$, then $s_{j+1} = s'_j$ with probability $e^{-\frac{(E(s')-E(s))}{Temp(j)}}$ and s_j otherwise. The algorithm returns the solution s_j at the final iteration.

In our application, we search over protographs G with a fixed n, u and r. Since the code rate R is determined by these parameters, it is constant throughout the optimization space. Thus we can design good codes for any desired rate by a judicious choice of n, u and r.

We define 3 types of perturbations on a protograph G in the search space: removing an edge in G, adding an edge to G, and swapping the endpoints of two edges in G. The random perturbation p chooses one of these types at random, and chooses uniformly randomly among all possibilities for that type.

We would ideally like to use the DE threshold as our Energy Function E. However, it is quite impractical to run full DE at each iteration. Instead, we use the reciprocal-channel approximation introduced in [10], a single-parameter approximation to DE for the AWGN channel. Further, instead of running approximate DE several times per SA iteration (j) to determine the threshold (under approximate DE) accurately, we run approximate DE just once at an operating point above the threshold for the current solution G_j. We let our energy function E be the number of decoding iterations (i) required to drive the output error probability below some small constant ε. Empirically, we have observed that this quantity varies approximately monotonically with the threshold.

4.7 Optimization Results

Figure 5 shows an example of a simple rate $\frac{1}{2}$ protograph found by the SA search method. This optimized protograph has n =8transmitted variables, $u = 1$ untransmitted variable, $r = 5$ checks,

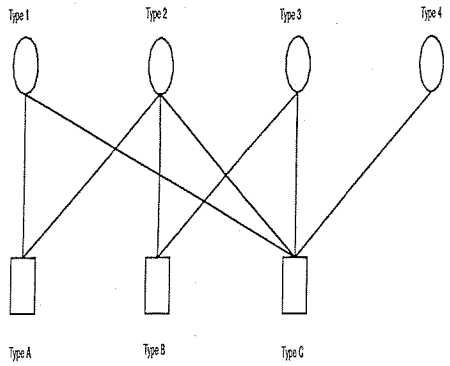


Figure 4.4: change this picture

and a total of $|E| = 29$ edges, including 5 pairs of parallel edges. The untransmitted variable node has degree 9, the transmitted variable nodes have degrees $2, 2, 2, 2, 3, 3, 3, 3$, and the check nodes have degrees $4, 5, 6, 6, 8$. The approximate DE threshold on the AWGN channel for long codes constructed from this protograph is $Eb/N0 = 0.283dB$, which is about $0.10dB$ better than the exact DE threshold of $0.4090dB$ found by Richardson and Urbanke [23] for optimized irregular LDPC codes with maximum variable node degree 9. The irregular LDPC codes in [23] are connected according to a random ensemble constrained only by the distribution of node degrees, not by the additional structure built into the protograph.

We constructed a large $(n = 8192, k = 4096)$ protograph code by interconnecting $T = 1024$ copies of the protograph in Fig. 4.7. The interconnections are determined by 29 separate permutations of $1, ..., 1024$, selected using a variation of the progressive edge growth (PEG) algorithm [21] to avoid short loops in the derived graph. The decoding performance of this code is shown in Fig. 6 and compared to that of a multi-edge-type LDPC code that we designed using the principles discussed in [8]. The performance of the protograph code is perhaps 0.1 dB better than that of the multi-edge-type code in the threshold region, but it suffers from the appearance of a true error floor due to low-weight codewords above a word-error rate of 10^{-4}. Our current criterion for selecting a good protograph minimizes the protograph code's (asymptotic) threshold but does not penalize code structures that might yield high error floors. Future work will investigate how to eliminate the error floor or to trade it off versus a small sacrifice in the optimized threshold.

4.8 Conclusion

The teaming of an underlying protograph structure with circulant edge permutations lends microscopic regularity to the large derived graph. It simplifies the analysis of iterative decoding performance and yields hierarchically parallelizable decoder designs suitable for high-speed FPGA im-

plementations that are scalable according to the available silicon area. Just as importantly, there does not appear to be any fundamental limitation on code performance imposed by constraining the design to have the protograph-and-circulant structure. More research is needed to jointly select a protograph and a corresponding set of circulant permutations to simultaneously lower the code's asymptotic iterative decoding threshold and its error floor.

Chapter 5

A Scalable Architecture of a Structured LDPC Decoder

We present a scalable decoding architecture for a certain class of structured LDPC codes. The codes are designed using a small (n, r) protograph that is replicated Z times to produce a decoding graph for a $(Z \times n, Z \times r)$ code. Using this architecture, we have implementated a decoder for a $(4096, 2048)$ LDPC code on a Xilinx Virtex-II 2000 FPGA, and achieved decoding speeds of 31 Mbps with 10 fixed iterations. The implemented message-passing alogrithm uses an optimized 3-bit non-uniform quantizer that operates with $0.2dB$ implementation loss relative to a floating point decoder.

5.1 Introduction

Low-Density-Parity-Check (LDPC) codes[14] have recently received a lot of attention because of their excellent error-correcting capability. LDPC codes have been shown to be able to perform close to the Shannon limit [10]. They also can achieve very high throughput because of the parallel nature of their decoding algorithms. In the past decade or so, much of the research on LDPC codes has focused on the analysis and improvement of codes under decoding algorithms with floating point precision. However, to make LDPC codes practical in the real world, the design of an efficient hardware architecture is crucial.

5.2 Structured LDPC codes

Given unlimited hardware resources, a well-understood strategy is to allocate one processing element to each check and variable node in the Tanner graph[26] of an LDPC code. However, for the sake of error-correcting capability, it may be desirable to use a code with many more nodes than can be instantiated with limited hardware resources. To this end, we have developed an architecture for

decoding structured LDPC codes in which computations are scheduled in space and time.

5.3 Protograph Construction

By "structured", it is meant that the code is constructed via a specific construction called a "protograph"[20]. The protograph is typically a small (n, r) graph that is used as a template for a large $(Z \times n, Z \times r)$ code graph.

The code graph is constructed from the protograph by making Z copies of each variable and check node. Each edge in the small protograph represents a set of edges in the larger code graph which connect Z copies of a variable node with Z copies of a check node via an arbitrary permutation (see figure 1).

As a matter of terminology, although we use the protograph formalism in [20], other researchers have referred to a "projected graph" [27] or "base graph"[28], which are mostly functionally equivalent.

5.3.1 Decoder Architecture

The basic computation performed in message-passing decoding is a message-update, in which a node computes its set of outgoing messages from its set of incoming messages. In our hardware architecture, the processing elements are variable node units and check node units, each of which computes its respective message updates. These processing elements can be highly decentralized and distributed across the available area. Our strategy is to instantiate hardware units for each of the n variable nodes and r check nodes in the small LDPC protograph. All n variables node units or all r check node units decode synchronously and in parallel. The Z copies of the identical small protograph share this hardware and are operated on serially. The fundamental unit of time is called a "computation cycle", in which a processing element can read the incoming messages from a memory and compute and store outgoing messages. Messages are stored in memory modules, which each correspond to an edge in the small protograph. Each memory module consists of two memory banks capable of storing Z messages and a permutation table. One memory bank stores variable-to-check messages, and is writable by an associated variable node unit and readable by an associated check node unit. The other memory bank stores the check-to-variable messages and is writable by an associated check node unit and readable by an associated variable node unit. The permutation table specifies a permutation $\pi_e : \{1, 2, ..., Z\} \rightarrow \{1, 2, ..., Z\}$ such that if $\pi_e(i) = j$, then the i^{th} variable node is connected to the j^{th} check node.

5.3.2 Computation Scheduling

The cornerstone of our hardware architecture is the scheduling of message-updates in space and time. One iteration consists of a check node phase, followed by a variable node phase. In each phase, there are Z computation cycles. In the check node phase, all check node modules read messages from the edge memory in ascending order, update the messages, and write their results back to the edge memory in ascending order. This computation across all r check node units occurs in parallel.

In the variable node phase, all variable node modules read messages from the edge memory in permuted order, update the messages, and write back the edge memory in permuted order. The computation across all n variable node units also occurs in parallel. The decoding stops at the maximum iteration number, or when a stopping rule is satisfied.

Although this work was underway before the Flarion decoder patent was published, we can now make a useful comparison to that architecture. Flarion's design operates on all Z copies of the template LDPC graph in parallel and processes the individual nodes serially. In this manner, memory and processing can be centralized and a Single-Instruction-stream-Multiple-Data-stream (SIMD) instruction is used to access all Z messages[27].

In contrast, our system has multiple decentralized processing elements with multiple separate memories (see figure 2). All nodes in the template LDPC graph are operated on simultaneously in parallel and each of the Z copies are processed serially (see figure 3).

5.3.3 Structured LDPC Implementation Methodology

1. Choose a small (n, r) protograph by some methods (e.g. [20]).

2. Replicate the protograph Z times and apply a "girth conditioning" algorithm such as Progressive-Edge-Growth (PEG)[29] to permute the end points of each set of edges to obtain a large $(Z \times n, Z \times r)$ code graph that does not contain short cycles.

3. Generate a decoder design by applying the protograph and the chosen permutations to parameterized Verilog HDL

4. Automatically synthesize, place and route design using Xilinx XST

Particular attention is given to the degree distribution of the small protograph chosen, as the larger code graph will have the same degree distribution. For all our protographs implemented, regular $(3, 6)$ protograph was used.

5.4 Quantized Belief Propagation Algorithm

We use the non-uniform quantization scheme proposed in [30], which applies to regular $(3, 6)$ LDPC codes.

Initially, variable nodes read the channel memory, compute initial variable-to-check messages $v_{i\to j}(0)$, and directly deposit into corresponding edge memory according to the permutation tables:

$$v_{i\to j}\left(0\right) = Q_{ch}\left(channel_i\right), i \in \{1..Zn\} \tag{5.1}$$

where $Q_{ch}\left(channel_i\right)$ is the quantization rule for the channel.

At the t^{th} iteration, the parity check phase occurs first. All r check node units read the variable-to-check messages $v_{i\to j}$ from edge memory connecting the i^{th} variable node to the j^{th} check node in the large code graph, update the message by equation (2), then write the check-to-variable messages $u_{j\to i}$ back to the edge memory according to the permutation tables. r check node units are running in parallel, while Z copies of messages are being updated serially.

$$u_{j\to i}\left(t\right) = Q_c(\sum_{i'} \phi_c(v_{i'\to j}(t-1))),\ j \in \{1..Zr\} \tag{5.2}$$

where i' ranges over all edges connected connected to the j^{th} check node excluding i, Q_c is the quantization rule for the check-to-variable message $u_{j\to i}$, and ϕ_c is the reconstruction function for the variable-to-check message $v_{i\to j}$.

Next, the variable phase occurs. n variable node units read the check-to-variable messages $u_{j\to i}$ from edge memory, update the message by equation (3), then write the variable-to-check messages $v_{i\to j}$ back to edge memory according to the permutation tables.

$$v_{i\to j}\left(t\right) = Q_v\left(\phi_{ch}\left(Q_{ch}\left(channel_i\right)\right) + \sum_{j'\neq j} \phi_v\left(u_{j'\to i}(t)\right)\right),\ i \in \{1..Zn\} \tag{5.3}$$

where j' ranges over all edges connected connected to the i^{th} variable node excluding j, Q_v is the quantization rule for the variable-to-check message $v_{i\to j}$, ϕ_v is the reconstruction function for the check-to-variable message $u_{j\to i}$, and ϕ_{ch} is the reconstruction function for the channel message $Q_{ch}\left(channel_i\right)$.

At the final K^{th} iteration, hard decisions X_i are made in variable nodes following:

$$X_i = \begin{matrix} 0,\ \sum_j u_{j\to i}(K) \geq 0 \\ 1,\ \sum_j u_{j\to i}(K) < 0 \end{matrix} \tag{5.4}$$

x	$\phi_{ch}(x)$	$\phi_v(x)$	$\phi_c(x)$
-4	-21	-20	-1
-3	-15	-12	-2
-2	-9	-6	-6
-1	-3	-2	-26
0	3	2	26
1	9	6	6
2	15	12	2
3	21	20	1

Q_{ch} (ch)/ Q_v (v)/ Q_c (c)	ch	v	c
-4	$ch < -3.3$	$v < -18$	$-5 \leq c < 0$
-3	$-3.3 \leq ch < -2.2$	$-18 \leq v < -12$	$-9 \leq c < -5$
-2	$-2.2 \leq ch < -1.1$	$-12 \leq v < -6$	$-26 \leq c < -9$
-1	$-1.1 \leq ch < 0$	$-6 \leq v < 0$	$c < -26$
0	$0 \leq ch \leq 1.1$	$0 \leq v \leq 6$	$c > 26$
1	$1.1 < ch \leq 2.2$	$6 < v \leq 12$	$9 < c \leq 26$
2	$2.2 < ch \leq 3.3$	$12 < v \leq 18$	$5 < c \leq 9$
3	$ch > 3.3$	$v > 18$	$0 \leq c \leq 5$

5.5 Performance

5.5.1 FPGA utilization

The performance of decoder can always be improved by increasing the block length. However, the block length of the LDPC code is limited by the area constraints on the FPGA chip. The LDPC decoder consists of processing units and edge memory. The area consumed by the processing units is proportional to the size of (n, r) protograph, which is proportional to the throughput. The area consumed by the edge memory is proportional to the size of $(n \times Z, r \times Z)$ code graph, which is proportional to the error-correcting capability. We implemented several size of LDPC codes, and measured the utilization of the decoder on a Xilinx Virtex-II 2000 FPGA.

LDPC template (n, r)	Copies Z	Block length $(n \times Z, r \times Z)$	Slice Utilization %
$(64, 32)$	16	$(1024, 512)$	85
$(64, 32)$	32	$(2048, 1024)$	99
$(32, 16)$	32	$(1024, 512)$	53
$(32, 16)$	64	$(2048, 1024)$	66
$(32, 16)$	128	$(4096, 2048)$	97

5.5.2 Speed/Throughput

We measured the real decoding throughput by the FPGA decoder of a $(128 \times 32, 128 \times 16)$ LDPC code at fixed iteration numbers without stopping rules.

# iteration	Throughput without communication overhead (Mbps)	Throughput with communication overhead (Mbps)
1	314.47	10.01
10	31.45	7.74
20	15.72	6.20
50	6.29	3.91
100	3.14	2.41
150	2.10	1.74
200	1.57	1.37
250	1.26	1.12

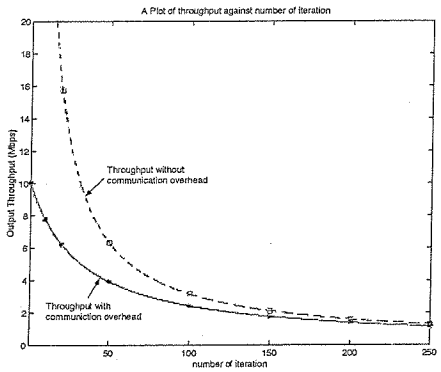


Figure 5.1: Throughput vs. iteration

The measured delay consists of communication overhead and decoder latency, in which decoder latency is proportional to the number of iterations. The decoder latency is 3.18 ns/bit/iteration. The communication overhead is 97.1 ns/bit in our tests. Communication overhead includes the buffer delay outside decoder module, and the time delay writing to and reading from the FPGA board.

5.5.3 Error Correcting Capability

We implemented several codes of different block lengths, and ran performance tests to compare their performance differences. The largest block length code we can implement to fit into a Xilinx Virtex-II 2000 FPGA chip is a $(32, 16) \times 128$ copies $= (4096, 2048)$ code. The results demonstrate that doubling the block length can improve the performance by about 0.5 dB.

The performance of 3-bit non-uniform quantization is another interesting topic to investigate. Compared to the full floating point simulation done in software, hardware 3-bit non-uniform quantization is only off about 0.2 dB, with drastically smaller hardware implement requirements. The speed advantage of the FPGA over software simulation allows the detection of errors down to 10^{-9} BER. The error floor at 10^{-9} BER is resulted from the quantization error.

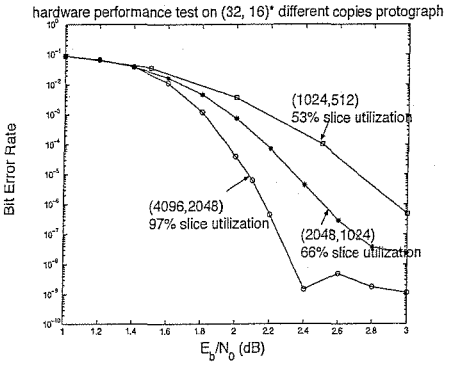


Figure 5.2: Full floating point vs. 3-bit

5.6 Conclusion

We have presented a scalable decoding architecture for a certain class of structured LDPC codes protograph, and demonstrated a FPGA implementation of a (4096, 2048) regular (3, 6) structured LDPC code. Partially parallel structure allows high throughput, while the serial processing of multiple copies of the protograph allows a large block length in implementation to improve the performance. Our use of three-bit non-uniform quantization allows near floating point performance in the waterfall region. As demonstrated by this work, an FPGA implementation of LDPC codes can

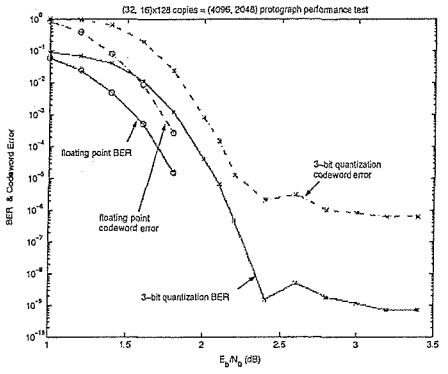


Figure 5.3:

have excellent performance, high throughput, low hardware complexity and easy reconfigurability; FPGA implementation of LDPC codes are expected to be employed for many applications in next-generation communication systems.

Chapter 6

Memory-Efficient Quantized Belief Propagation Decoders

The standard belief propagation algorithm 1.8 that is used to decode Low-Density-Parity-Check codes defines real-valued messages that are passed along edges in a graph. The standard way to simulate this algorithm is to use a very accurate representation of the real numbers, such as floating-point numbers, to store the value of each message. However, for very high-speed decoders, it is clear that the high complexity associated with computing and storing such numbers is to be avoided if possible. Indeed, Gallager's Algorithm A [14] can be seen as a single bit approximation to belief-propagation algorithm. This paper explores some of the ground which lies between the two extremes. In this chapter, we investigate analytically (via density evolution) and through simulation several rules having message size between 1 and 4 bits.

This work was extended by Jason Lee [31] to use slightly different parameters and with new hardware simulations.

6.1 Quantized Belief Propagation

We define a new algorithm which approximates BP, quantized belief propagation (QBP). QBP essentially approximates the BP algorithm as it is stated chapter 1. It operates in each of the additive domains 1.8.

Whereas the BP algorithm is defined by a set of messages m, QBP will be defined in terms of messages m'. While in the standard BP algorithm defines messages $m_v \in \mathbb{R}$, in QBP we define messages $m_v \in M_{ch} = \left\{\pm 1, \pm 2, ..., \pm \frac{q_c}{2}\right\}$. For simplicity, we formulate QBP for

We first give an abstract quantization rule $Q_\tau : X \mapsto \left\{\pm 1, \pm 2, ..., \pm \frac{q_c}{2}\right\}$ where $\tau = \left\{\tau_1, ..., \tau_{\frac{q}{2}-1}\right\}$, $\tau_i \in \mathbb{R}^+$ is a set of thresholds with $\tau_i \, \tau_{i+1}$.

$$Q_\tau(x) = \begin{cases} -\frac{q}{2}, & x < -\tau_{\frac{q}{2}-1} \\ -\arg\min_i \{\tau_i > -x\}, & -\tau_{\frac{q}{2}-1} \leq x < 0 \\ \arg\min_i \{\tau_i > -x\}, & 0 \leq x < \tau_{\frac{q}{2}-1} \\ \frac{q}{2}, & \tau_{\frac{q}{2}-1} < x \end{cases}$$

The reconstruction function $\phi : \left\{\pm 1, \pm 2, ..., \pm\frac{q_c}{2}\right\} \to \mathbb{N}$ is an anti-symmetric function which is characterized by its values on $\left\{1, 2, ...\frac{q}{2}\right\}$.

$$\phi(-i) = -\phi(i)$$

The input to the QBP decoder are defined in terms of the input to the ideal BP decoder and the parameter τ_{ch}:

$$m'_v = -Q_{\tau_{ch}}(m_v)$$

At iteration 0, the message passed along each edge (v, c) is equal to v's input message:

$$m'_v = Q_{\tau_v}(\phi_v(m_v))$$

Each message $m^{(l)}_{c,v}$ are calculated with respect to several of the messages $m^{(l)}_{v',c}$ as:

$$m'^{(l)}_{c,v} = Q_{\tau_c}\left(\sum_{v'|c\neq c|} \phi_v\left(m^{(l-1)}_{c',v}\right)\right)$$

After a sufficient number L of iterations, we estimate the symbol X_v as:

$$X_v = \begin{cases} 0, \phi_{ch}(m_v) + \sum_{c'|v} \phi_v\left(m^{(L)}_{c',v}\right) > 0 \\ 1, \text{ otherwise} \end{cases}$$

Note that if ϕ_{ch} and ϕ_v are such that the sum in the previous equation can be exactly 0, then the algorithm cannot be both symmetric. A simple way to avoid this is for $\phi_{ch}(i)$ to be odd for all i and $\phi_v(i)$ to be even for all i.

6.2 QBP Rules for the $(3, 6)$ Regular LDPC Ensemble

There are a number of ways to test the goodness of different QBP algorithm. A simple and direct way is to simulate the algorithm on particular codes for a given channel. Another way is density evolution, by which we calculate the fractions of edges transmitting each message, assuming infinite block-length. This method essentially calculates the asymptotic performance of the code as the

Name	I	II	III	IV	V[2]
q_{ch}	2	4	4	8	8
q_v	2	4	4	4	8
q_c	2	4	4	4	8
τ_{ch}	∞	1.76	1.4	1.4	1.4
ϕ_{ch}	(1)	(1, 3)	(3, 15)	(1, 3, 5, 7)	(3, 9, 15, 21)
τ_v	()	(2)	(7)	(2)	(6, 12, 18)
ϕ_v	(1)	(1, 3)	(3, 11)	(1, 3)	(2, 6, 8, 12)
τ_c	()	(5)	(5)	(1, 3)	(7, 11, 24)
ϕ_c	(1)	(1, 2)	(1, 2)	(1, 2)	(1, 2, 6, 21)
SNR^*	4.896	2.248	1.955	1.689	1.443
E	0	0	$3 \cdot 10^{-3}$	$7 \cdot 10^{-3}$	$5 \cdot 10^{-4}$

Table 6.1: Quantized rules I through V

Name	VI	VII	Ideal
q_{ch}	8	16	∞
q_v	8	16	∞
q_c	8	16	∞
τ_{ch}	1.1	.66	ε
ϕ_{ch}	(3, 9, 15, 21)	(15, 45, 75, 105, 135, 163, 193, 245)	$\mathbb{R}$
τ_v	(6, 12, 18)	(30, 60, 90, 120, 150, 180, 210)	$\mathbb{R}$
ϕ_v	(2, 6, 12, 20)	(9, 23, 37, 53, 71, 97, 125, 167)	$\mathbb{R}$
τ_c	(5, 9, 26)	(8, 15, 28, 45, 135, 163, 193, 245)	$\mathbb{R}$
ϕ_c	(1, 2, 6, 26)	(8, 15, 28, 45, $*$, $*$, $*$, $*$)	$\mathbb{R}$
SNR^*	1.409	1.199	1.09
E	0	$3 \cdot 10^{-6}$	0

Table 6.2: Quantized rules VI, VII and Ideal

code length n approaches infinity. This has the advantage of being quite fast to compute, but the disadvantage that it may fail to predict performance when there are loops in the graph, as will be seen in section 5.

In this section, density evolution is used to predict the performance of QBP for the class of regular (3,6) LDPC codes. The results are characterized by a value SNR^* for which if $SNR < SNR^*$ we have bad performance, and for which if $SNR > SNR^*$, we have bit error given by E, which may or may not be equal to 0. if $E = 0$, we say the algorithm has no error floor, otherwise it has an error floor. It is not necessarily true that algorithms with higher values of q_{ch}, q_v, and q_c have no error floor if the same holds for an algorithm with lower values. error floor, while more complex algorithms do have error floors.

The following table summarizes the best known QBP rules for a range of interesting values of q_{ch}, q_v, and q_c.

Rule V corresponds precisely to the rule suggested by Richardson, as do the threshold and error-floor predictions, though this is not trivial to see.

6.3 Simulation Results

The following two figure shows the performance of several of the QBP algorithms applied to a regular $(3, 6)$ LDPC code.

In the waterfall region, the performance of each rule is predicted quite well from density evolution. In addition, in each instance where an error floor appears in density evolution, it appears in the simulation. However, the is in fact an error floor on rule II which is not predicted from density evolution. Preliminary analysis strongly suggests that this error floor is in fact due to loops in the graph, as opposed to tree-like configurations as in the other error floors.

6.4 Discussion

Optimization techniques

Error Floors

In general, the error floors inherent in all of the above rules can be understood to be caused by the inability of the strongest internal messages to completely overcome the strongest messages from the channel. This suggests increasing the reliability values and thresholds τ_v and ϕ_v on the internal messages, effectively trading resolution at low reliability levels for range of expression, which would likely decrease the error floor at the expense of a threshold further from capacity. Other methods have also been suggested [9] to mitigate or eliminate error floors, such as artificially lowering [14] the channel messages at a sufficiently late iteration. More work is needed to explore this tradeoff (and should be completed by the time of this presentation)

6.5 Acknowledgements

Kenneth Andrews and Gill Chinn provided the simulation results for several of the decoding rules.

Chapter 7

LDPC Graph Optimization for Parallel Hardware Implementation

in which we search for LDPC code graphs that can be suitably laid out in space so that the graph neighbors are spatially close, for the purpose of decoder implementation

A methodology for generating bipartite graphs for LDPC codes which both exhibit good performance under message passing decoding and are particularly amenable to direct hardware implementation is described. Performance depends nontrivially on the graph, which in many analyses [9] is assumed to be random. Since imposing geometric constraints reduces the graph's randomness, it is possible that the error-correcting performance will degrade. One specific mechanism which can cause performance to degrade is that

To this end, we define an apparently novel quantitative measure of the "loopiness" of a graph, as well as a quantitative measure of the cost of direct hardware implementation, and use the well-known simulated annealing algorithm to simultaneously minimize both quantities. [Finally, we simulate the decoding of several rather short codes to show that the performance is indeed predicted by our loopiness measure.]

7.1 Introduction

Recently, researchers have used large ensembles of very long block-length Low-density-parity-check (LDPC) codes [14] to demonstrate very nearly capacity achieving performance [23], [32] with low encoding and decoding complexity. Less explored, however, is the problem of designing good graphs for short block-length codes. Recently, Mao and Banihashemi have proposed selection of graphs based on the "average girth distribution" [33] with apparent success. MacKay [34] has also used another set of heuristics to modify a random graph into a better one.

In this paper, we [will] make a systematic attempt to find criteria which accurately predict code performance, and use powerful optimization algorithms (as opposed to just selection) to find good

graphs by such criteria. This should help to answer the question of how much the performance of short block-length LDPC's vary over different graphs. Further, we use such optimization algorithms to design codes which not only should have good performance, but will be amenable to easy hardware design.

7.2 Performance and Cost Measures

Regular LDPC's are characterized by a bipartite graph $g = (V, C, E)$, where any edge in E connects a vertex in V with a vertex in C. The vertices in V represent symbols to be transmitted over a channel, while the vertices in c represent parity-check equations of the adjacent edges. In addition, there are λ edges incident with each node in V and ρ edges incident with each node in C.

In this section, we introduce two quantities which can be efficiently calculated and which have to do with the performance and implementation cost, respectively. These quantities are minimized using the Simulated Annealing algorithm, which we discuss in the following section.

7.2.1 Performance Measure

It is well known that the Message-passing algorithm [9], run for i iterations, calculates exactly the a posteriori probability of each code symbol x_j (corresponding to the node v_j)given the received symbols corresponding to nodes in a certain neighborhood of nodes reachable from v_j in $2i$ steps, but only as long as the that neighborhood is tree-like, meaning that it has no cycles. In order to preserve the exactness of the message-passing algorithm for as many iterations as possible, many have suggested maximizing the girth of a graph, the length of it's smallest cycle.

In this, we define a measure of loopiness which counts *all* of the loops in a graph, weighted by an exponential of its length:

$$L(g, \alpha) = \sum_{i=2,4..} \alpha^i N_i \tag{7.1}$$

where N_i represents the number of loops of length i in the graph. A loop is defined as a sequence $v_1, c_2, v_3, ..c_i$ such that an edge exists between v_{2j} and $c_{2j\pm1}$ for all j, and $v_{2j} \neq v_{2j+2}$ and $c_{2j-1} \neq c_{2j+1}$ for all j. All indexes are taken modulo i. It can be shown that this sum converges, and therefore that the measure is defined, only for $\alpha^2 < (\lambda - 1)(\rho - 1)$ for regular LDPC's. For values of α smaller than this threshold, there exist algorithms to efficiently estimate this expression for loopiness of a graph.

7.2.2 Cost Measure

Since the aim of this research is to generate codes whose quantized message-passing algorithms are easily implementable in direct-form on a microchip, we must consider the cost of such an implementation. direct-form realization refers to the method of instantiating many logic units, each dedicated to making the calculations associated with a particular vertex (in V or C) operating simultaneously to perform the decoding. The messages are passed along wires which are instantiated for each edge in the graph.

It is not difficult to see that if the graph is chosen at random and the vertices of the graph of places at random in a square area, the average length of wire increases like $O(n^{\frac{1}{2}})$, and thus the wiring complexity grows like $O(N^{\frac{3}{2}})$ while the complexity of the logic grows like $O(n)$. It is therefore natural to relate the implementation cost to the total length of wires needed to instantiate all of the edges in the graph. For simplicity, we take the cost measure $W(g)$ to be the sum of the Manhattan distance between the endpoints of all edges in the graph, where the nodes occupy fixed positions.

$$W(g) = \sum\nolimits_{e \in E} \text{length}(e) \tag{7.2}$$

7.3 Optimizing with Simulated Annealing

The simulated annealing algorithm is a probabilistic, iterative algorithm designed to approximately solve energy minimization problems, and can be successfully applied to a broad class of such problems. For this problem, we search for a minimum of:

$$E(g) = C_L \cdot L(g, \alpha) + C_W \cdot W(g) \tag{7.3}$$

for some pre-selected α and coefficients C_L and C_W.

In addition to an energy function, a set of transformations $\mathbb{T}(g) = g'$ must also be defined which take a solution into another solution. Such a transformation should have the property that the transformed solution has an energy somewhat similar to the original solution. We take the set of transformation to be those by which two edges, say $v_1 \rightarrow c_1$ and $v_2 \rightarrow c_2$ are replaced with the edges $v_1 \rightarrow c_2$ and $v_2 \rightarrow c_1$.

Finally, we define a positive profile $T(i)$ which defines the "temperature" of the system at each iteration i. Conservatively, $T(i)$ can start at a very high temperature decay exponentially with i to end at a very low temperature. The simulated annealing begins with an arbitrary solution g_0. For each i, the simulated annealing algorithm selects a transformation $\mathbb{T}(g_t)$ and selects

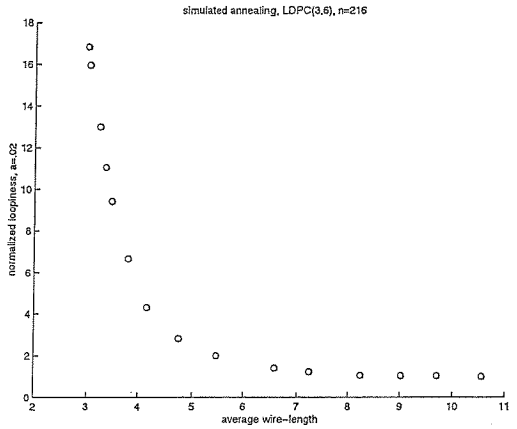


Figure 7.1: Locus of points obtainable by SA

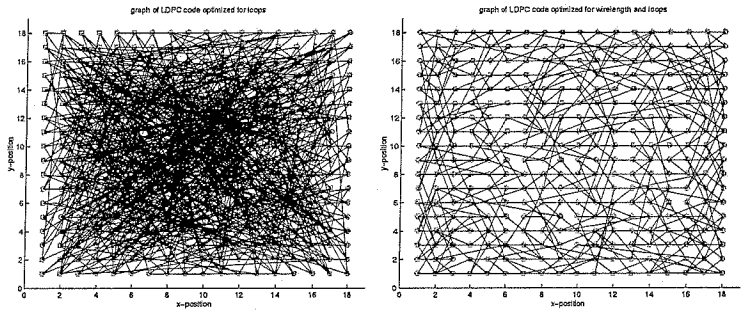


Figure 7.2: graphs optimized primarily for loopiness (left) and wire-length (right)

$$g_{t+1} = \begin{matrix} g_t & w.p \frac{e^{E(g_t)/T(t)}}{A} \\ g'_t & w.p \frac{e^{E(g'_t)/T(t)}}{A} \end{matrix} \tag{7.4}$$

where A is chosen to make this a valid probability distribution, namely $A = e^{E(g_t)/T(t)} + e^{E(g_t)/T(t)}$.

The algorithm terminates at a pre-determined time, and the output of the algorithm is the graph g_{final}. Roughly, the reason for a non-zero temperature that sometimes allows the higher-energy solution to be selected is to avoid local (but not global) minima.

Because there is flexibility in selecting the coefficients C_L and C_W (in fact, only the ratio matters), we can generate a locus of points achievable by the algorithm in terms of $L(g, \alpha)$ and $W(g)$. In the following chart, we plot $L(g, \alpha)/L_0$ vs. $W(g)/n$, where L_0 is the smallest $L(g)$ obtained by the algorithm.

The chart shows that average wirelength can be reduced substantially with extremely little increase in loopiness. However, at very short wirelengths, the graph must inevitably admit shorter loops. The following two layouts show the extrema of the previous chart.

7.4 Evaluation of Performance Measure

In this section, we will give comparisons of the performance achieved by various codes in order to verify (or refute) that our loopiness measure predicts the performance of actual codes. The results of Mao [33] give strong preliminary indication that such a prediction property will hold.

7.5 Conclusion

We have demonstrated an application of the Simulated Annealing algorithm to the problem of designing graphs for LDPC's which can substantially reduce the loopiness of a graph. We have preliminary evidence that this will improve the performance of a code under message-passing decoding.

Additionally, we have shown that the same algorithm allows a very favorable tradeoff in terms of the wiring complexity of the direct implementation. Further decoding simulations will quantify the extent to which loopiness is related to codec performance.

Appendix A

Software License

Appendix B

LDPCWorkbench Source Code

B.1 Data Structures I (graph.cs)

```
using System;
using System.IO;
using System.Collections;
using System.Threading;
using System.Text;

namespace LDPC {

  public class Graph {

    // elements
    public Edge[] edge;
    public VNode[] var;
    public CNode[] check;

    // Auxilary variables
    protected Random random;

    // Constructors

    public Graph() {
      random = new Random();
      var = new VNode[0];
      check = new CNode[0];
      edge = new Edge[0];
    }

    public Graph(Graph graph){
      random = new Random();

      graph.CheckIntegrity();
      graph.Renumber();

      // copy graph
      int[,] matrix = new int[graph.check.Length, graph.var.Length];
      foreach (Edge e in graph.edge)
```

```
    matrix[e.check.index, e.var.index]++;
  bool[] transmitted = new bool[graph.var.Length];
  for(int i=0; i<graph.var.Length; i++) transmitted[i] = graph.var[i].transmitted;
  ConstructFromMatrix(matrix, transmitted);
}

// Constructor Helpers

protected void ConstructFromMatrix(int[,] matrix, bool[] transmitted){
  ConstructFromMatrix(matrix);
  int n = matrix.GetLength(1);
  for (int i=0; i<n; i++)
    var[i].transmitted = transmitted[i];
}

protected void ConstructFromMatrix(int[,] matrix){

  // get parameters
  int n = matrix.GetLength(1);
  int r = matrix.GetLength(0);
  int ne = 0;
  foreach (int i in matrix) ne += i;

  // get degrees
  int[] varDegree = new int[n];
  int[] checkDegree = new int[r];
  for (int i=0; i<n; i++)
    for (int j=0; j<r; j++){
      varDegree[i] += matrix[j,i];
      checkDegree[j] += matrix[j,i];
    }

  // create structures
  edge = new Edge[ne];
  var = new VNode[n];
  for (int i=0; i<n; i++){
    var[i] = new VNode(i);
    var[i].edge = new Edge[varDegree[i]];
  }
  check = new CNode[r];
  for (int i=0; i<r; i++){
    check[i] = new CNode(i);
    check[i].edge = new Edge[checkDegree[i]];
  }

  // place edges
  int edgeIndex = 0;
  int[] varEdgeIndex = new int[n];
  int[] checkEdgeIndex = new int[r];
  for (int i=0; i<n; i++)
    for (int j=0; j<r; j++){
      for (int k=0; k<matrix[j,i]; k++){
        Edge e = new Edge(var[i], check[j]);
        edge[edgeIndex++] = e;
```

```
        var[i].edge[varEdgeIndex[i]++] = e;
        check[j].edge[checkEdgeIndex[j]++] = e;
      }
    }
}

// check the integrity of this graph
// this function is useful when debugging graph modifying functions
public void CheckIntegrity(){
  ArrayList edgeAL = new ArrayList();

  // make sure each edge is valid and connected to valid check and variable nodes
  foreach (Edge e in edge){
    if (e == null)
      throw new Exception("Graph edge list contains null edge.");
    if (edgeAL.Contains(e))
      throw new Exception("Duplicate edges in graph edge list");
    else
      edgeAL.Add(e);
    int vTimes = 0;
    foreach (Edge ee in e.var.edge)
      if (e == ee)
        vTimes++;
    if (vTimes == 0)
      throw new Exception("e.var == v, but v.edge does not contain e.");
    int cTimes = 0;
    foreach (Edge ee in e.check.edge)
      if (e == ee)
        cTimes++;
    if (cTimes == 0)
      throw new Exception("e.check == c, but c.edge does not contain e.");
    e.c2vMessage = e.v2cMessage = 0;
  }

  // make sure each variable has a valid edge list
  foreach (VNode v in var){
    foreach (Edge e in v.edge){
      if (e == null)
        throw new Exception("Variable edge list contains null edge.");
      if (e.c2vMessage == 1)
        throw new Exception("Edge contained in multiple variable nodes.");
      if (edgeAL.Contains(e))
        e.c2vMessage = 1;
      else
        throw new Exception("Variable edge list contains edge not in graph edge list.");
    }
  }

  // make sure each check has a valid edge list
  foreach (CNode c in check){
    foreach (Edge e in c.edge){
      if (e == null)
        throw new Exception("Check edge list contains null edge.");
      if (e.v2cMessage == 1)
```

```
          throw new Exception("Edge contained in multiple check nodes.");
        if (edgeAL.Contains(e))
          e.v2cMessage = 1;
        else
          throw new Exception("Check edge list contains edge not in graph edge list.");
      }
    }
  }

  // sparse graph representation
  public override string ToString(){
    StringBuilder sb = new StringBuilder();
    sb.Append(string.Format("{0} {1} {2}\n", var.Length, check.Length, edge.Length));
    foreach (VNode v in var)
      sb.Append(string.Format("{0} {1} {2}\n", v.index, v.edge.Length, v.transmitted ? 1 : 0));
    foreach (CNode c in check)
      sb.Append(string.Format("{0} {1}\n", c.index, c.edge.Length));
    foreach (Edge e in edge)
      sb.Append(string.Format("{0} {1}\n", e.var.index, e.check.index));
    return sb.ToString();
  }

  // Graph Modifying functions

  public void Renumber(){
    for (int i=0; i<var.Length; i++)
      var[i].index = i;
    for (int i=0; i<check.Length; i++)
      check[i].index = i;
  }

  public Edge AddEdge(int v, int c){
    return AddEdge(var[v], check[c]);
  }

  public Edge AddEdge(VNode v, CNode c){
    Edge e = new Edge(v, c);
    AddEdge(e);
    return e;
  }

  public void AddEdge(Edge e){
    edge = AddEdge(edge, e);
    e.var.edge = AddEdge(e.var.edge, e);
    e.check.edge = AddEdge(e.check.edge, e);
  }

  Edge[] AddEdge(Edge[] edge, Edge e){
    Edge[] edge_ = new Edge[edge.Length + 1];
    for (int i=0; i<edge.Length; i++)
      edge_[i] = edge[i];
    edge_[edge_.Length - 1] = e;
    return edge_;
  }
```

```
  public void RemoveEdge(Edge e){
    edge = RemoveEdge(edge, e);
    e.var.edge = RemoveEdge(e.var.edge, e);
    e.check.edge = RemoveEdge(e.check.edge, e);
  }

  Edge[] RemoveEdge(Edge[] edge, Edge e){
    Edge[] edge_ = new Edge[edge.Length - 1];
    int index = 0;
    for (int i=0; i<edge.Length; i++)
      if (edge[i] != e)
        edge_[index++] = edge[i];
    return edge_;
  }

  VNode[] RemoveVNode (VNode[] list, VNode v){
    VNode[] list_ = new VNode[list.Length - 1];
    int index = 0;
    foreach (VNode vv in list)
      if (vv != v)
        list_[index++] = vv;
    return list_;
  }

  CNode[] RemoveCNode (CNode[] list, CNode c){
    CNode[] list_ = new CNode[list.Length - 1];
    int index = 0;
    foreach (CNode cc in list)
      if (cc != c)
        list_[index++] = cc;
    return list_;
  }

  protected void RemoveVNode(VNode n){
    foreach (Edge e in n.edge){
      edge = RemoveEdge(edge, e);
      e.check.edge = RemoveEdge(e.check.edge, e);
    }
    var = RemoveVNode(var, n);
  }
  protected void RemoveCNode(CNode n){
    foreach (Edge e in n.edge){
      edge = RemoveEdge(edge, e);
      e.var.edge = RemoveEdge(e.var.edge, e);
    }
    check = RemoveCNode(check, n);
  }

  public void SwapEdges(Edge e0, Edge e1){
    CNode c0 = e0.check;
    CNode c1 = e1.check;
    e0.check = c1;
    e1.check = c0;
```

```
      for (int i=0; i<c0.edge.Length; i++)
        if (c0.edge[i] == e0)
          c0.edge[i] = e1;
      for (int i=0; i<c1.edge.Length; i++)
        if (c1.edge[i] == e1)
          c1.edge[i] = e0;
    }

    public void Sort(){
      Array.Sort(var);
      Array.Sort(check);
      Array.Sort(edge);
      for (int i=0; i<var.Length; i++){
        var[i].index = i;
      }
      for (int i=0; i<check.Length; i++){
        check[i].index = i;
      }
    }

    public void SwapVNodes(int v0, int v1){
      VNode t = var[v0];
      var[v0] = var[v1];
      var[v1] = t;
      var[v0].index = v0;
      var[v1].index = v1;
    }
  }

  public class Edge : IComparable{
    public VNode var;
    public CNode check;
    public double v2cMessage;
    public double c2vMessage;

    public Edge(){
    }
    public Edge(VNode v, CNode c) {
      this.var = v;
      this.check = c;
    }
    public int CompareTo(object o){
      if (o.GetType() != typeof(Edge)) return -1;
      Edge e = (Edge)o;
      if (var.edge.Length != e.var.edge.Length) return (var.edge.Length - e.var.edge.Length);
      if (var.index != e.var.index) return (var.index - e.var.index);
      return (check.index - e.check.index);
    }
  }

  public class VNode : Node{
    public VNode(int index) : base(index){}
  }
```

```
    public class CNode : Node{
      public CNode(int index) : base(index){}
    }

    public abstract class Node : IComparable{
      public int index;
      public Edge[] edge;
      public double val;

      public bool transmitted;
      public bool zero;

      public int degree{
        get{
          return edge.Length;
        }
      }

      public Node(int index) {
        this.index = index;
      }

      public int CompareTo(object o){
        if (o.GetType() != typeof(Node))
          return 1;
        Node n = (Node)o;
        if (transmitted != n.transmitted)
          return transmitted ? -1 : 1;
        if (edge.Length == n.edge.Length)
          return (index - n.index);
        return edge.Length - n.edge.Length;
      }
    }
  }
```

B.2 Data Structures II (ldpc.cs)

```
using System;
using System.IO;
using System.Collections;
using System.Xml;
using System.Reflection;

namespace LDPC {
  ///<summary>
  ///LDPC implements all of the basic functionality associated with LDPC codes.
  ///The encoding is based on a systematic generator matrix.  Decoding is via
  ///the well-known message-passing or belief-propagation algorithm.  In addition,
  ///this class provides a method based on simulated annealing to remove short
  ///loops from the graphs in order to improve the performance of short block
  ///length codes, as well as edge-length constrained codes.
  ///</summary>
```

```
public abstract class Code {
  public int n;
  public int k;
  public int r;

  public virtual double rate{get{return (double)k/(double)n;}}
  public abstract BitArray Encode(BitArray m);
  public abstract BitArray Decode(double[] llr);
  public abstract void Display();
}

public class LDPC : Code {

  // The unique name of this code
  public string name;

  // LDPC parity graph
  public AnnotatedGraph graph;

  // the generator and parity check matrices
  public BitArray[] genMatrix;
  public BitArray[] checkMatrix;

  // circulant generator and parity check matrices
  public BitArray[,] circulantGenMatrix;
  public BitArray[,] circulantCheckMatrix;

  // decoding parameters
  int maxIter = 100;

  public LDPC(){}

  public LDPC(string filename) {
    graph = new AnnotatedGraph(filename);
    ReadGraphParameters();
    string[] fileTokens = filename.Split('/');
    this.name = fileTokens[fileTokens.Length - 1].Split('.')[0];
  }

  public void ReadGraphParameters(){
    this.n = graph.nt + graph.nu;
    this.r = graph.nr;
    this.k = this.n - this.r;
  }

  public override double rate{
    get{
      return graph.rate;
    }
  }

  public LDPC(AnnotatedGraph g){
    graph = g;
```

```
  ReadGraphParameters();
}

public bool IsWord(BitArray cw){

  // check whether the word has the right length
  if (cw.Length != graph.var.Length) return false;

  // check whether the word satisfies all checks
  foreach (CNode c in graph.check){
    bool okay = true;
    foreach (Edge e in c.edge)
      okay ^= cw[e.var.index];
    if (!okay)
      return false;
  }
  return true;
}

public void GenMatrices(){

  Console.Write("Generating matrices...");
  checkMatrix = new BitArray[graph.check.Length];
  for (int i=0; i<graph.check.Length; i++)
    checkMatrix[i] = new BitArray(graph.var.Length);
  foreach (Edge e in graph.edge)
    checkMatrix[e.check.index][e.var.index] ^= true;

  // systematize parity matrix by Gauss elimination (row operations)
  for (int i=0; i<r; i++){
    bool good = false;
    int j;
    for (j=0; j<n; j++){
      if (checkMatrix[i][j] == true){
        good = true;
        SwapColumns(j, i+k);
        break;
      }
    }
    if (!good) {
      throw new Exception("parity matrix is singular.");
    }
    if (!checkMatrix[i][i+k]) {
      throw new Exception(i+":"+j+":"+k);
    }

    // eliminate all 1's in (i+k)th column after ith row
    for (int l=0; l<r; l++)
      if (l != i & checkMatrix[l][i+k] == true)
        checkMatrix[l].Xor(checkMatrix[i]);
  }

  // generate systematic generator matrix from systematic parity matrix
  genMatrix = new BitArray[k];
```

```
    for (int i=0; i<k; i++){
      genMatrix[i] = new BitArray(graph.var.Length);
      genMatrix[i][i] = true;
      for (int j=k; j<graph.var.Length; j++)
        genMatrix[i][j] = checkMatrix[j-k][i];
    }

    // renumber vertices
    graph.Renumber();

    Console.WriteLine("done.");
  }

  void SwapColumns(int a, int b){
    if (a == b) return;
    graph.SwapVNodes(a,b);
    for (int i=0; i<r; i++){
      checkMatrix[i][b] ^= checkMatrix[i][a];
      checkMatrix[i][a] ^= checkMatrix[i][b];
      checkMatrix[i][b] ^= checkMatrix[i][a];
    }
  }

  void SwapRows(int a, int b){
    if (a == b) return;
    for (int i=0; i<n; i++){
      checkMatrix[b][i] ^= checkMatrix[a][i];
      checkMatrix[a][i] ^= checkMatrix[b][i];
      checkMatrix[b][i] ^= checkMatrix[a][i];
    }
  }

  override public void Display() {
    Console.WriteLine("     n:"+n);
    Console.WriteLine("     k:"+k);
    Console.WriteLine("     r:"+r);
  }

  public void Save(string filename){

    StreamWriter sw = new StreamWriter(filename);
    sw.Write(graph.Annotation());
    sw.Write(graph.ToString());
    sw.Close();
  }

  public void SaveTxt(string filename){
    StreamWriter sw = new StreamWriter(filename);
    graph.Renumber();
    foreach (VNode v in graph.var)
      foreach (Edge e in v.edge)
        sw.WriteLine(@"{0} {1} 1", e.check.index + 1, e.var.index + 1);
    sw.WriteLine(@"{0} {1} 0", graph.check.Length, graph.var.Length);
    sw.Close();
```

```
}

// code functions Encode() and Decode()

override public BitArray Encode(BitArray m){
  BitArray cw = new BitArray(graph.var.Length);
  for (int i=0; i<k; i++)
    if (m[i])
      cw = cw.Xor(genMatrix[i]);
  return cw;
}

override public BitArray Decode(double[] llr){
  int iter;
  return (Decode(llr, maxIter, out iter));
}

// this implementation of the message-passing algorithm is in the multiplicative
// domains, which are related by the bilinear transform.

public BitArray Decode(double[] llr, int maxIter, out int iter){

  // add very small noise to the exactly zero llr's
  Random random = new Random();
  for (int i=0; i<llr.Length; i++)
    if (llr[i] == 0)
      llr[i] = (random.NextDouble() - .5) * .0000001;

  double[] lr = new double[llr.Length];
  for (int i=0; i<llr.Length; i++)
    lr[i] = Math.Exp(llr[i]);

  BitArray dMsg = new BitArray(graph.var.Length);
  foreach (Edge e in graph.edge)
    e.c2vMessage = 1;
  for (iter=0; iter<maxIter; iter++){
    foreach (VNode v in graph.var)
      v.val = lr[v.index];
    foreach (Edge e in graph.edge)
      e.var.val *= e.c2vMessage;
    foreach (Edge e in graph.edge)
      e.v2cMessage = Bilin(e.var.val / e.c2vMessage) * .999999;
    foreach (CNode c in graph.check)
      c.val = 1;
    foreach (Edge e in graph.edge)
      e.check.val *= e.v2cMessage;
    foreach (Edge e in graph.edge)
      e.c2vMessage = Bilin(e.check.val / e.v2cMessage);

    // check if we're at a codeword
    dMsg = new BitArray(graph.var.Length);
    for (int i=0; i<n; i++)
      dMsg[i] = graph.var[i].val > 1;
    if (IsWord(dMsg))
```

```
            break;
        }
        return (dMsg);
      }

      double Bilin(double x){
        return (1-x)/(1+x);
      }
    }
  }
```

B.3 Density Evolution (density.cs)

```
using System;
using System.Collections;

namespace LDPC {

  public class DE {

    // graph we're operating on
    public AnnotatedGraph graph;

    // Chung DE parameters
    static int[,] uTable;
    static int[,] rTable;
    static int nLevels;
    static double reliabilityRange;
    static double unreliabilityRange;
    static double reliabilityQuant;
    static double unreliabilityQuant;

    public DE(AnnotatedGraph graph) {
      this.graph = graph;
    }

    public double  Predict(double x1, double x2, double x3){
      double y1 = EvolveDensity(x1, 1000);
      double y2 = EvolveDensity(x2, 1000);
      double y3 = EvolveDensity(x3, 1000);

      double dx = (x2 - x3) / (x1 - x3);
      double dy = (y2 - y1) / (y3 - y1);

      double t = dx * dy / (1 - dx - dy);
      double x0 = x3 - t * (x1 - x3);

      return x0;
    }

    public double DEThreshold(int accuracy){
      int maxIter = 500;
```

```
    return DEThreshold(accuracy, maxIter);
  }

  public double DEThreshold(int accuracy, int maxIter){
    double delta = 1e-3;
    return DEThreshold(accuracy, maxIter, delta);
  }

  public double DEThreshold(int accuracy, int maxIter, double delta){
    double targetError = 1e-12;
    return DEThreshold(accuracy, maxIter, delta, targetError);
  }

  public double DEThreshold(int accuracy, int maxIter, double delta, double targetError){

    SetDEParams(accuracy);

    // if the threshold is above 10.0 dB, assume infinite threshold
    if (EvolveDensity(10.0, maxIter) == maxIter)
      return double.PositiveInfinity;

    double rcaThreshold = RCAThreshold();

    double snrMax = rcaThreshold + .1;
    double snrMin = rcaThreshold - .1;

    // increase the bounds until proper
    while (EvolveDensity(snrMax, maxIter) == maxIter)
      snrMax = 2 * snrMax - snrMin;
    while (EvolveDensity(snrMin, maxIter) != maxIter)
      snrMin = 2 * snrMin - snrMax;
    double snr = (snrMax + snrMin) / 2;

    // decrease the bounds until interval is smaller than delta
    while (snrMax - snrMin > delta * 2){
      snr = (snrMax + snrMin) / 2;
      if (EvolveDensity(snr, maxIter) == maxIter)
        snrMin = snr;
      else
        snrMax = snr;
    }
    // debug
    return (Predict(snrMax, 2 * snrMax - snrMin, 3 * snrMax - 2 * snrMin));

  }

  public double ECThreshold(){
    int maxIter = 500;
    return ECThreshold(maxIter);
  }

  public double ECThreshold(int maxIter){
    double delta = 1e-3;
    return ECThreshold(maxIter, delta);
```

```
 }

 public double ECThreshold(int maxIter, double delta){
   double targetError = 1e-12;
   return ECThreshold(maxIter, delta, targetError);
 }

 public double ECThreshold(int maxIter, double delta, double targetError){

   double pMax = 1;
   double pMin = 0;

   // decrease the bounds until interval is smaller than delta
   while (pMax - pMin > delta){
     double p = (pMax + pMin) / 2;
     if (EvolveDensityEC(p, maxIter, targetError) == maxIter)
       pMax = p;
     else
       pMin = p;
   }
   return pMin;
}

public double EvolveDensityEC(double p, int maxIter, double targetError){

  double targetSumError = targetError * graph.var.Length;
  double lastSumError = 0;

  foreach (Edge e in graph.edge){
    e.c2vMessage = 0;
  }
  for (int iter=0; iter<maxIter; iter++){
    foreach (VNode v in graph.var)
      v.val = v.transmitted ? p : 1-1e-10;
    foreach (Edge e in graph.edge)
      e.var.val *= 1 - e.c2vMessage;
    foreach (Edge e in graph.edge)
      e.v2cMessage = e.var.val / (1 - e.c2vMessage);
    foreach (CNode c in graph.check)
      c.val = 1;
    foreach (Edge e in graph.edge)
      e.check.val *= (1 - e.v2cMessage);
    foreach (Edge e in graph.edge)
      e.c2vMessage = e.check.val / (1 - e.v2cMessage);

    // check if the sum bit error probability is less than the target.
    double sumError = 0;
    foreach (VNode v in graph.var){
      sumError += v.val;
    }
    if (sumError < targetSumError)
      return iter - (targetSumError - sumError) / (lastSumError - sumError);
    lastSumError = sumError;
}
```

```
    return maxIter;
  }


  public double RCAThreshold(){
    int maxIter = 500;
    return RCAThreshold(maxIter);
  }

  public double RCAThreshold(int maxIter){
    double delta = 1e-3;
    return RCAThreshold(maxIter, delta);
  }

  public double RCAThreshold(int maxIter, double delta){

    // debug change positive to negative infinity

    // if the threshold is above 10.0 dB, assume infinite threshold
    if (EvolveDensityRCA(10.0, maxIter) == maxIter)
      return double.NegativeInfinity;

    double snrMax = 2.0;
    double snrMin = 0.0;

    // increase the bounds until proper
    while (EvolveDensityRCA(snrMax, maxIter) == maxIter)
      snrMax = 2 * snrMax - snrMin;
    while (EvolveDensityRCA(snrMin, maxIter) != maxIter)
      snrMin = 2 * snrMin - snrMax;
    double snr = (snrMax + snrMin) / 2;

    // decrease the bounds until interval is smaller than delta
    while (snrMax - snrMin > delta * 2){
      snr = (snrMax + snrMin) / 2;
      if (EvolveDensityRCA(snr, maxIter) == maxIter)
        snrMin = snr;
      else
        snrMax = snr;
    }
    return snrMax;
  }

  public double EvolveDensityRCA(double snr, int maxIter){

    // debug change target from 1e-12 to 1e-8
    double targetError = 1e-8;
    return EvolveDensityRCA(snr, maxIter, targetError);
  }

  public double EvolveDensityRCA(double snr, int maxIter, double targetError){

    //return (Math.Sign(snr));
```

```
    double rate = graph.rate;

    double sigma = Util.SNR2Sigma(snr, rate);
    double power = Math.Pow(sigma,-2);
    double targetSumError = targetError * graph.var.Length;
    int iter;
    double lastSumError = 1;

    foreach (Edge e in graph.edge){
      e.c2vMessage = 0;
    }
    for (iter=0; iter<maxIter; iter++){
      foreach (VNode v in graph.var)
        v.val = v.zero ? double.PositiveInfinity : (v.transmitted ? power : 0);
      foreach (Edge e in graph.edge)
        e.var.val += e.c2vMessage;
      foreach (Edge e in graph.edge){
        double p = e.var.val - e.c2vMessage;
        e.v2cMessage = Util.Reciprocal(p); // debug
      }
      foreach (CNode c in graph.check)
        c.val = 0;
      foreach (Edge e in graph.edge)
        e.check.val += e.v2cMessage;
      foreach (Edge e in graph.edge)
        e.c2vMessage = Util.Reciprocal(e.check.val - e.v2cMessage);

      // check if the sum bit error probability is less than the target.
      double sumError = 0;
      foreach (VNode v in graph.var){
        double t = Math.Exp(v.val);
        sumError += 1 / (1 + t);
      }
      if (sumError < targetSumError)
        return iter - (Math.Log(sumError) - Math.Log(targetError)) / (Math.Log(sumError) - Math.Log(lastSumError));
      lastSumError = sumError;
    }
    return iter;
  }

  public double EvolveDensity(double snr, int maxIter){
    double targetError = 1e-5;
    return EvolveDensity(snr, maxIter, targetError);
  }

  public double EvolveDensity(double snr, int maxIter, double targetError){

    double targetSumError = targetError * graph.var.Length;
    double lastSumError = 0;

    // let all densities be passed in reliability range
    Hashtable v2cDensity = new Hashtable();
    Hashtable c2vDensity = new Hashtable();
```

```
// initialization
foreach (Edge e in graph.edge){
  v2cDensity[e] = (e.var.transmitted) ? ChannelDensity(snr) : NullRDensity();
}

// iteration
for (int iter=0; iter<maxIter; iter++){

  // check node update
  foreach (CNode c in graph.check){
    int n = c.edge.Length;

    double[][] fCumConvolution = new double[n][];
    fCumConvolution[0] = NullUDensity();
    for(int i=1; i<n; i++){
      fCumConvolution[i] = ConvolveUnreliability(fCumConvolution[i-1], (double[])v2cDensity[c.edge[i - 1]]);
    }
    double[][] bCumConvolution = new double[n][];
    bCumConvolution[n-1] = NullUDensity();
    for (int i=n-1; i>0; i--){
      bCumConvolution[i-1] = ConvolveUnreliability(bCumConvolution[i], (double[])v2cDensity[c.edge[i]]);
    }
    for (int i=0; i<n; i++){
      c2vDensity[c.edge[i]] = ConvolveUnreliability(fCumConvolution[i], bCumConvolution[i]);
    }
  }

  // variable node update
  foreach (VNode v in graph.var){
    int n = v.edge.Length;
    double[][] edgeReliability = new double[n][];
    for (int i=0; i<n; i++)
      edgeReliability[i] = (double[])c2vDensity[v.edge[i]];
    double[][] fCumConvolution = new double[n][];
    fCumConvolution[0] = (v.transmitted) ? ChannelDensity(snr) : NullRDensity();
    for(int i=1; i<n; i++)
      fCumConvolution[i] = ConvolveReliability(fCumConvolution[i-1], edgeReliability[i-1]);
    double[][] bCumConvolution = new double[n][];
    bCumConvolution[n-1] = NullRDensity();
    for (int i=n-1; i>0; i--)
      bCumConvolution[i-1] = ConvolveReliability(bCumConvolution[i], edgeReliability[i]);
    for (int i=0; i<n; i++){
      v2cDensity[v.edge[i]] = ConvolveReliability(fCumConvolution[i], bCumConvolution[i]);
    }
  }

  // renormalize all densities every iteration
  foreach (Edge e in graph.edge){
    Normalize((double[])v2cDensity[e]);
  }

  // check for convergence every iteration
  double sumError = 0;
  foreach (VNode v in graph.var){
```

```
        double[] vDensity = v.transmitted ? ChannelDensity(snr) : NullRDensity();
        foreach (Edge e in v.edge)
          vDensity = ConvolveReliability(vDensity, (double[])c2vDensity[e]);
        double goodDensity = 0;
        for (int i=0; i<vDensity.Length/2; i++)
          goodDensity += vDensity[i];
        sumError += 1 - goodDensity;

      }
      if (sumError < targetSumError)
        return iter - (Math.Log(sumError) - Math.Log(targetError)) / (Math.Log(sumError) - Math.Log(lastSumError));
      lastSumError = sumError;
    }

    return maxIter;
  }

  double[] ConvolveReliability(double[] r0, double[] r1){
    double[] ret = new double[nLevels];
    for (int i=0; i<nLevels; i++){
      for (int j=0; j<nLevels; j++){
        ret[rTable[i,j]] += r0[i] * r1[j];
      }
    }
    return ret;
  }

  double[] ConvolveUnreliability(double[] r0, double[] r1){
    double[] ret = new double[nLevels];
    for (int i=0; i<nLevels; i++){
      for (int j=0; j<nLevels; j++){
        ret[uTable[i,j]] += r0[i] * r1[j];
      }
    }
    return ret;
  }

  public double[] ChannelDensity(double snr){
    double[] channelDensity = Util.ChannelDensity(snr, reliabilityQuant, nLevels, graph.rate);
    return channelDensity;
  }

  double[] NullRDensity(){
    double[] nullRDensity = new double[nLevels];
    nullRDensity[nLevels / 2] = 1.0;
    return nullRDensity;
  }

  double[] NullUDensity(){
    double[] nullRDensity = new double[nLevels];
    nullRDensity[0] = 1.0;
    return nullRDensity;
  }
```

```
void Normalize(double[] density){
  double sum = 0;
  foreach (double d in density)
    sum += d;
  for (int i=0; i<density.Length; i++)
    density[i] /= sum;
  if (sum == 0)
    throw new Exception("divide by 0");
}

public void SetDEParams(int accuracy){
  nLevels = 1<<accuracy;
  reliabilityRange = 12.0;
  unreliabilityRange = 2.5;
  reliabilityQuant = (2 * reliabilityRange) / nLevels;
  unreliabilityQuant = (2 * unreliabilityRange) / nLevels;

  ConstructTables();
}

void ConstructTables(){
  rTable = new int[nLevels, nLevels];
  uTable = new int[nLevels, nLevels];
  for (int i=0; i<nLevels; i++){

    double r0 = (i - (nLevels / 2)) * reliabilityQuant;
    bool sign0 = (r0 > 0);
    double u0 = (r0 == 0) ? double.PositiveInfinity : -Math.Log(-Bilin(Math.Exp(Math.Abs(r0))));

    for (int j=0; j<nLevels; j++){

      double r1 = (j - (nLevels / 2)) * reliabilityQuant;
      bool sign1 = (r1 > 0);
      double u1 = (r1 == 0) ? double.PositiveInfinity : -Math.Log(-Bilin(Math.Exp(Math.Abs(r1))));

      double u = u0 + u1;
      bool sign = sign0 ^ sign1;
      double r = -Math.Log(-Bilin(Math.Exp(Math.Abs(u))));
      int rq = (int)(r / reliabilityQuant + 0.5);
      int k = sign ? (nLevels / 2) + rq : (nLevels / 2) - rq;
      k = Math.Max(k, 1);
      k = Math.Min(k, nLevels - 1);

      uTable[i,j] = k;
      if (i == 0)
        uTable[i,j] = j;
      if (j == 0)
        uTable[i,j] = i;

      int kk = i + j - nLevels / 2;
      kk = Math.Max(kk, 1);
      kk = Math.Min(kk, nLevels - 1);
```

```
          rTable[i,j] = kk;
        }
      }
    }

    double Bilin(double x){
      return (1 - x) / (1 + x);
    }
  }
}
```

B.4 Simulated Annealing (optimizer.cs)

```
using System;
using System.Collections;
using System.Threading;

using Workbench;

namespace LDPC {

  public class Optimizer {

    // debug objects
    Workbench.Workbench workbench;

    // instance variables
    public AnnotatedGraph graph;
    public DE de;
    Random random;

    // Anneal variables;
    double energy;
    int maxPerturbationType;
    GraphType targetGraphType;

    public Optimizer() {
      random = new Random();
    }

    public Optimizer(AnnotatedGraph graph) : this(){
      this.graph = graph;
      de = new DE(graph);
      maxPerturbationType = 5;
    }

    public void SetWorkbench(Workbench.Workbench workbench){
      this.workbench = workbench;
    }

    public void SetMaxPerturbationType(int maxPerturbationType){
      this.maxPerturbationType = maxPerturbationType;
```

```
}

int annealIter;
int maxIter;
int minVarDegree;
int maxVarDegree;
int minCheckDegree;
int maxCheckDegree;

public void Anneal(
  int annealIter,
  int maxIter,
  int maxVarDegree,
  int maxCheckDegree
  ){
  this.annealIter = annealIter;
  this.maxIter = maxIter;
  this.maxVarDegree = maxVarDegree;
  this.maxCheckDegree = maxCheckDegree;
  Anneal();
}

public void Anneal(){

  // degrees less than these do not make sense.
  minVarDegree = 1;
  minCheckDegree = 3;

  // parameters
  double startTemperature = .01;
  double endTemperature = .00001;
  targetGraphType = GraphType.Linear;

  double temperature = startTemperature;

  energy = de.RCAThreshold(maxIter, temperature * .01);

  for (int i=0; i<annealIter; i++) {
    lock(typeof(Thread)){

      temperature = Math.Exp(Math.Log(startTemperature) - Math.Log(startTemperature / endTemperature) * i / annealIter);

      Perturbation p = GetPerturbation();

      // Make sure the constraints on degrees are not violated
      if (CheckDegrees(p) == false) continue;

      // Make sure the graph has the correct asymptotic type
      if (GraphTypeGood(p) == false) continue;

      // see if the energy is low enough
      double maxDiff = temperature * -Math.Log(random.NextDouble());
      if (IsGood(p, maxDiff) == false) continue;
```

```
        // commit the change
        CommitPerturbation(p);

        energy = de.RCAThreshold(maxIter, temperature * 0.01);
        Write(string.Format("{0} RCAThresh : {1:N4} backbone nodes : {2} iter : {3}\r\n", p.Representation(), energy, 0, i));
      }
    }
    graph.Annotate();
  }

  bool IsGood(Perturbation p, double maxDiff){

    double thresh = energy;

    bool isGood = true;

    CommitPerturbation(p);
    if (de.EvolveDensityRCA(thresh + maxDiff, maxIter) == maxIter)
      isGood = false;
    UndoPerturbation(p);

    return isGood;
  }

  bool GraphTypeGood(Perturbation p){

    graph.CheckIntegrity();

    bool isGood = true;

    CommitPerturbation(p);

    GraphType graphType = new TypedGraph(graph).DestructivelyGetGraphType();
    if (graphType < targetGraphType)
      isGood = false;

    UndoPerturbation(p);

    return isGood;
  }

  public bool HasMultipleEdge(int m){
    return HasMultipleEdge(graph, m);
  }

  public bool HasMultipleEdge(Graph graph, int m){
    int[,] connection = new int[graph.var.Length, graph.check.Length];
    foreach (Edge e in graph.edge){
      connection[e.var.index, e.check.index]++;
      if (connection[e.var.index, e.check.index] >= m)
        return true;
    }
    return false;
  }
```

```
  public Perturbation GetPerturbation(){

    Perturbation p = new Perturbation();

    while(true){

      p.type = (PerturbationType)(int)(random.NextDouble() * maxPerturbationType);

      if (p.type == 0){
        // swap two edges
        int j = 0;
        int k = 0;
        while (j==k){
          j = random.Next(graph.edge.Length);
          k = random.Next(graph.edge.Length);
        }
        p.edge0 = graph.edge[j];
        p.edge1 = graph.edge[k];
        if (p.edge0.var == p.edge1.var) continue;
        if (p.edge0.check == p.edge1.check) continue;
        break;
      }

      if (p.type == PerturbationType.ChangeEdgeVar){
        p.edge0 = graph.edge[random.Next(graph.edge.Length)];
        p.edge1 = new Edge(graph.var[random.Next(graph.var.Length)], p.edge0.check);
        if (p.edge0.var == p.edge1.var) continue;
        break;
      }

      if (p.type == PerturbationType.ChangeEdgeCheck){
        p.edge0 = graph.edge[random.Next(graph.edge.Length)];
        p.edge1 = new Edge(p.edge0.var, graph.check[random.Next(graph.check.Length)]);
        if (p.edge0.check == p.edge1.check) continue;
        break;
      }

      if (p.type == PerturbationType.RemoveEdge){
        p.edge0 = graph.edge[random.Next(graph.edge.Length)];
        break;
      }

      if (p.type == PerturbationType.AddEdge){
        p.edge0 = new Edge(graph.var[random.Next(graph.var.Length)], graph.check[random.Next(graph.check.Length)]);
        break;
      }
    }
    return p;
  }

  public void CommitPerturbation(Perturbation p){
    if (p.type == PerturbationType.SwapEdges){
      graph.SwapEdges(p.edge0, p.edge1);
```

```
    }
    if (p.type == PerturbationType.ChangeEdgeVar){
      graph.AddEdge(p.edge1);
      graph.RemoveEdge(p.edge0);
    }
    if (p.type == PerturbationType.ChangeEdgeCheck){
      graph.AddEdge(p.edge1);
      graph.RemoveEdge(p.edge0);
    }
    if (p.type == PerturbationType.RemoveEdge){
      graph.RemoveEdge(p.edge0);
    }
    if (p.type == PerturbationType.AddEdge){
      graph.AddEdge(p.edge0);
    }
    p.committed = true;
  }

  public void UndoPerturbation(Perturbation p){
    if (p.committed == false)
      throw new Exception("exception not committed");
    if (p.type == PerturbationType.SwapEdges)
      graph.SwapEdges(p.edge0, p.edge1);
    if (p.type == PerturbationType.ChangeEdgeVar){
      graph.RemoveEdge(p.edge1);
      graph.AddEdge(p.edge0);
    }
    if (p.type == PerturbationType.ChangeEdgeCheck){
      graph.RemoveEdge(p.edge1);
      graph.AddEdge(p.edge0);
    }
    if (p.type == PerturbationType.RemoveEdge)
      graph.AddEdge(p.edge0);
    if (p.type == PerturbationType.AddEdge)
      graph.RemoveEdge(p.edge0);
    p.committed = false;
  }

  public bool CheckDegrees(Perturbation p){

    if (p.type == PerturbationType.ChangeEdgeVar){
      if (p.edge1.var.degree >= maxVarDegree) return false;
      if (p.edge0.var.degree <= minVarDegree) return false;
    }
    if (p.type == PerturbationType.ChangeEdgeCheck){
      if (p.edge1.check.degree >= maxCheckDegree) return false;
      if (p.edge0.check.degree <= minCheckDegree) return false;
    }
    if (p.type == PerturbationType.RemoveEdge){
      if (p.edge0.var.edge.Length <= minVarDegree) return false;
      if (p.edge0.check.edge.Length <= minCheckDegree) return false;
    }
    if (p.type == PerturbationType.AddEdge){
      if (p.edge0.var.edge.Length >= maxVarDegree) return false;
```

```
        if (p.edge0.check.edge.Length >= maxCheckDegree) return false;
      }
      return true;
    }


    public void Write(string format, params object[] o){
      workbench.Write(format, o);
    }

  }

  public class Perturbation{
    public bool committed;
    public PerturbationType type;
    public Edge edge0;
    public Edge edge1;

    public Perturbation(){
    }

    public string Representation(){
      return new string[]{"*","\\","/","+","-"}[(int)type];
    }
  }

  public enum PerturbationType : int{
    SwapEdges = 0,
    ChangeEdgeVar = 1,
    ChangeEdgeCheck = 2,
    AddEdge = 3,
    RemoveEdge = 4,
  }
}
```

Bibliography

[1] M. Klimesh, *Optimal Strategies for Communicator-Jammer Problems.* PhD thesis, University of Michigan, Ann Arbor, 1995.

[2] M. Luby, "LT codes," 43rd Annual IEEE Symposium on the Foundations of Computer Science.

[3] C. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, pp. 379–423 and 623–656, July and October 1948.

[4] R. M. Tanner, "On graph constructions for ldpc codes by quasi-cyclic extension," *Information, Coding and Mathematics*, pp. 209–220, 2002.

[5] T. Richardson and R. Urbanke, "Efficient encoding of low-density parity-check codes," *IEEE Transactions on Information Theory*, pp. 638–656, feb 2001.

[6] J. Pearl, *Probabilistic Reasoning in Intelligent Systems.* Kaufmann, 1988.

[7] S. M. Aji and R. J. McEliece, "The generalized distributive law," *IEEE Transactions on Information Theory*, pp. 325–343, mar 2000.

[8] T. Richardson, "Multi-edge type ldpc codes." Presented at the workshop in honor of Prof. Bob McEliece's 60th birthday (not in proceedings), California Insitute of Technology, Pasadena, CA, may 2002.

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

[10] S. Y. Chung, *On the Construction of Some Capacity-Approaching Coding Schemes.* PhD thesis, Massachusetts Institute of Technology, sep 2000.

[11] C. Di, D. Proietti, E. Telatar, T. Richardson, and R. Urbanke, "Finite length analysis of low-density parity-check codes." Submitted to IEEE Transactions on Information Theory, 2001.

[12] B. J. Frey, R. Koetter, and A. Vardy, "Skewness and pseudocodewords in iterative decoding,"

[13] T. Richardson, "Multi-edge-type ldpc codes.".

[14] R. Gallager, "Low density parity check codes," *Research Monograph Series*, no. 21, 1963.

[15] S. Litsyn and V. Shevelev, "On ensembles of low-density parity-check codes: Asymptotic distance distributions," vol. 48, no. 4, pp. 887–908, 2002.

[16] C. Di, *Asymptotic and Finite-Length Analysis of Low-Density Parity-Check Codes.* PhD thesis, Ecole Polytechnique Fdrale de Lausanne, 2004.

[17] T. Cover and J. Thomas, *Elements of Information Theory.* Wiley Interscience, 1991.

[18] S. Aji, S. Fogal, R. McEliece, and B. Wang, "Constrained entropy, free energy, and the legendre transform," 2005. Submitted to International Symposium on Information Theory.

[19] J. Nocedal and S. J. Wright, *Numerical Optimization.* Springer-Verlag, 1999.

[20] J. Thorpe, "Low-density parity-check (ldpc) codes constructed from protographs," IPN Progress Report 42-154, JPL, aug 2003.

[21] J. Xu and S. Lin, "A combinatoric superposition method for constructing low density parity check codes," in *IEEE International Symposium on Information Theory*, p. 30, jun 2003.

[22] X. Hu, E. Eleftheriou, and D. Arnold, "Irregular progressive edge-growth (peg) tanner graphs," in *IEEE International Symposium on Information Theory*, p. 480, jun 2002.

[23] T. Richardson, M. Shokrollahi, and R. Urbanke, "Design of capacity-approaching irregular low-density parity-check codes," *IEEE Transactions on Information Theory*, pp. 619–637, feb 2001.

[24] J. Hartmanis and H. Stearns, *Algebraic Structure Theory of Sequential Machines.* Prentice-Hall, 1966.

[25] J. Lee, B. Lee, J. Thorpe, S. Dolinar, and J. Hamkins, "A scalable architecture of a structured ldpc decoder," ISIT, jun 2004.

[26] R. M. Tanner, "A recursive approach to low complexity codes,"

[27]

[28] H. Zhong and T. Zhong, "Design of vlsi implementation-oriented ldpc codes," *IEEE Semiannual Vehicular Technology Conference*, oct 2003.

[29] E. E. X. Hu and D. Arnold, "Progressive edge-growth tanner graphs," in *Global Telecommunications Conference*, vol. 2, pp. 25–29, nov 2001.

[30] J. Thorpe, "Low-complexity approximations to belief propagation for ldpc codes.".

[31] J. Lee and J. Thorpe

[32] S. Chung, D. Forney, T. Richardson, and R. Urbanke, "On the design of low-density parity-check codes within 0.0045 db of the shannon limit," *IEEE Communications Letters*, pp. 58–60, Feb 2001.

[33] Y. Mao and A. Banihashemi, "A heuristic search for good low-density parity-check codes at short block lengths," *IEEE Internationl Conference on Communications*, 2001.

[34] D. MacKay, "Good error correcting codes based on very sparse matrices," vol. 45, pp. 399–431, March 1999.