Message-Passing Decoding on Self-dual $\mathbb{F}_4$-additive Codes

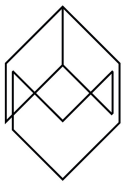
Hannah A. Hansen

Master's thesis in Software Engineering at

Department of Computing, Mathematics and Physics,
Bergen University College

Department of Informatics,
University of Bergen

June 2016



HØGSKOLEN
I BERGEN

Abstract

In this thesis we work with message-passing decoding on self-dual $\mathbb{F}_4$-additive codes. We present and disprove the a decoding scheme proposed by Parker et al. in an unpublished manuscript [17]. We refer to this scheme as Embedded Factor Graph decoding. The method of Discriminative decoding is developed and described in this work as a replacement for the embedded decoding. It is shown to be exact for trees. We implement a simulation tool for simulating message-passing decoding. The tool can simulate both the embedded decoding and the discriminative decoding. Some results from executing simulations on trees and graph with cycles are presented.

Acknowledgements

I wish to thank my supervisors, Pål Ellingsen, Constanza S. Riera, and Matthew G. Parker for their advice, guidance, patience, and encouragement. I gratefully acknowledge the love and encouragement I received from my family and friends. To Mom and Dad for always taking the time to listening and encourage. Your love and support means the world. To my siblings, Chris and Leah for being so kind and awesome. To Pål for all the support and understanding. To the family Grønås Drange; Kari, Helge, Jonas, Elise, Lars, and Agnes, (and Hans) for all the laughter. To my friends and computer science companions; Maja, Erik, Kari, Truls, Piotr, Eivind, Anna, Mathias, Kjetil, Henrik, Pim, Markus, and Ole. You have collectively made the years of my time at the University of Bergen a complete joy.

Please know, my gratitude is more than I can express.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 The Problems of Communication

Alice and Bob are each holding a tin-can connected by a string; it's their tin-can telephone. Any message spoken into a can on the one side carries along the string and into the other. They use it to convey important messages, but unfortunately, this mode of communication is fallible. Transmission interference can at any moment disturb their communication, as the string is vulnerable to external influences, such as cats and wind. It may also have inherent flaws that cannot be fixed nor changed. They need to reliably communicate from afar, but all they have is a tin-can telephone. What should Alice and Bob do to ensure they get their messages across? It is possible for them to increase the energy of their signals by yelling. Another instinctive strategy would be to repeat messages over and over again until certainty is reached in that the other party has received them. But how loud of a signal will the string transmit - is there a limit to volume? And how long are Alice and Bob willing to stand there repeating themselves? Dealing with noise imposed by communication channels is one of the fundamental problems of communication. Decades of research have been dedicated to devise tools and techniques aiding Alice and Bob, and it is safe to say that yelling and repetition are not very efficient strategies.

Forward error correction (FEC) is a technique for limiting errors during transmission. The main idea is to use *error correcting codes* to strengthen transmissions, such that they can withstand noisy conditions. Using one of

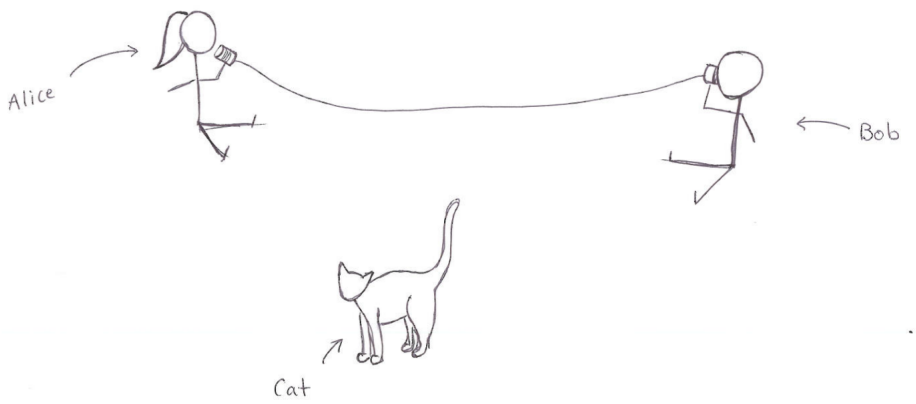


Figure 1.1: Alice and Bob communicating via a tin-can telephone.

them increases the probability of Alice and Bob getting their messages across. All error correcting codes share a common design: *add redundancy*. Take the message you wish to send and add some strategic redundancy, such that the original message can be pieced together in the event of transmission distortion. The acts of adding redundancy and piecing messages together are formally called, *encoding* and *decoding*, respectively.

For the sake of demonstration, let's assume Alice and Bob decide to use an inefficient method of repetition. They encode by enforcing a chosen pattern on a message and decode by performing *majority vote* [14] on said pattern. The pattern they choose to encode with is to repeat each word three times in a row. Alice sends the following transmission.

> "The the the president president president rode rode rode
> his his his bicycle bicycle bicycle into into into a a a tree
> tree tree"

Bob, knowing the encoding pattern, pays attention to each triplet of words and listens for what is most common. For example, should he hear the triplet 'rode rode ode' he would determine that what Alice meant to say was 'rode' and that somewhere along the way noise disturbed the last repetition. For Bob knows that under reasonable assumption, it is less likely that noise has disturbed two repetitions as opposed to one. This type of decoding is called majority vote and is based on the assumption that the majority of received symbols represents the symbol that was sent. Bob receives the following erroneous message.

> "The the the president president president rode rode rode his his his bicycle bicycle bicycle into *onto* into a a a tree tree *me*"

The two errors *'onto'* and *'me'* are easily handled, because the redundancy in this case leaves no room for confusion. The majority of the two corresponding triplets are 'into' and 'tree', respectively. Bob decodes the received transmission to the message 'The president rode his bicycle into a tree'. The redundancy they added to their transmissions guarded their information against noise. The main intuition to take away from this example is that redundancy added by error correcting codes increases the reliability of communication in the presence of noise.

1.2 A Model of Communication

The need for reliability is not exclusively relevant to conversations on the phone. All systems that handle data need to do so accurately; this includes hard drives, CDs, mobile phones, DNA sequencing systems and the deep space network. Despite differences in function and physical manifestation, all of these systems can benefit from using error correcting techniques in the presence of noise. Hence, we abstract away from engineering specific details and employ a general model of communication, which we depict in Figure 1.2. Here we see that information flows from Alice – the source, through the channel and two stages of encoding/decoding, in order to end up with Bob – the sink. *Source coding*, or data compression, is the procedure of transforming the information from Alice into a bit stream. An efficient representation of her message is achieved by removing redundancy from the source, while maintaining the most accurate representation of the information as possible [19]. For our purposes we assume the components of this process to be managed.

Channel coding, is the process of adding redundancy to Alice's bit stream in order to guard against noise [19]. There are two major branches of channel coding schemes, forward-error-correction (FEC) and request-for-repeat (ARQ) [20]. In short, ARQ refers to a collection of schemes for requesting a retransmission once an error has been detected by a code. These schemes are not included in the text, since we are solely concerned with FEC. As illustrated by our introductory example, FEC schemes involve detection and correction of transmission errors by means of error-correcting codes.

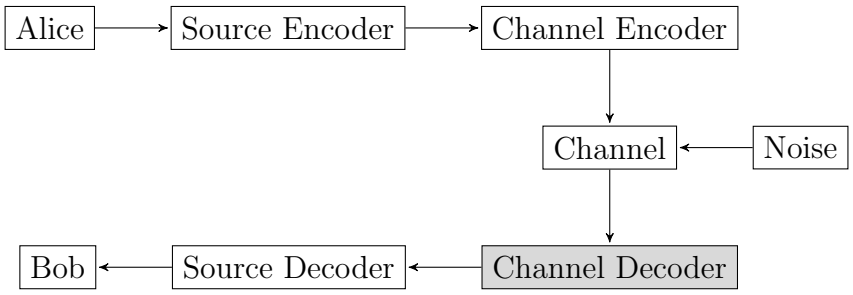


Figure 1.2: Point-to-point communication.

1.3 Graph Codes

The class of codes we work with in this thesis are additive code over $\mathbb{F}_4$, self-dual with respect to the Hermitian inner product. $\mathbb{F}_4$ has the elements $\{0, 1, \omega, \omega^2\}$, where $\omega^2 = \omega + 1$. That is to say, the information Alice and Bob send using these codes is not given by natural language, nor by a classic binary representation, but a *quaternary* representation.

These codes may be interpreted as quantum codes [5], or boolean functions [7]. It is also know that every self-dual $\mathbb{F}_4$-additive code is equivalent to a *graph code* [9], which are codes that have interpretations as graphs. This has also been proven in the context of isotropic systems [4, 7], and quantum stabilizer codes [21]. Definitions and concepts that are relevant for this thesis are given in the following Chapter.

1.4 Iterative Decoding

In this thesis we aim to find an *iterative decoder* [19] for self-dual additive codes over $\mathbb{F}_4^n$ [18]. The general term iterative decoding refers to a class of algorithms that iteratively solve the problem of decoding; the main principle of which is to do so by performing local operations of graphical representations of the code [19]. This technique was first introduced to coding theory by Gallager's work on Low-density Parity-check codes (LDPC) [13].

The particular sub-class of algorithms we work with are called *message passing* algorithms [23]. They are used to solve a broad range of inference, optimization, and constraint satisfaction problems far beyond the context of coding theory. Message passing algorithms operate on structures known as *factor graphs* [23], which are visual representations of the factorization

of a global function [16]. These representations are usually *bipartite graphs* comprised of *factor nodes* and *variable nodes*. The set of variable nodes represent the entire domain of the global function and factor nodes represent factorizations of the global function, such that any edge between a variable node and a factor node indicates that the variable node is a part of the domain of that factorization. This representation facilitates the decomposition of a large complex problem into a finite collection of lesser problems.

In particular, we pursue message passing decoding on self-dual $\mathbb{F}_4$-additive codes using the *sum-product algorithm (SPA)*. This generic message passing algorithm computes the marginals of functions associated with the global function of a factor graph [16]. When performed on a factor graph representing the global function of a code, it serves as a tool in error-detection and error-correction. The marginal values computed provide the foundation for decoding decisions akin to the ones Bob made when interpreting Alice's transmission.

1.5 Structural Overview

In Chapter 2 we give an introduction to the concepts of coding theory, algebra, and combinatorics relevant for this thesis. In Chapter 3 we present and disprove the a decoding scheme proposed by Parker et a. in an unpublished manuscript [17]. In this thesis we refer to this scheme as Embedded Factor Graph decoding. In Chapter 4, we describe the method of Discriminative decoding, which was developed in this work as a replacement for the embedded decoding. At the end of this we provide proof that Discriminative decoding computes the *global marginals* for nodes on any tree. In Chapter 5, we describe the decoding simulation tool developed during the course of this work. It can be used to simulate both decoding methods of Chapter 3 and Chapter 4. In Chapter 6, we provide some general results from executing the Discriminative decoding scheme with the simulation tool of Chapter 5. Finally, in Chapter 7, we conclude our work and describe a direction for further development of our presented method.

Chapter 2

Theoretical Background

Both modern and classical coding theory depend on elements from algebra and combinatorics, such as *fields* and *graphs*. The beginning of this chapter focuses briefly on the algebraic structures and graph theoretical notions central to work of this thesis. In subsequent sections we move on to present our subjects of inquiry; error correcting codes and message passing decoding. The full scope of the subjects in this chapter is not covered. There are many books and papers available to the reader who wishes to explore in-depth explanations. For our purposes we need only to establish the following.

2.1 Finite Fields, Vector Spaces, and Graphs

The codes we work with are *vector spaces* – objects with strong algebraic properties and the decoding methods we use are dependent on graph theory. Here we give two brief but self-contained presentations of the algebraic structures and graph theoretical notions important to the work of this thesis. For more on the topic of algebraic structures the reader is directed to John B. Fraleigh's introductory book on abstract algebra [12]. Readers interested in graphs are directed to the seminal text book on graph theory by Reinhard Diestel [11].

A vector space requires the algebraic ingredients of a *field* and a *group*. We say that a set G closed under a binary operation $*$, is a group $\langle G, * \rangle$, if the following conditions hold:

1. $\forall a, b, c \in G : (a * b) * c = a * (b * c)$ (Associativity)
2. $\exists e \in G \; \forall a \in G : e * x = x * e = x$ (Identity element)
3. $\forall a \in G \; \exists a' \in G : a * a' = a' * a = e$ (Inverse)

A group $\langle G', * \rangle$ is a subgroup of $\langle G, * \rangle$ if $G' \subseteq G$ and G' forms a group under $*$. If $\forall a, b \in G : a * b = b * a$, i.e. G is commutative, then we say that $\langle G, * \rangle$ is *abelian*. The notion of a commutative group provides the grounds for defining a field. We say that a field, $\langle F, +, * \rangle$, is a set F under the operations *addition* and *multiplication*, such that the following holds.

1. F is a commutative group under $+$, with identity element 0. (Commutativity under +)
2. $F \setminus \{0\}$ is a commutative group under $*$. (Commutativity under *)
3. $\forall a, b, c \in F : a * (b + c) = a * b + a * c$ (Distributivity)

The field $\langle F, +, * \rangle$ is defined over two operations, and as such, we denote the *additive identity* of F as 0 and the *multiplicative identity* by 1. There are many interesting and useful properties to both fields and groups, but for now we are only interested in them as means to understand vector spaces, which we shall now define.

Definition 2.1.1. *Given a field $\mathbb{F} = \langle F, +, * \rangle$, abelian with respect to $\mathbb{V} = \langle V, + \rangle$, and a multiplicative operation $*$ between $\mathbb{F}$ and $\mathbb{V}$, we say that $\mathbb{V}$ is a vector space over $\mathbb{F}$ if $\mathbb{V}$ satisfies the following axioms.*

1. $\forall a \in \mathbb{F} \; \forall v \in \mathbb{V} : a * v \in \mathbb{V}$
2. $\forall a, b \in \mathbb{F} \; \forall v \in \mathbb{V} : (a * b) * v = a * (b * v)$
3. $\forall a \in \mathbb{F} \; \forall v, u \in \mathbb{V} : a * (u + v) = a * u + a * v$
4. $\forall a, b \in \mathbb{F} \; \forall v \in \mathbb{V} : (a + b) * v = a * v + b * v$
5. $\forall v \in \mathbb{V} : 1 * v = v * 1 = v$, *where* $1 \in \mathbb{F}$

Any element $a \in \mathbb{F}$ with respect to a vector space is a referred to as a *scalar* and may be combined with any vector $v \in \mathbb{V}$ by multiplication, as seen in Item 1. of Definition 2.1.1. This is called *scalar multiplication*. Furthermore, we call the addition on $\mathbb{V}$, $+$, *vector addition*. Given the vectors $v_1, v_2, ..., v_k$ and the scalars $a_1, a_2, ..., a_k$, we say that (2.1) is a *linear combination*.

$$a_1 v_1 + a_2 v_2 + ... + a_k v_k \tag{2.1}$$

The sum of two linear combinations is also a linear combination. The same holds for the scalar multiplication of a linear combination. We say that a

set of vectors $v_1, v_2, ..., v_k$ are *linearly independent* if for all scalars $a_j \in \mathbb{F}$ $a_1 v_a + a_2 v_2 + ... + a_k v_k = 0$ implies that $a_1 = a_2 = ... = a_k = 0$ [20].

Definition 2.1.2. *$\mathbb{F}$ is a field and $\mathbb{V} = \langle V, + \rangle$ is a vector space over $\mathbb{F}$, and $S \subseteq V$. If $\mathbb{S} = \langle S, + \rangle$ satisfies the conditions of Definition 2.1.1, then we say that $\mathbb{S}$ is a* subspace *of $\mathbb{V}$.*

A graph is a pair of sets $G = (V, E)$ such that $E \subseteq V \times V$. The elements of V are called *nodes* (or *vertices*, or *points*), and the elements of E are called *edges*. Generally, graphs are depicted as dots with lines between them. The dots represent nodes and the lines represent edges. The number of nodes in G is called the *order* and is denoted $|G|$, or $|V|$. We denote the number of edges in G by $|E|$. Both the order and $|E|$ may in principle be finite, infinite, or even uncountably infinite. We only work with graphs of a finite order, therefore G denotes a finite graph, unless otherwise is stated.

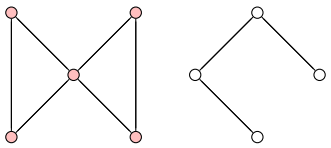


Figure 2.1: A butterfly graph, $K_1 \bowtie 2P_2$, and a tree.

Any two nodes connected by an edge are *neighbors*. For any $n \in V$, we denote its set of neighbors by $\mathcal{N}_n$. The *degree* of a node $n \in V$, $deg(n)$, is the number of edges incident on n. By the definition of a graph provided above we have that $deg(n) = |\mathcal{N}_n|$ for all $n \in V$[1]. A *path* is a non-empty graph $P = (V, E)$ such that $V = \{n_0, n_1, ..., n_k\}$ and $E = \{(n_0, n_1), (n_1, n_2), ..., (n_{k-1}, n_k)\}$, where all n_i are distinct. If the edge (n_0, n_k) also exists then P is a *cycle*. The *length* of any path or cycle is equal to the number of edges.

Let $G = (V, E)$ and $G' = (V', E')$ be graphs. G' is a *subgraph* of G, if $V' \subseteq V$ and $E' \subseteq E$. Furthermore, we say that G is *acyclic*, or a *tree*, if it does not contain a cyclic subgraph. The butterfly graph in Figure 2.1 has two cycles. G is *complete* if there exists an edge between every pair of nodes $n_i, n_j \in V$. We call complete graphs and subgraphs *cliques*. Lastly, G is said to be *bipartite* if V can be partitioned into two sets, V' and V'', such that for all $(n_i, n_j) \in E$ we have that $n_i \in V'$ and $n_j \in V''$, or vice versa.

[1]Multigraphs allow *loops*, which are edges that begin and end in the same node, and they allow several edges between two nodes. [11]

2.2 Block Codes

Block codes are a class of error-correcting codes that add redundancy to *blocks* of information at a time. An encoding function maps k-length messages to n-length codewords. We define a block code, C, to be the set of n-length codewords, i.e. the codomain of encoding function. A *linear code*, denoted $[n, k]$, is a code of length n and rank k over a field $\mathbb{F}_q$, such that it is a linear subspace of the vector space $\mathbb{F}_q^n$ [14]. Linear codes have the property that any linear combination of any two codewords yields another codeword [15]. Any linear code may be expressed by means of two matrices; the generator matrix and the parity-check matrix. The $k \times n$ generator matrix, G, is a basis for the vector space of codewords, and may consist of any set of k linearly independent codewords. It defines the encoding function such that, for any message $\mathbf{m}$ of length k, we have by the multiplication, $\mathbf{m}G = c$, a codeword of length n. The parity-check matrix, H, defines an indicator function for determining code membership. It is a $(n-k) \times n$ parity-check matrix consisting of any set of $n-k$ linearly independent row vectors of the *dual code* of $\mathcal{C}$, such that for all $c \in \mathcal{C}$, we have that $cH^\top = 0$. Generally, there are several potential generator matrices and parity-check matrices [14].

Definition 2.2.1 (Dual code)**.** *Given a $[n, k]$ code C spanned by the $k \times n$ generator matrix G, we have that the code spanned by the $(n-k) \times n$ parity-check matrix H is the generator matrix of the $[n, n-k]$ code $C^\perp$, dual to C [14].*

Definition 2.2.2 (Self-orthogonal code [14])**.** *A code C is self-orthogonal if $C \subseteq C^\perp$.*

Definition 2.2.3 (Self-dual code [14])**.** *A code C is self-dual if $C = C^\perp$.*

We define the *Hamming distance* between two codewords to be the total number of digits in which they differ [15]. The *minimum distance*, or simply *distance*, d, of a code C is said to be the least number of digits with which any two codewords can differ. A code of minimum distance d, denoted $[n, k, d]$, can detect $d-1$ errors and correct $(d-1)/2$ errors. The terminology of Hamming distance stems from the work of Richard W. Hamming, who in the late 40s developed error correcting codes out of spite for a relay computer he had access to only on weekends [22].

> "Two weekends in a row I came in and found that all my stuff had been dumped and nothing was done.. And so I said, 'Damn it, if the machine can detect an error, why can't it locate the position of the error and correct it?" [22, p.vii]

Example 2.2.1. *For the* $[7, 4, 3]$ Hamming code*, we have the following generator matrix,* G*, and parity-check matrix,* H*.*

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \quad H = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

2.2.1 Graph Codes

Given a vector space $\mathbb{F}_4^n$ over $\mathbb{F}_4 = \{0, 1, \omega, \omega^2\}$, where $\omega^2 = \omega + 1$, we define an additive code $\mathcal{C}$ over $\mathbb{F}_4^n$, as an additive subgroup of $\mathbb{F}_4^n$ for some $0 \leq k \leq 2n$, such that $|\mathcal{C}| = 2^k$ [9]. $\mathcal{C}$ may be defined by a $k \times n$ generator matrix G, whose rows *additively* span $\mathcal{C}$.

The Hermitian inner product [9] of two vectors, $u, v \in \mathbb{F}_4$, $u = (u_0, u_1, ..., u_n)$ and $v = (v_0, v_1, ..., v_n)$ is given by:

$$u \star v = \sum_{i=0}^{n} u_i v_i^2 + u_i^2 v_i (mod\ 2) \tag{2.2}$$

The dual code $\mathcal{C}^\perp$ of $\mathcal{C}$, with respect to the Hermitian inner product, is $\mathcal{C}^\perp = \{u \in \mathbb{F}_4^n | u \star c = 0, forall c \in \mathcal{C}\}$. If $\mathcal{C} = \mathcal{C}^\perp$, then $\mathcal{C}$ is *self-dual* and we call $\mathcal{C}$ a self-dual $\mathbb{F}_4$-additive code with respect to the Hermitian inner product. Two codes are considered to be equivalent if one can be obtained from the other by some permutation of vector coordinates, or equivalently, a permutation of the columns of a generator matrix. Two codes $\mathcal{C}$ and $\mathcal{C}'$ are also equivalent if a generator matrix G of $\mathcal{C}$ can be obtained from a generator matrix G' of $\mathcal{C}'$, via an invertible linear transformation $G' = L \times G$.

Definition 2.2.4 (Graph Code)**.** *A graph code is an additive code over* $\mathbb{F}_4$ *that has a generator matrix of the form* $G = A + \omega I$*, where* I *is the identity matrix and* A *is the adjacency matrix of a simple graph. [9]*

Danielsen and Parker showed [9] that all self-dual $\mathbb{F}_4$-additive codes with respect to the Hermitian inner product are equivalent to a graph code. We occasionally refer to these codes as *graph codes*, or *self-dual* $\mathbb{F}_4$*-additive codes*, instead of referring to their full description.

2.3 The Sum-Product Algorithm

The sum-product algorithm (SPA) is a generic message-passing algorithm for performing optimization, inference, and constraint satisfaction on graphical models, such as *Bayesian networks*, *Trellis graphs*, *Markov models*, *factor graphs* [16, 23, 19]. It is a generalization of several algorithms; the Forward/Backward algorithm, the Viterbi algorithm, Belief Propagation on Bayesian networks, among others.

At the receiving end of a transmission the algorithm serves as a tool for error-detection and error-correction. The execution of SPA in our context is dependent upon the indicator function defined by the parity-check matrix H and soft information provided by the channel. With H we construct a factor graph representing the constraints of the code and by performing SPA on it we compute the marginals of each bit of the codeword, which are used to make decisions about the state of each transmitted bit. Given an acyclic factor graph, the marginals computed by the SPA are exact, otherwise it computes approximates of the marginals [16, 20].

Assume $\mathcal{C}$ is a code of length n defined over $\mathbb{F}_4$. We describe the decoding scenario using the sum-product algorithm as follows. Alice sends a transmission $c \in \mathcal{C}$ over the channel. Bob receives the vector $r \in \mathbb{F}_4^n$, which may or may not be a codeword of $\mathcal{C}$. Additionally, he receives n vectors of length four containing *soft information* from the channel. Each length-4 vector contains probability values representing the beliefs regarding the state of a bit of in the originally transmitted message x, given the observed vector r. This information is the basis for our error-detection and error-correction process. We denote the channel information for each r_i of the received vector by $s_i = (e_i, f_i, g_i, h_i)$, such that:

$$
\begin{aligned}
s_0 &= P(x_i = 0 |\ r_i) \\
s_1 &= P(x_i = 1 |\ r_i) \\
s_2 &= P(x_i = \omega |\ r_i) \\
s_3 &= P(x_i = \omega^2 |\ r_i)
\end{aligned}
$$

Message-passing begins after initial beliefs have been provided. However, before examining this procedure, we introduce the factor graph and the message calculation procedures performed by each type of node during message-passing.

2.3.1 Factor Graphs

Factor graphs graphically represent the factorization of global functions. They provide a suitable structure upon which to compute the marginals of functions, such as joint mass probability functions, by exploiting the *distributive law* [16, 19]. Consider the function $g(X)$, where $X = \{x_0, x_1, x_2, x_3, x_4\}$, such that:

$$g(X) = f_0(x_0)f_1(x_1)f_2(x_0, x_1, x_2)f_3(x_2, x_3)f_4(x_2, x_4) \tag{2.3}$$

From this factorization of $g(X)$, we can construct a factor graph, G. We let every variable of $x_i \in X$ constitute a variable node, and every factorization f_i constitute a factor node. Moreover, for every $f_i(X_i)$ and every $x_j \in X_i$ there is an edge (f_i, x_j) in G.

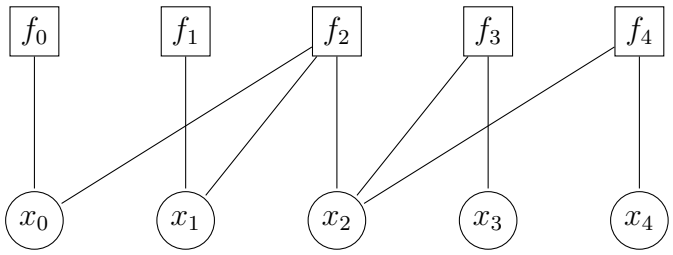


Figure 2.2: Factor graph G of $g(X)$.

Given the parity-check matrix, H, of a code C, we construct a factor graph of C. Every row of the matrix represents a factor node and every column represents a variable node, such that, for all factor nodes f_i and variable nodes x_j, there is an edge between them if and only if the entry a_{ij} of the parity-check matrix is a 1. Using the parity-check matrix of Example 2.2.1 we obtain the factor graph of Figure 2.3.

2.3.2 Message Calculation

Factor nodes. Each factor node f_i represents an indicator function $f_i(X_i)$ defined by a truth table of size $4^{|X_i|}$, where $X_i \subseteq X$. Messages sent by f_i are interpreted as f_i's beliefs concerning the state of the recipient. They are length-4 messages computed entirely on the basis of the information f_i has received from all of its neighbors – excluding the recipient. We define the

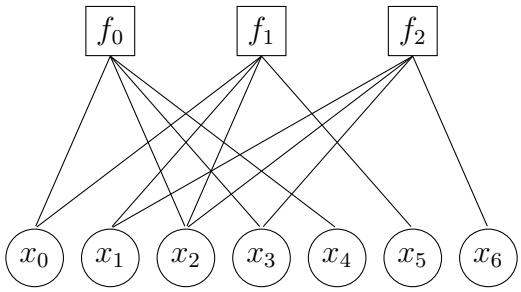


Figure 2.3: Factor graph of function in Example 2.2.1.

message calculation performed by f_i in terms of its truth table. Consider the scenario where $f_0(x_0, x_1, x_2)$ is sending a message to x_0 and has received the following messages from x_1 and x_2, representing their beliefs about their own state.

$$m_{(x_1,f_0)} = \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} \quad m_{(x_2,f_0)} = \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix}$$

$$m_{(f_0,x_0)} = \begin{pmatrix} ae + af + be + bf \\ cg + ch + dg + dh \\ ae + af + be + bf \\ cg + ch + dg + dh \end{pmatrix}$$

For the general case messages from f_i to any $x_j \in \mathcal{N}_i$ are given by (2.4) [16], where the notation $\sum_{\tilde{x}_j}$ denotes the sum of products over all values of $x_j \in \{0, 1, \omega, \omega^2\}$.

$$m_{(f_i,x_j)} = \left(\sum_{\tilde{x}_j} f(X_i) \prod_{x_k \in \mathcal{N}_i \setminus x_j} m_{(x_k,f_i)}) \right) \tag{2.4}$$

Variable Nodes. Messages passed by variable nodes are considered to be beliefs about themselves. For x_j to send a message to f_i, it simply takes the pointwise product of all messages it has received from neighbors other than f_i. This message is given by (2.5) [16]. This message then becomes equivalent of x_j telling f_i what everyone else believes about him.

x_2	x_1	x_0	f_0	
e	a	p_0	1	ae
e	a	p_1	0	ae
e	a	p_ω	1	ae
e	a	p_{ω^2}	0	ae
e	b	p_0	1	be
e	b	p_1	0	be
e	b	p_ω	1	be
e	b	p_{ω^2}	0	be
e	c	p_0	0	ce
e	c	p_1	1	ce
e	c	p_ω	0	ce
e	c	p_{ω^2}	1	ce
e	d	p_0	0	de
e	d	p_1	1	de
e	d	p_ω	0	de
e	d	p_{ω^2}	1	de
f	a	p_0	1	af
f	a	p_1	0	af
f	a	p_ω	1	af
f	a	p_{ω^2}	0	af
f	b	p_0	1	bf
f	b	p_1	0	bf
f	b	p_ω	1	bf
f	b	p_{ω^2}	0	bf
f	c	p_0	0	cf
f	c	p_1	1	cf
f	c	p_ω	0	cf
f	c	p_{ω^2}	1	cf
f	d	p_0	0	df
f	d	p_1	1	df
f	d	p_ω	0	df
f	d	p_{ω^2}	1	df

	f_0	x_0	x_1	x_2
ag	0	p_0	a	g
ag	1	p_1	a	g
ag	0	p_ω	a	g
ag	1	p_{ω^2}	a	g
bg	0	p_0	b	g
bg	1	p_1	b	g
bg	0	p_ω	b	g
bg	1	p_{ω^2}	b	g
cg	1	p_0	c	g
cg	0	p_1	c	g
cg	1	p_ω	c	g
cg	0	p_{ω^2}	c	g
dg	1	p_0	d	g
dg	0	p_1	d	g
dg	1	p_ω	d	g
dg	0	p_{ω^2}	d	g
ah	0	p_0	a	h
ah	1	p_1	a	h
ah	0	p_ω	a	h
ah	1	p_{ω^2}	a	h
bh	0	p_0	b	h
bh	1	p_1	b	h
bh	0	p_ω	b	h
bh	1	p_{ω^2}	b	h
ch	1	p_0	c	h
ch	0	p_1	c	h
ch	1	p_ω	c	h
ch	0	p_{ω^2}	c	h
dh	1	p_0	d	h
dh	0	p_1	d	h
dh	1	p_ω	d	h
dh	0	p_{ω^2}	d	h

Figure 2.4: Truth table of $f_0(x_0, x_1, x_2)$.

$$m_{(x_j,f_i)} = \left(\prod_{f_k \in \mathcal{N}_j \setminus i} m_{(f_k,x_j)} \right) \tag{2.5}$$

2.3.3 Message Scheduling

It is possible to use the SPA to compute single marginals, but we only describe the scenario where each marginal is computed, as we are interested in the marginal of each bit of the received codeword. If the factor graph in question is cycle-free, i.e. is a tree, then the algorithm is exact and the marginal values always converge [23]. Otherwise, the algorithm is not guaranteed to be exact. When cycles are present one runs the risk of getting stuck in an infinite loop, if the values do not converge inside the cycle. [23, 19] Furthermore, cycles may amplify false information by reverberating assumptions. We will explain the message scheduling for trees and graphs with cycles that may be used in this thesis.

Trees. Choose an arbitrary node as the root node - either factor or variable. Begin message-passing at leaf nodes and propagate up towards the root. After the root has received messages from each child, begin message-passing from root through the rest of the tree towards the leaves. When the leaves have received messages, then one can calculate the marginals of each variable by multiplying the incoming and the outgoing messages of a single edge incident on each variable node. [16]

Cycles. Here we exploit an iterative approach with no natural termination, and the general procedure [16] is as follows. Assume initially that an identity message has been sent on every edge in each direction. Thereafter, we iterate over each edge of the graph and pass messages in both directions, until the values have converged, or a decided upon stop criteria is met. We say that *one* iteration of the decoding has been performed once messages have been passed on every edge.

Chapter 3

Embedded Factor Graph Decoding

Developing a decoding scheme for self-dual $\mathbb{F}_4$-additive codes, or graph codes, is motivated by that fact they have been identified as quantum error correction codes [6, 18]. The Embedded Factor Graph decoding scheme was proposed in an unpublished manuscript by Parker et al. [17]. Its strategy is to perform message passing decoding with the sum-product algorithm on a factor graph implicitly represented in the graph of these codes.

The first section of this chapter is dedicated to describing the manner in which factor graphs are represented, or embedded, in graph codes. Followed by a brief account of message scheduling. In the second section of this chapter we show that messages sent by embedded factor nodes are consistently ambiguous — they convey uncertainty with respect to the state of the recipient. We end this chapter by arguing that this approach is does not lead to a decoding decision due to the ambiguity of messages propagated during message passing.

3.1 Factor Graph Embedding

In embedded factor graph decoding we define two structural levels; the graph of the hermitian self-dual $\mathbb{F}_4$-additive codes and the embedded factor graph described by it. Here we give a precise description of the embedded factor graph. Assume a graph code, $\mathcal{C}$, of length n described by the parity check matrix $H = \omega I + A$. From A we have the graph $\mathcal{G} = (V_G, E_G)$, where $|V_G| = n$. Let $\mathcal{FG} = (V_{FG}, E_{FG})$ denote the factor graph of $\mathcal{G}$, such that $|V_{FG}| = 2n$

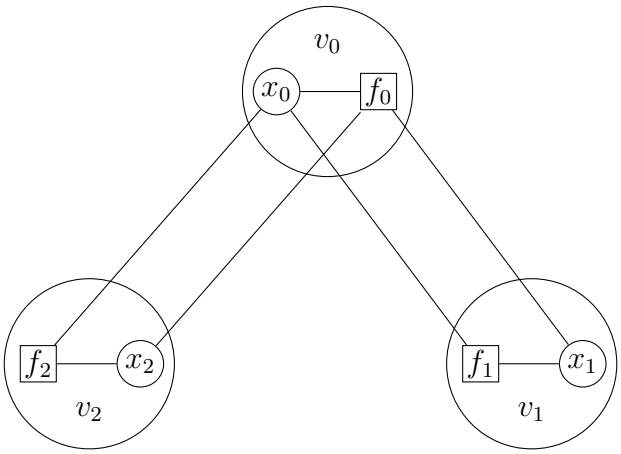


Figure 3.1: The graph $\mathcal{G}$ of $\mathcal{C}$ with the embedded factor graph $\mathcal{FG}$.

and $|E_{FG}| = 2|E_G| + n$.

For each vertex $v_i \in V_G$ we associate two nodes in V_{FG}: a variable node x_i and a factor node f_i. We call them *associates* and say they are embedded into v. The associates are given the same index as the vertex in which they are embedded. In order to distinguish between structural levels, we refer to the vertices of V_G explicitly as *meta nodes*, and the nodes of V_{FG} as variable nodes and factor nodes.

Between each associate of a meta node we draw the edge, $(x_i, f_i) \in E_{FG}$. The rest of their respective neighborhoods are defined by the hermitian constraint associated with their meta node. Meaning, for each edge $(v_i, v_j) \in E_G$, we have two undirected edges (x_i, f_j), $(x_j, f_i) \in E_{FG}$.

Example 3.1.1. *Here we have the parity check matrix, H, of the code C, in graph form. From H we construct the graphs in Figure 3.1.*

$$H = \begin{pmatrix} \omega & 1 & 1 \\ 1 & \omega & 0 \\ 1 & 0 & \omega \end{pmatrix}$$

As a consequence, we have that each f_i is described by the same hermitian constraint as their meta node, i.e. the i-th row of H. Consider the example 3.1.1, here we see that f_1 is described by $(1, w, 0)$. This means that $f_1(x_0, x_1) = 1$ if and only if $(x_0, x_1, 0) \star (1, w, 0) = 0$, see figure 3.1.

x_0	x_1	f_1
0	0	1
0	1	0
0	ω	1
0	ω^2	0
1	0	1
1	1	0
1	ω	1
1	ω^2	0
ω	0	0
ω	1	1
ω	ω	0
ω	ω^2	1
ω^2	0	0
ω^2	1	1
ω^2	ω	0
ω^2	ω^2	1

Table 3.1: The indicator values of f_1 from Example 3.1.1.

3.1.1 Message Scheduling

The embedded decoding scheme involves performing the well known sum-product algorithm on $\mathcal{FG}$, where message scheduling is governed by the edges of $\mathcal{G}$. First, soft information is given to each $x_i \in V_{FG}$ by the channel, thereafter initial messages are passed between all neighbors. Once this setup has been performed, we iterate over edges $(v_i, v_j) \in E_G$ until a given stop criteria is met.

For each iteration we pass messages in both directions between meta nodes. Passing a message from one meta node to another amounts to passing messages on the embedded nodes of $\mathcal{FG}$. This means that when v_i sends a message to v_j, its associates x_i and f_i pass messages between each other and to their neighbors f_i, x_j embedded in v_j, respectively. This scheduling may not be optimal, however, due to the discovery of essential flaws elsewhere in the method we abandoned the scheme before addressing the issue.

3.2 Proof of Ambiguity

In this section we show that the embedded factor graph decoding scheme presented in the previous section is unsound for any $\mathbb{F}_4$-additive code, self-dual with respect to the hermitian inner product. We claim that the scheme cannot compute the marginals of the code bits, as the messages passed by factor nodes are consistently *ambiguous*. As a consequence, message-passing decoding on embedded factor graphs produces a propagation of uncertainties, rather than a propagation of beliefs.

We say that a message is ambiguous if all values are equal, or if there are two values competing to be the strongest belief of the message and two values competing to be the weakest belief of the message. This translates into either total, or near total, uncertainty on behalf of the sender with regards to the state of the recipient. It is our finding that all embedded factor nodes are incapable of sending unambiguous messages. Proof of this is divided into two sections.

First, we identify a universal structure found in factor nodes of $\mathbb{F}_4$-additive codes. This structure manifests as a particular pattern of indicator values in the truth tables of the factor nodes. It is here we find the corruption of the decoding process, as the pattern forces the production of ambiguous messages. Thereafter, we show how the truth table representation of our indicator functions may be replaced by decision diagrams similar to those introduced by Aker [3], which represent boolean functions, where edges are labeled with the potential variable values. Their purpose is to provide a simple procedure for determining the output of a boolean function by examining the values of the of input.

Our diagrams differ in that their purpose is to compute a four-valued message representing the belief a factor node with regards to a recipient. We refer to the diagrams in this thesis as *quaternary decision diagrams*, since they represent indicator functions of a quaternary domain. We end the chapter by proving the ambiguity of all factor node messages.

3.2.1 Factor Node Structure

Consider the factor nodes of our embedded graphs, they are constrained by their respective rows of the parity matrix H $(n \times n)$. In other words, the indicator values of their truth tables are derived from their hermitian constraints. The brute force method of determining these values involves a computation of

size $O(4^n)$, as one would have to go through each n-permutation of $x \in \mathbb{F}_4$ and calculate the hermitian inner product of the permutation and the constraint. The equation (3.2) of Lemma 3.2.1 was discovered in an effort to find a more efficient method of determining indicator values.

f_0	x_0	x_1
1	0	0
0	1	0
1	ω	0
0	ω^2	0
..	..	..
..	..	..
0	ω	ω^2
1	ω^2	ω^2

Figure 3.2: Compressed truth table of $f_0(x_0, x_1)$.

The variables of equation 3.2 refer to the components of the double binary equation (3.1), which expresses any element $x \in \mathbb{F}_4$, in terms of elements of $\mathbb{F}_2$ and ω. In Fig.3.3 one may find a table of the double binary equation and a truth table of f_0 with two neighbors. Consider the third row of the truth table, as an example of (3.2) we see that $a_0 = 0$ and $b_1 = 0$, thus the third entry of the table is 1.

Lemma 3.2.1. *Given the $n \times n$ parity check matrix H of any graph code, we have that the hermitian constraint $r_i \star v = 0$ for the i-th row of H, corresponds to the following condition on the vector $v = ((a_0 \oplus \omega b_0), ..., (a_{n-1} \oplus \omega b_{n-1}))$.*

$$f_i = a_i \oplus \sum_{j \in \mathcal{N}_i} b_j = 0 \tag{3.2}$$

$$x = a \oplus \omega b \tag{3.1}$$

x	a	b
0	0	0
1	1	0
ω	0	1
ω^2	1	1

Figure 3.3: The double binary equation.

Proof. Assume a graph code described by an $n \times n$ parity check matrix H. Let r_i be the i-th row of H and $v = ((a_0 \oplus \omega b_0), ..., (a_{n-1} \oplus \omega b_{n-1}))$. We show here that $r_i \star v = 0$ if and only if $f_i = a_i \oplus \sum_{j \in \mathcal{N}_i} b_j = 0$. Recall the definition of the Hermitian inner product $u \star v = \sum_{i=0}^{n} u_i v_i^2 \oplus u_i^2 v_i$.

$$v_i^2 = (a_i \oplus \omega b_i)^2 = a_i^2 \oplus b_i^2(\omega \oplus 1) = a_i \oplus \omega b_i \oplus b_i = v_i \oplus b_i \tag{3.3}$$

$$\begin{aligned}
r_i \star v &= \sum_{j=0}^{n} r_j v_j^2 \oplus r_j^2 v_j && (3.4)\\
&= \omega v_i^2 \oplus \omega^2 v_i \oplus \sum_{j \in \mathcal{N}_i}^{n} 1 v_j^2 \oplus 1^2 v_j && (3.5)\\
&= \omega v_i^2 \oplus \omega^2 v_i \oplus \sum_{j \in \mathcal{N}_i}^{n} v_j^2 v v_j && \text{Eq.3.3} \quad (3.6)\\
&= \omega v_i^2 \oplus \omega^2 v_i \oplus \sum_{j \in \mathcal{N}_i}^{n} (v_j \oplus b_j) \oplus v_j && (3.7)\\
&= \omega v_i^2 \oplus \omega^2 v_i \oplus \sum_{j \in \mathcal{N}_i}^{n} 2 v_j \oplus b_j && (3.8)\\
&= \omega v_i^2 \oplus \omega^2 v_i \oplus \sum_{j \in \mathcal{N}_i}^{n} b_j && (3.9)\\
&= \omega(v_i \oplus b_i) \oplus (\omega \oplus 1) v_i \oplus \sum_{j \in \mathcal{N}_i}^{n} b_j && \text{Eq.3.3} \quad (3.10)\\
&= \omega b_i \oplus v_i \oplus \sum_{j \in \mathcal{N}_i}^{n} b_j && (3.11)\\
&= \omega b_i \oplus (a_i \oplus \omega b_i) \oplus \sum_{j \in \mathcal{N}_i}^{n} b_j && (3.12)\\
&= a_i \oplus \sum_{j \in \mathcal{N}_i}^{n} b_j && (3.13)\\
& && (3.14)
\end{aligned}$$

□

Using the indicator value equation (3.2), we want to derive an expression of the indicator vector of any f_i in terms of the size of its neighborhood. Lemma 3.2.2 provides a recursion expressing just that.

Lemma 3.2.2. *Let $iv_i(t)$ denote the indicator vector of the factor node $f_i \in \mathcal{FG}$ with $|N_i| = t$ neighbors, then the values of $iv_i(t)$ are determined by the following recursion.*

$$iv_i(1) = (1, 0, 1, 0)$$
$$iv_i(t) = (iv_i(t-1), iv_i(t-1), \overline{iv}_i(t-1), \overline{iv}_i(t-1)).$$

Proof. We conduct proof by induction on the number of neighbors t. Assume the embedded factor node f_i, let its associate be denoted by x_i.

In the base case, when $t = 1$, it is only possible for the associate of f_i to be a neighbor. Therefore, $iv_i(1) = (1, 0, 1, 0)$. Induction hypothesis – assume it holds for all $t = n$ that

$$iv_i(n) = (iv_i(n), iv_i(n), \overline{iv}_i(n-1), \overline{iv}_i(n-1)).$$

Adding a neighbor to f_i corresponds to extending the truth table of f_i by adding a column for the $n+1$ neighbor and repeating the old truth table for each value of $\{0, 1, \omega, \omega^2\}$ associated with the new neighbor $n+1$. Let b_z denote the b value when $x_{n+1} = z$, where $z \in \mathbb{F}_4$. Given equation (3.2) we have that:

$$\begin{aligned} &iv_i(n+1) \\ &= (b_0 + iv_i(n), b_1 + iv_i(n), b_\omega + iv_i(n), b_{\omega^2} + iv_i(n)) \\ &= (0 + iv_i(n), 0 + iv_i(n), 1 + iv_i(n), 1 + iv_i(n)) \\ &= (iv_i(n), iv_i(n), \overline{iv}_i(n), \overline{iv}_i(n) \end{aligned}$$

□

We observe from this pattern that for any neighbor x_j of f_i there are pairs of values of x_j that coincide with respect to f_i's indicator values. For example consider the associate of f_0 in Fig.3.3, in the first four rows the pairs $(0, \omega)$ and $(1, \omega^2)$ coincide. This holds true for all rows of the table, and would hold true even if we gave f_0 additional neighbors. The other neighbors of f_0 coincide with respect to the pairs $(0, 1)$ and (ω, ω^2). We now define the concept of *strands* to denote the values which coincide for a given variable.

Definition 3.2.3. *The strands of any variable x_j with respect to f_i are the two possible pairs of values, $s_{j,1}$ and $s_{j,2}$, that coincide according to the indicator values of f_i. These may be either twisted or separated:*

Twisted: $s_j^0 = \{0, \omega\}$ *and* $s_j^1 = \{1, \omega^2\}$

Separated: $s_j^0 = \{0, 1\}$ *and* $s_j^1 = \{\omega, \omega^2\}$

Lemma 3.2.4. *For any f_i, the strands of its associate, x_i are twisted. The strands of any other neighbor are separated.*

Proof. Consider the equation (3.2). We see that for any associate x_i the value b_i from its double binary are never present, therefore a_i will dictate correspondence with respect to the indicator vector of f_i. By examining the double binary table from figure 3.3 we see that $a_i = 0 \iff (x_i = 0 \vee x_i = \omega)$ and $a_i = 1 \iff (x_i = 1 \vee x_i = \omega^2)$.

As for any other variable, we see that only the b_j's play a role in the equation. By the same argument as for the associate we have that $b_j = 0 \iff (x_j = 1 \vee x_j = 0)$ and $b_j = 1 \iff (x_j = \omega \vee x_j = \omega^2)$. □

3.2.2 Quaternary Decision Diagrams

The quaternary decision diagram of an embedded factor node f_i is a directed acyclic graph representing the set of input that satisfy f_i's hermitian constraint, upon which messages from f_i to x_j may be computed. Here we explain the construction of quaternary decision diagrams and demonstrate their relationship to the truth tables of embedded factor nodes. In the following section, we describe how they can be used in message calculation and show the ambiguity of all factor node messages.

Given an indicator function f_i with a neighborhood of size n, we have its quaternary decision diagram $B(f_i) = K_{1,2,2,\ldots,2,1}$, a complete $n+1$-partite graph directed from right to left, see Figure 3.4. The sink, denoted 1, represents the $1s$ from f_i's indicator vector. All other partitions represent neighbors of f_i. The source represents the neighbor of the last column in f_i's truth table, and the partition to the right of 1 is reserved for the associate x_i.

We describe each partition, a part from the sink, and their outgoing edges in terms of the indicator vector recursion of Lemma 3.2.2. For any x_k positioned at column t of f_i's truth table we have the following:

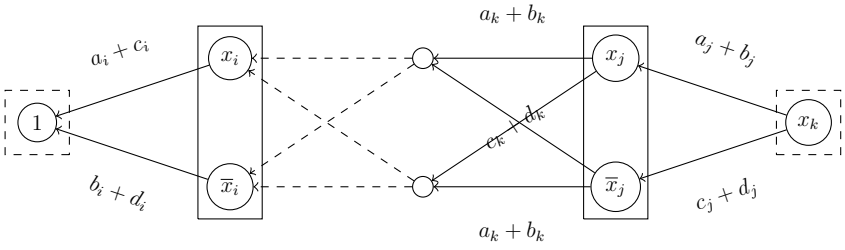


Figure 3.4: A generic quaternary decision diagram.

Case 1: If there is another x_m at position $t+1$, then we let x_k's partition contain two nodes; x_k and $\overline{x}_k$. The node x_k represents the portions of $iv_i(t+1)$ equal to $iv_i(t)$. The node $\overline{x}_k$ represents the portions of $iv_i(t+1)$ equal to $\overline{iv}_i(t)$.

Let x_j denote the neighbor associated with $t-1$ partition containing the nodes x_j and $\overline{x}_j$. From x_k we draw the direct weighted edges (x_k, x_j, s_k^0) and $(x_k, \overline{x}_j, s_k^1)$. From $\overline{x}_k$ we draw the edges $(\overline{x}_k, x_j, s_k^1)$ and $(\overline{x}_k, \overline{x}_j, s_k^0)$. Where, with a slight abuse of notation, we let s_k^0 and s_k^1 refer to the sum of values from x_k's message pertaining to the elements of x_k's strands.

The edges from x_k and their weights are justified by the indicator vector recursion $iv_i(t) = (iv_i(t-1), iv_i(t-1), \overline{iv}_i(t-1), \overline{iv}_i(t-1)$. We know for the sub-portions, $iv_i(t)$, of f_i's indicator vector containing $iv_i(t-1)$ that $f_i = 1$ implies that $x_k \in \{0, 1\} = s_k^0$, therefore we draw the edge (x_k, x_j, s_k^0). For the portions of f_i's indicator vector containing $\overline{iv}_i(t-1)$ we know that $f_i = 1$ implies that $x_k \in \{\omega, \omega^2\} = s_k^1$, hence the edge $(x_k, \overline{x}_j, s_k^1)$. Conversely for $\overline{x}_k$, which represents $\overline{iv}_i(t)$ in $iv_i(t+1)$. For the sub-portions, $\overline{iv}_i(t)$, of f_i's indicator vector containing $iv_i(t-1)$ we have that $f_i = 1$ implies $x_k \in \omega, \omega^2 = s_k^1$, hence the edge $(\overline{x}_k, x_j, s_k^1)$. Similarly for the edge $(\overline{x}_k, \overline{x}_j, s_k^0)$.

Case 2: If there is no other x_m at position $t+1$, then x_k is the source and its partition contains only the node denoted x_k. Let x_j denote the neighbor associated with the $t-1$ partition containing the nodes x_j and $\overline{x}_j$. We draw the directed edges (x_k, x_j, s_k^0) and $(x_k, \overline{x}_j, s_k^1)$.

From Truth Table to QDD In order to demonstrate the relationship between quaternary decision diagrams and the truth tables of embedded factor nodes, we construct trees representing the truth tables and show how editing them produces our decision diagrams. These trees follow the same

construction as binary decision diagrams [3], only with a branching of factor four since our variables are quaternary.

There are 4^n leaf nodes representing the indicator values of f_i and $\sum_{k=0}^{n-1} 4^k$ nodes representing the neighbors of f_i. Each level of the tree, except for the leaves, corresponds to a neighbor of f_i. In order to maintain the same indicator values pattern as in the truth tables of our previous section we enforce *one* restriction: let the level of nodes parent to the leaves correspond to the *associate* of the factor node. Edges of the tree are directed and weighted with values from the messages sent to f_i by the parent of the edge. Each path from the root to a leaf corresponds to a row in f_i's truth table.

x_1	x_0	f_0
p_0	p_0	1
p_0	p_1	0
p_0	p_ω	1
p_0	p_{ω^2}	0
p_1	p_0	1
p_1	p_1	0
p_1	p_ω	1
p_1	p_{ω^2}	0
p_ω	p_0	0
p_ω	p_1	1
p_ω	p_ω	0
p_ω	p_{ω^2}	1
p_{ω^2}	p_0	0
p_{ω^2}	p_1	1
p_{ω^2}	p_ω	0
p_{ω^2}	p_{ω^2}	1

Table 3.2: Truth table of a factor node with two neighbors.

To obtain a quaternary decision diagram we prune our tree of unnecessary structure. For our purposes, we only care about paths of the tree that end at a 1, as the other leaves play no role in the message calculation – all leaf nodes labeled 0 are removed. Furthermore, there are duplicate subtrees on each level. The roots of these subtrees are merged together; weights on incoming edges from the same parent are simply added together.

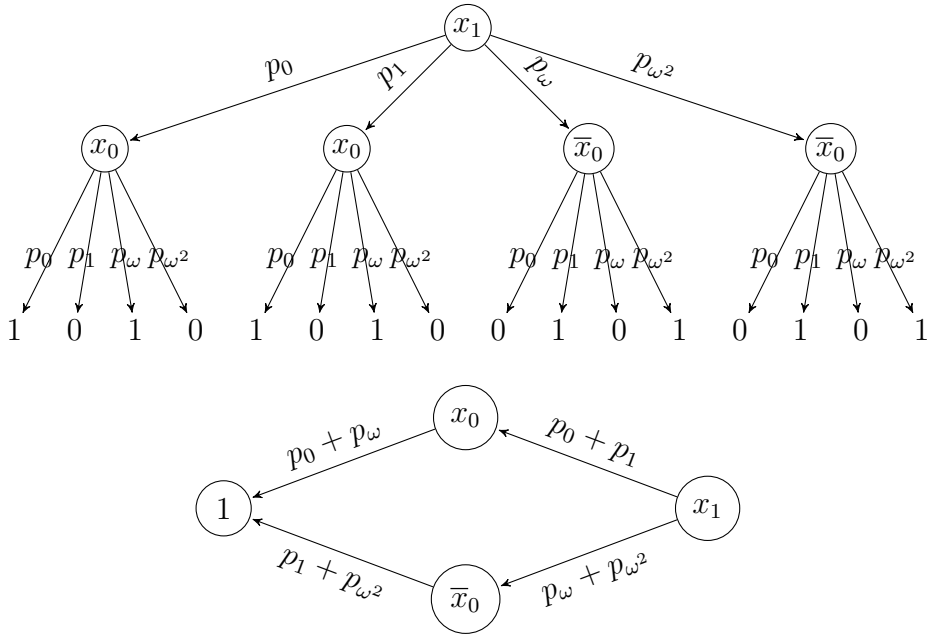


Figure 3.5: Decision tree of Table 3.2 (top) and its corresponding the QDD $B(f_0)$ (bottom).

3.2.3 Mixed Messages

Here we describe message calculation on QDDs, and use them to show how messages sent by embedded factor nodes are always ambiguous. Given a factor node f_i we calculate messages by traversing edited versions of $B(f_i)$. Let $B(f_i)_j^z$, where j is of the neighbor x_j and $z \in \mathbb{F}_4$, denote the QDD where the p_z-edges of x_j are weighted with 1 and all other edges of x_j are weighted with 0. We then define $sp(f_i, j, z)$ to be the sum of all paths from source to sink on $B(f_i)_j^z$, where paths are interpreted as products of the edge weights along them. From this we have that any message from f_i to x_j is given by:

$$m_{(f_i,x_j)} = \begin{pmatrix} sp(f_i, j, 0) \\ sp(f_i, j, 1) \\ sp(f_i, j, \omega) \\ sp(f_i, j, \omega^2) \end{pmatrix} \tag{3.15}$$

Definition 3.2.5. *A message is ambiguous if all values are equal, or if there are two values, $a, b \in \mathbb{R}$, competing to be the strongest belief of the message and two values competing to be the weakest belief of the message.*

v_0	v_1
0	0
1	ω
ω	1
ω^2	ω^2

Table 3.3: Codespace from Example 3.3.1.

Let $m_{(f_i,x_j)}$ be an ambiguous message. We say that $m_{(f_i,x_j)}$ is *twisted*, if there are two values $a, b \in \mathbb{R}$ such that $a \neq b$ and $m_{(f_i,x_j)} = (a, b, a, b)$. If $m_{(f_i,x_j)} = (a, a, b, b)$, then $m_{(f_i,x_j)}$ is instead called *separated*.

Theorem 3.2.6. *Any message from f_i to its associate x_i is twisted. Messages from f_i to x_j, where $j \neq i$, are separated.*

Proof. For f_i's associate x_i we know that $sp(f_i, i, 0) = sp(f_i, i, \omega)$, because the 0-edges and ω-edges of x_i are the same. Similarly, $sp(f_i, i, 1) = sp(f_i, i, \omega^2)$, because 1-edges and ω^2-edges are the same.

For any other neighbor of $x_{j\neq i}$, we have that $sp(f_i, j, 0) = sp(f_i, j, 1)$, because their strands are separated, i.e. their 0-edges are the same as their 1-edges. Likewise, their ω-edges and ω^2-edges are the same, therefore $sp(f_i, j, \omega) = sp(f_i, j, \omega^2)$. □

3.3 Decoding Example

Here we provide an example of marginals calculated using the Embedded decoding scheme. Using the code C of Example 3.3.1, we show that the scheme does not compute the global marginal for v_0.

Example 3.3.1. *Here we have the parity check matrix, H, of the code C, in graph form. From H we construct the graphs in Figure 3.6. Below we also have a table of the entire code space of C.*

$$H = \begin{pmatrix} \omega & 1 \\ 1 & \omega \end{pmatrix}$$

Let the vectors $s_0 = (a, b, c, d)$ and $s_1 = (e, f, g, h)$ be the soft information the embedded nodes x_0 and x_1 receive from the channel, respectively. We

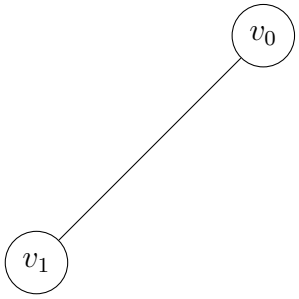


Figure 3.6: K_2

know from the previous sections that two messages are sent to x_0; $m_{(f_0,x_0)}$ and $m_{(f_1,x_0)}$.

$$m_{(f_0,x_0)} = \begin{pmatrix} e+f \\ g+h \\ e+f \\ g+h \end{pmatrix} \quad m_{(f_1,x_0)} = \begin{pmatrix} e+g \\ e+g \\ f+h \\ f+h \end{pmatrix} \tag{3.16}$$

The results from decoding using the embedded scheme is given in (3.17) by $emb(x_0)$. Next to it one may find the global marginals, $g(x_0)$, as given by the table of Example 3.3.1. These two vectors are not equal; the Embedded decoding scheme did not compute the marginals from the global function.

$$emb(x_0) = \begin{pmatrix} a(e+f)(e+g) \\ b(g+h)(e+g) \\ c(e+f)(f+h) \\ d(g+h)(f+h) \end{pmatrix} \quad g(x_0) = \begin{pmatrix} ae \\ bg \\ cf \\ dh \end{pmatrix} \tag{3.17}$$

When simulations of the embedded decoding where executed using the simulation tool of Chapter 5., we found similar results on star graphs with n nodes, where $n \in \{3, ..., 7\}$. Justified by these simulations and the proof of ambiguity, we discarded the Embedded approach and moved on to the decoding scheme of Chapter 4 – Discriminative Decoding.

Chapter 4

Discriminative Decoding

This chapter introduces the *Discriminative Decoding* scheme, for which we replaced the Embedded Factor Graph scheme. This scheme is also a message passing procedure, however it differs in graph structure and with respect to local operations. We begin with a motivation and a general description of Discriminative Decoding and move on to describe the local operations in detail. We end this Chapter by showing that Discriminative decoding is computes the global marginals for any node on any tree, thus proving it to be an instance of the sum-product algorithm described in Chapter 2.

4.1 Introduction

In Discriminative decoding, we move away from the embedded factor graph structure and rely entirely upon the graph of the code as described directly from the parity check matrix in graph form. Our nodes are simple, and their relationships are defined by the matrix. However, during message passing there is a differentiation between messages from leaf nodes and internal nodes, hence the name *Discriminative Decoding*. This necessitates that nodes have certain knowledge about their own neighborhoods. We argue that this does not compromise the locality of our scheme, as the only knowledge necessary is whether or not a neighbor is a leaf. An affirmation of whether or note a node is itself a leaf may easily be reached by any node who has only one neighbor. Passing along this positive information provides any internal node with a confirmation of which neighbors are leaves. Any who have not given such a confirmation message are treated as not-leaves, i.e. internal nodes.

There are in total four vector products in use during message passing – three of which we introduce in this work. These products are vital with respect to the local operations performed by the nodes of our graph. We introduce them in the next section, but first lets recapitulate the description of our graph codes and scheduling of messages during message passing decoding on them.

Graphs and Message Scheduling A graph code of length n is a self-dual $\mathbb{F}_4$-additive code with an $n \times n$ parity check matrix of the form $H = \Gamma + \omega I$, where Γ is an adjacency matrix and I is the identity matrix [7]. The adjacency matrix Γ represents a simple, undirected graph with all 0s along its diagonal [7].

If Γ is the matrix of a graph with cycles, then we iterate over the edges of the graph and pass messages in each direction. One iteration of the decoding is complete once each edge has had messages passed in each direction. If Γ is tree, then we pass messages in two phases. First we pass messages from the leaves to the root, where each node sends a message to its parent based on the messages it has received from its children. Thereafter, messages are passed from the root down towards the leaves. In this phase nodes send messages to all of their children.

4.2 Vector Products

We use four vector operations in Discriminative decoding. The first being the *pointwise product*, denoted '$\cdot$', where the i-th entry of the product vector is the product of the i-th entry of both factors, see Definition 4.2.1. The remaining operations *divided straight-straight (dSS)*, *divided straight-cross (dSX)*, and *twisted straight-cross (tSX)* are given in Definitions 4.2.2, 4.2.3, and 4.2.4, respectively. Each operation takes two vectors of length four and produces a vector of length 4 where each entry consists of a sum of two products.

Definition 4.2.1. *The pointwise product,* $\cdot(u, v) : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$*, of the vectors* $u = (u_0, u_1, ..., u_n)$ *and* $v = (v_0, v_1, ..., v_n)$ *is given by:*

$$\cdot(u, v) = \begin{pmatrix} u_0 v_0 \\ u_1 v_1 \\ ... \\ u_n v_n \end{pmatrix}$$

Definition 4.2.2. *The function* divided straight-straight *is defined by* $dSS : \mathbb{R}^4 \times \mathbb{R}^4 \to \mathbb{R}^4$ *such that given the vectors* $u, v \in \mathbb{R}^4$ *we have:*

$$dSS(u,v) = \begin{pmatrix} v_0u_0 + v_1u_1 \\ v_2u_2 + v_3u_3 \\ v_0u_1 + v_1u_0 \\ v_2u_3 + v_3u_2 \end{pmatrix}$$

Definition 4.2.3. *The function* divided straight-cross *is defined by* $dSX : \mathbb{R}^4 \times \mathbb{R}^4 \to \mathbb{R}^4$ *such that given the vectors* $u, v \in \mathbb{R}^4$ *we have:*

$$dSX(u,v) = \begin{pmatrix} v_0u_0 + v_1u_1 \\ v_0u_1 + v_1u_0 \\ v_2u_2 + v_3u_3 \\ v_2u_3 + v_3u_2 \end{pmatrix}$$

Definition 4.2.4. *The function* twisted straight-cross *is defined by* $tSX : \mathbb{R}^4 \times \mathbb{R}^4 \to \mathbb{R}^4$ *such that given the vectors* $u, v \in \mathbb{R}^4$ *we have:*

$$tSX(u,v) = \begin{pmatrix} v_0u_0 + v_2u_1 \\ v_0u_1 + v_2u_0 \\ v_1u_2 + v_3u_3 \\ v_1u_3 + v_3u_2 \end{pmatrix}$$

The terms divided, twisted, straight-straight, and straight-cross describe the interaction between the two input vectors. Figures 4.2, 4.1, 4.3 provide some illustration to these concepts. Divided and twisted describe which values of the input vectors are multiplied and summed to form an entry in the product vector. Straight-straight and straight-cross describe the ordering of the entries.

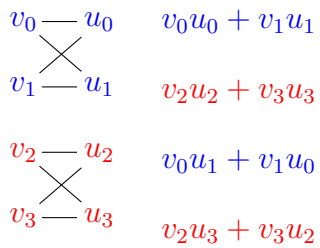


Figure 4.1: Vector product $dSS(u,v)$

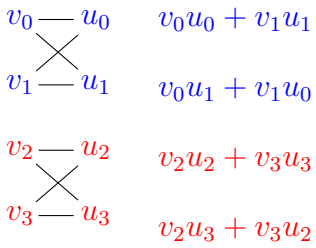


Figure 4.2: Vector product $dSX(u, v)$

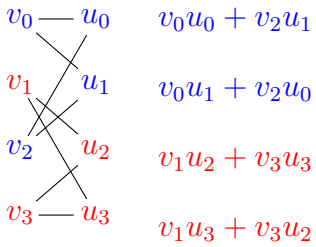


Figure 4.3: Vector product $tSX(u, v)$

4.3 Local Operations

The key aspect of discriminative decoding is the difference in behavior among leaves and internal nodes when passing a message or computing marginals. Leaf nodes perform no computations when passing a message. Their only neighbor is their parent, therefore they merely pass along their soft information. Internal nodes, on the other hand, discriminate between neighbors who are leaves and neighbors who are internal nodes. This discrimination effects the computation of both their marginals and messages. In this section we focus mainly on the details of message and marginal computations of internal nodes. The following paragraph is devoted to explaining the corresponding operations for leaves.

Leaves Whenever a leaf is elicited to pass a message, it passes along its soft information to its parent. This is the only neighbor it can send a message to and the soft information is the only belief it has to send. When determining their own marginals, they take the pointwise product of their soft information and the belief they have received from their parent.

4.3.1 Internal nodes

The foundation of message and marginal computation for internal nodes lies in the procedure of computing two products. The product of beliefs sent by leaves, and the product of beliefs sent by internal nodes, the *leaf-product* and the *internal-product*, respectively. Computing these products for the sake of a message or a marginal is fundamentally the same, except for the fact that when computing a message we disregard any beliefs sent by the recipient. This is so one avoids false amplification of a preexisting belief belonging to the recipient. The Algorithms 3, 4, 5, and 6 show how the leaf-product and internal-product are computed by recursive application of vectors products defined above. We move on to the computation of messages and marginals.

Passing a Message The computation of a message to a neighbor can be divided into two stages. First the node in question computes the leaf-product and the internal-product, while in the process disregarding any messages sent by the recipient. Thereafter there are two options; if the node in question does not have any internal neighbors, or if the recipient is its only internal-neighbor, then it passes along the dSS-product of its own soft information and the leaf-product. Otherwise, the message is calculated in two steps. First, take the dSX-product the soft information and the leaf-product. Afterwards, take the dSS-product of the previous result and the internal-product. Pseudocode for this procedure may be found in Algorithm 1.

Computing Marginals When computing its own marginals, an internal node will first produce the leaf-product and the internal-product using all the messages it has received. Once that is completed, there are three options with respect to completing the computation of the marginals. The choice of which depends upon the state of the node's neighborhood. For, if the node in question is in the middle of a star graph, i.e. it has no internal neighbors, then its estimated marginal are given by the pointwise product of its soft information and the leaf-product. Similarly, if the node in question is not parent to any leaves, then its estimated marginals are given by the pointwise product of its soft information and the internal-product. Otherwise, if the node's neighborhood consists of both leaves and internal nodes, then its estimated marginals are given by the pointwise product of its soft information and the result of performing dSX(leaf-product, internal-product). Algorithm 2 provides pseudocode for this procedure.

Algorithm 1 Pass a message.

Let s be the soft information of the sender.

```
1: procedure passMessage(Node recipient)
2:     leafProduct ← leafProduct(recipient)
3:     internalProduct ← internalProduct(recipient)
4:
5:     if internalProduct ≠ null then
6:         message ← dSX(leafProduct, s)
7:         message ← dSS(internalProduct, message)
8:         recipient.receiveMessage(message)
9:     else
10:        message ← dSS(leafProduct, s)
11:        recipient.receiveMessage(message)
12:
```

Algorithm 2 Compute marginals.

Let s be the soft information from the channel.

```
1: procedure marginalize
2:     leafProduct ← leafProduct()
3:     internalProduct ← internalProduct()
4:
5:     if internalProduct == null then
6:         marginals ← ·(leafProduct, s)
7:     else if (internalProduct ≠ null) ∧ (|leaves| == 0) then
8:         marginals ← ·(internalProduct, s)
9:     else
10:        marginals ← dSX(internalProduct, leafProduct)
11:        marginals ← ·(marginals, s)
12:
```

Algorithm 3 Compute the leaf-product of all leaves, except for Node recipient.

M is the set of all messages from leaf neighbors.

```
1: function LEAFPRODUCT(Node recipient)
2:     product ← [1, 0, 1, 0]
3:     for all m_i ∈ M do
4:         if sender(m_i) ≠ recipient then
5:             product ← tSX(m_i, product)
6:
       return product
```

Algorithm 4 Compute the internal-product of all internal nodes, except for Node recipient.

M is the set of all messages from internal neighbors. $R \subseteq M$ is the set of messages from the recipient. If the recipient is a leaf, then $R = \varnothing$.

```
1: function INTERNALPRODUCT(Node recipient)
2:     M' ← M \ R
3:     product ← m'_0
4:     for all m'_i ∈ M', where i > 0 do
5:         product ← dSX(m'_i, product)
6:
       return product
```

Algorithm 5 Compute the leaf-product of all leaves.

M is the set of all messages from leaf neighbors.

```
1: function LEAFPRODUCT
2:     product ← [1, 0, 1, 0]
3:     for all m_i ∈ M do
4:         product ← tSX(m_i, product)
5:
       return product
```

Algorithm 6 Compute the internal-product of all internal nodes.

M is the set of all messages from internal neighbors.

```
1: function internalProduct
2:     product ← m_0
3:     for all m_i ∈ M, where i > 0 do
4:         product ← dSX(m_i, product)
5:
       return product
```

4.4 Global Marginals for Trees

In this section we show that Discriminative decoding computes the global marginals for trees exactly. First we examine the leaf-product and the internal-product produced by Algorithms 5 and 6 to show their sums of products take on a particular form. Thereafter, we show by induction on the height of the tree that all internal nodes send messages that satisfy their own constraints. We use this induction in a final proof concluding that for any tree the global marginals of the internal nodes and the leaf nodes are computed. By this our procedure is an instance of the sum-product algorithm for the special case of graph codes.

4.4.1 Leaf-product and Internal-product

In this section we show that the leaf-product and the internal product maintain a certain pattern of entries. For the sake of convenience, we name two varieties of sums of products, the *even* type and the *odd* type. A sum of products A is said to be even with respect to a type of factor, y, if all products of A contain an even amount of y factors. If it is the case that all products of A contain an odd amount of y factors, then A is odd with respect to y.

Lemma 4.4.1. *Let A be a sum of products containing factors of type x and y.*

1. *If A is even wrt. y, then xA is even.*
2. *If A is even wrt. y, then yA is odd.*
3. *If A is odd wrt. y, then xA is odd.*
4. *If A is odd wrt. y, then yA is even.*

Proof. Proof of this is trivial. □

Lemma 4.4.2. *Let A_1 and A_2 be two sums of products containing factors of type x and y.*

1. *If A_1 is even and A_2 is odd wrt. y, then A_1A_2 is odd.*
2. *If A_1 and A_2 are both either odd or even wrt. y, then A_1A_2 is even.*

Proof. Proof of this is trivial. □

We say that a length four vector takes on the *even-odd form*, if the first and the third entries are even with respect to a factor of type y and the second and fourth entries are odd with respect to y. We denote the channel information of any node v_k by the vector $s_k = (e_k, f_k, g_k, h_k)$. Entries of messages may be referred to as p_0, p_1, p_ω, p_{ω^2} values under circumstances when we are not concerned with where the message comes from.

Leaf-product Here we show that the leaf-product of any node v_0 always takes on the form of even-odd with respect to factors of type p, where $p \in \{p_\omega, p_{\omega^2}\}$. Let $\mathcal{L}_0$ denote the set of leaves, neighbor to v_0. From the base case $|\mathcal{L}_0| = 1$ we have the leaf product, $leaves_1$.

$$leaves_1 = \begin{pmatrix} e_1 \\ g_1 \\ f_1 \\ h_1 \end{pmatrix} = \begin{pmatrix} E_1 \\ F_1 \\ G_1 \\ H_1 \end{pmatrix} \tag{4.1}$$

Assume for $|\mathcal{L}_0| = n$ that the leaf product, $leaves_n$, is even-odd with respect to p. We now add a leaf node, v_{n+1}, to the neighborhood of v_0 and show that the operation $tSX(s_{n+1}, leaves_n)$ produces a leaf product, $leaves_{n+1}$ also is of even-odd form.

$$leaves_n = \begin{pmatrix} E \\ F \\ G \\ H \end{pmatrix} \quad leaves_{n+1} = \begin{pmatrix} Ee_{n+1} + Ff_{n+1} \\ Fe_{n+1} + Ef_{n+1} \\ Gg_{n+1} + Hh_{n+1} \\ Hg_{n+1} + Gh_{n+1} \end{pmatrix} \tag{4.2}$$

By Lemma 4.4.1 and the induction hypothesis, we have that $leaves_{n+1}$ is of even-odd form. The entries l_0 and l_1 are even and odd with regards to p_ω values, respectively. Similarly for l_2 and l_3 with respect to p_{ω^2} values.

Internal-product Let $\mathcal{I}_0$ denote the set of internal nodes neighboring v_0. Here we show that the internal-product of any node v_0 always takes on the form of even-odd with respect to factors of type p, where p relates to the internal neighbors of v_0 and $p \in \{p_\omega, p_{\omega^2}\}$. From the base case $|\mathcal{I}_0| = 1$ we have the internal-product, $internal_1$ (4.3), which is the single message received from v_i. We see here that $internal_1$ is trivially even-odd with respect to p factors of v_i.

$$internal_1 = \begin{pmatrix} e_iE + f_iF \\ g_iG + h_iH \\ e_iF + f_iE \\ g_iH + h_iG \end{pmatrix} = \begin{pmatrix} A_1 \\ B_1 \\ C_1 \\ D_1 \end{pmatrix} \tag{4.3}$$

Let $internal_n$ denote the internal-product that is the result of n messages from internal nodes. Assume $internal_n$ (4.4) takes on the form of even-odd with respect to p. Let v_0 gain an internal neighbor v_{n+1} and receive the message $m_{(v_{n+1},v_0)}$ (4.4). From the base case, we know this message to also be even-odd with respect to p factors.

$$internal_n = \begin{pmatrix} A_n \\ B_n \\ C_n \\ D_n \end{pmatrix} \quad m_{(v_{n+1},v_0)} = \begin{pmatrix} M_0 \\ M_1 \\ M_2 \\ M_3 \end{pmatrix} \tag{4.4}$$

From Lemma 4.4.2, the base case, and the induction hypothesis, we know that the product $internal_{n+1}$ (4.5), is even-odd with respect to p factors.

$$internal_{n+1} = dSX(m_{(v_{n+},v_0)}, internal_n) = \begin{pmatrix} M_0A_n + M_1B_n \\ M_0B_n + M_1A_n \\ M_2C_n + M_3D_n \\ M_2D_n + M_3C_n \end{pmatrix} \tag{4.5}$$

4.4.2 Message Satisfaction for Internal Nodes

Here we prove by induction on the height of the tree d that internal nodes send messages that satisfy their own constraints. Let v_d be the root of a tree of height d.

Base Case

When $d = 1$, we have that the neighborhood of v_1 consists entirely of leaves. Any message calculated by v_1 follows the steps from Algorithm 1 lines 9.-11. From Chapter 3, we know the constraints on v_1 to be the following:

$$f_1 = a_1 + \sum_{k \in \mathcal{N}_1} b_k = 0 \tag{4.6}$$

We see here that if $v_1 \in \{0, \omega\}$, then $\sum_{k \in \mathcal{N}_1} b_k = 0$, which is to say that any sum of products multiplied by e_1 or g_1 should be *even* with respect to p_ω and p_ω^2 values. Similarly, if $v_1 \in \{1, \omega^2\}$, then $\sum_{k \in \mathcal{N}_1} b_k = 1$, and any sum of products multiplied by f_1 or h_1 should be *odd* with respect to p_ω and p_ω^2 values. When considering the constraints on v_1 in relation to the recipient of a message, v_i, we can state (4.6) as the following:

$$f_1 = a_1 + b_i + \sum_{k \in \mathcal{N}_1 \setminus i} b_k = 0 \tag{4.7}$$

Analyzing this equation tells us that when $v_1 \in \{0, \omega\}$, then $b_i = \sum_{k \in \mathcal{N}_1 \setminus i} b_k$. This indicates that if $v_i \in \{0, 1\}$, then $\sum_{k \in \mathcal{N}_1 \setminus i} b_k$ requires an even number of b_k values. For $v_i \in \{\omega, \omega^2\}$, we have that $\sum_{k \in \mathcal{N}_1 \setminus i} b_k$ requires an odd number of b_k values. When $v_1 \in \{1, \omega^2\}$ we know that $b_i \neq \sum_{k \in \mathcal{N}_1 \setminus i} b_k$. This inequality leads to the conclusion that when $v_i \in \{0, 1\}$, then $\sum_{k \in \mathcal{N}_1 \setminus i} b_k$ requires an odd number of b_k values. Conversely, when $v_i \in \{\omega, \omega^2\}$, an even number of b_k values are required.

These conditions reveal the belief v_1 has about v_i under its own constraints. In terms of message calculation, this means that v_1's message should reflect, for the p_0 or p_1 entries of the message, any sum of products multiplied by e_1 or g_1 should be even. Conversely, for sums of products multiplied by e_1 or g_1 in the p_ω and p_{ω^2} entries of the message. Similar behavior is implied when we consider the case where $b_i \neq \sum_{k \in \mathcal{N}_1 \setminus i} b_k$

The message (4.8) is the result of executing the Algorithm 1 lines 9.-11., where $leaves_{v_1}(i) = (E_i, F_i, G_i, H_i)$ is the leaf-product not containing information from v_i. By the proof in the previous section, we know $leaves_{v_i}(i)$ to be even-odd with respect to $p \in \{p_\omega, p_{\omega^2}\}$.

$$m_{(v_1,v_i)} = \begin{pmatrix} e_1E_i + f_1F_i \\ g_1G_i + h_1H_i \\ e_1F_i + f_1E_i \\ g_1H_i + h_1G_i \end{pmatrix} \tag{4.8}$$

We see here that the message reflects v_1's beliefs, as it satisfies v_1's own constraints in relation to v_i. Any product in the message represents a vector that satisfies the Hermitian constraint. Products in the first entry represent vectors where $v_i = 0$. In accordance with the aforementioned constraints, and by the leaf-product proof, we know that sums of products multiplied by e_1 are all even, and sums of products multiplied by f_1 are all odd. Similarly, the constraints of v_1 are satisfied by the rest of the entries in $m_{(v_1,v_i)}$.

Induction

Assume for height $d = n$, that messages sent by v_n that satisfy its own constraint. We now show for height $d = n + 1$ that v_{n+1} sends messages that satisfy its own constraints. We let the set $\mathcal{I}_{n+1}$ denote the set of internal neighbors to v_{n+1}, and the set $\mathcal{L}_{n+1}$ denote the set of leafs neighbor to v_{n+1}. We handle two cases here. One where $\mathcal{L}_{n+1}$ is empty (4.12), and the other where $\mathcal{L}_{n+1}$ is non-empty (4.13).

$$f_{n+1} = a_{n+1} + b_n + \sum_{k \in \mathcal{I}_{n+1} \setminus n} b_k = 0 \tag{4.9}$$

$$f_{n+1} = a_{n+1} + b_n + \sum_{k \in \mathcal{L}_{n+1} \setminus i} b_k + \sum_{k \in \mathcal{I}_{n+1} \setminus i} b_k = 0 \tag{4.10}$$

Let the internal-product $internal_{n+1}$ and the leaf-product $leaves_{n+1}$ be as in (4.11). From the leaf-product and the internal-product proofs, we know both of these vectors to be even-odd with respect to p factors, where $p \in \{p_\omega, p_{\omega^2}\}$.

$$leaves_{n+1} = \begin{pmatrix} E \\ F \\ G \\ H \end{pmatrix} \quad leaves_{n+1} = \begin{pmatrix} Q \\ R \\ S \\ T \end{pmatrix} \tag{4.11}$$

Without Leaves The message from v_{n+1} to v_n as a result of the Algorithm 1 when v_{n+1} has no leaves is as given in (4.12). The constraint in (4.9) implies the same even-odd behavior of products as in the base case constraint (4.7), except we are now dealing with 'sums of products' sent by internal nodes.

When $v_{n+1} \in \{0, \omega\}$, then $b_n = \sum_{k \in \mathcal{N}_{n+1} \setminus n} b_k$. This indicates that if $v_n \in \{0, 1\}$, then $\sum_{k \in \mathcal{N}_{n+1} \setminus n} b_k$ requires an even number of b_k values. For $v_n \in \{\omega, \omega^2\}$, we have that $\sum_{k \in \mathcal{N}_{n+1} \setminus n} b_k$ requires an odd number of b_k values. When $v_{n+1} \in \{1, \omega^2\}$ we know that $b_n \neq \sum_{k \in \mathcal{N}_{n+1} \setminus n} b_k$. This inequality leads to the conclusion that when $v_n \in \{0, 1\}$, then $\sum_{k \in \mathcal{N}_{n+1} \setminus n} b_k$ requires an odd number of b_k values. Conversely, when $v_n \in \{\omega, \omega^2\}$, an even number of b_k values are required.

$$m_{(v_{n+1}, v_n)} = \begin{pmatrix} Qe_{n+1} + Rf_{n+1} \\ Sg_{n+1} + Th_{n+1} \\ Re_{n+1} + Qf_{n+1} \\ Tg_{n+1} + Sh_{n+1} \end{pmatrix} \tag{4.12}$$

We know from the internal-product proof that $internal_{n+1}$ is even-odd with respect to $p \in \{p_\omega, p_{\omega^2}\}$. We see from (4.12) that v_{n+1} satisfies its own constraints when passing a message to v_n.

With Leaves The message from v_{n+1} to v_i as a result of the Algorithm 1 when $\mathcal{L}_{n+1}$ is non-empty is as given in (4.13).

$$m_{(v_{n+1}, v_n)} = \begin{pmatrix} Q(e_{n+1}E + f_{n+1}F) + R(e_{n+1}F + f_{n+1}E) \\ S(g_{n+1}G + h_{n+1}H) + T(g_{n+1}H + h_{n+1}G) \\ Q(e_{n+1}F + f_{n+1}E) + R(e_{n+1}E + f_{n+1}F) \\ S(g_{n+1}H + h_{n+1}G) + T(g_{n+1}G + h_{n+1}H) \end{pmatrix} \tag{4.13}$$

$$= \begin{pmatrix} e_{n+1}(QE + RF) + f_{n+1}(QF + RE) \\ g_{n+1}(SG + TH) + h_{n+1}(SH + TG) \\ e_{n+1}(QF + RE) + f_{n+1}(QE + RF) \\ g_{n+1}(SH + TG) + h_{n+1}(SG + TH) \end{pmatrix} \tag{4.14}$$

Analyzing the constraint (4.10) in the same manner as before gives us the following.

Lemma 4.4.3. *1. If $v_{n+1} \in \{0, \omega\}$ and $v_n \in \{0, 1\}$, then $\sum_{k \in \mathcal{L}_{n+1} \setminus i} b_k = \sum_{k \in \mathcal{I}_{n+1} \setminus i} b_k$*

2. *If* $v_{n+1} \in \{0, \omega\}$ *and* $v_n \in \{\omega, \omega^2\}$, *then* $\sum_{k \in \mathcal{L}_{n+1} \setminus i} b_k \neq \sum_{k \in \mathcal{I}_{n+1} \setminus i} b_k$

3. *If* $v_{n+1} \in \{1, \omega^2\}$ *and* $v_n \in \{0, 1\}$, *then* $\sum_{k \in \mathcal{L}_{n+1} \setminus i} b_k \neq \sum_{k \in \mathcal{I}_{n+1} \setminus i} b_k$

4. *If* $v_{n+1} \in \{1, \omega^2\}$ *and* $v_n \in \{\omega, \omega^2\}$, *then* $\sum_{k \in \mathcal{L}_{n+1} \setminus i} b_k = \sum_{k \in \mathcal{I}_{n+1} \setminus i} b_k$

We show entry by entry how the message in (4.13) satisfies the constraints of v_{n+1}.

Entry 1

i. The product $e_{n+1}(QE + RF)$ represents when $v_{n+1} = 0$ and $v_n = 0$.

   We know from the internal-product proof that both QE and RF are products of 'sums of products' of the same type with respect to being even or odd.

ii. The product $f_{n+1}(QF + RE)$ represents when $v_{n+1} = 1$ and $v_n = 0$.

   We also know that QF and RE are products of 'sums of products' that differ with respect to being even or odd.

iii. Item Entry 1.i. satisfies the 4.4.3.1 and item Entry 1.ii. satisfies the 4.4.3.3

Entry 2

i. The product $g_{n+1}(SG + TH)$ represents when $v_{n+1} = \omega$ and $v_n = 1$.

   The internal-product proof gives us that both SG and TH are products of 'sums of products' of the same type with respect to being even or odd.

ii. The product $h_{n+1}(SH + TG)$ represents when $v_{n+1} = \omega^2$ and $v_n = 1$.

   We also know that SH and TG are products of 'sums of products' that differ with respect to being even or odd.

iii. Item Entry 2.i. satisfies the 4.4.3.1 and item Entry 2.ii. satisfies the 4.4.3.3

Entry 3

i. The product $e_{n+1}(QF + RE)$ represents when $v_{n+1} = 0$ and $v_n = \omega$.

We know that QF and RE are products of 'sums of products' that differ with respect to being even or odd.

ii. The product $f_{n+1}(QE + RF)$ represents when $v_{n+1} = 1$ and $v_n = \omega$.

The internal-product proof gives us that both QE and RF are products of 'sums of products' of the same type with respect to being even or odd.

iii. Item Entry 3.i. satisfies the 4.4.3.2 and item Entry 3.ii. satisfies the 4.4.3.4

Entry 4

i. The product $g_{n+1}(SH + TG)$ represents when $v_{n+1} = \omega$ and $v_n = \omega^2$.

We know that SH and TG are products of 'sums of products' that differ with respect to being even or odd.

ii. The product $h_{n+1}(SG + TH)$ represents when $v_{n+1} = \omega^2$ and $v_n = \omega^2$.

The internal-product proof gives us that both SG and TH are products of 'sums of products' of the same type with respect to being even or odd.

iii. Item Entry 3.i. satisfies the 4.4.3.2 and item Entry 3.ii. satisfies the 4.4.3.4

4.4.3 Proof of Global Marginal Computation

We now move on to showing how any node on any tree computes its own global marginals. We do this by showing that the Algorithm 2 for computing the marginals satisfies the constraints of the computing node. The complete argument is as follows; Given that all messages satisfy the constraints of the sender. If a node satisfies its own constraints while computing the marginals based on information that satisfies its neighborhood and the rest of the graph, then its computed marginals will satisfy the global function and will be equal to the global marginals.

Leaves

For any leaf node v_i neighbor to v_n, we have the constraint $f_i = a_i + b_n = 0$. It follows from the base case of the previous induction that the pointwise product of v_i's channel information and the message from v_n satisfies v_i's constraint.

Internal Nodes

To show that an internal node satisfies its own constraints when computing its marginals we consider three types of neighborhoods. The first consists entirely of leaf nodes, the second consists of internal nodes, and the third is a combination of both.

Neighborhood of Leaves

Lemma 4.4.4. *For any leaf node v_i on any graph code, where v_n denotes its parent, we have the following.*

$$v_i \in \{0, \omega\} \iff v_n \in \{0, 1\}$$

$$v_i \in \{1, \omega^2\} \iff v_n \in \{\omega, \omega^2\}$$

Proof. The constraint on any leaf v_i, connected to an internal node v_n, is given by the equation $f_i = a_i + b_n = 0$. If $v_i \in \{0, \omega\}$, then $a_i = 0$. The equation f_i is only satisfied if $b_n = 0$, i.e. if $v_n \in \{0, 1\}$. Similarly, if $v_i \in \{1, \omega^2\}$, then $a_i = 1$, which means that f_i is only satisfied if $b_n = 1$, i.e. if $v_n \in \{\omega, \omega^2\}$. □

Lemma 4.4.5. *v_n is a node with a neighborhood, $\mathcal{N}_n$, consisting entirely of leaves. N_z denotes the set of neighbors of v_0 bearing the value $z \in \mathbb{F}_4$. We have the following for all $v_i \in \mathcal{N}_0$.*

1. $v_0 = 0 \implies v_i \in \{0, \omega\}$ *and* $|N_\omega| = 2k$, *where* $0 \leq k \leq n - 1$.
2. $v_0 = 1 \implies v_i \in \{0, \omega\}$ *and* $|N_\omega| = 2k + 1$, *where* $0 < k \leq n - 1$.
3. $v_0 = \omega \implies v_i \in \{1, \omega^2\}$ *and* $|N_{\omega^2}| = 2k$, *where* $0 \leq k \leq n - 1$.
4. $v_0 = \omega^2 \implies v_i \in \{1, \omega^2\}$ *and* $|N_{\omega^2}| = 2k + 1$, *where* $0 < k \leq n - 1$.

Proof. Here we demonstrate the lemma for each of the possible values of v_0.

Let $v_n = 0$, then $a_n = 0$ and $b_n = 0$. For all leaves v_i, we have that $a_i = 0$, since $b_n = 0$. This restricts the possible values of our leaves to either $\{0, \omega\}$. Furthermore, since $a_n = 0$, we know that $\sum_{u \in N_n} b_u = 0$, which can only happen if there is an even amount of $b_u = 0$. This corresponds to the requirement that for any codeword where $v_n = 0$, there is an even amount of ωs, among the remaining v_i. It follows that if $v_n = 1$, i.e. $a_n = 1$ and $\sum_{u \in N_n} b_u = 1$, that the constraint is only satisfied if there is an odd amount of ωs.

Let $v_n = \omega$, then $a_n = 1$ and $b_n = 1$. For all leaves v_i, we have that $a_i = 0$, since $b_n = 1$. This restricts the possible values of our leaves to either $\{1, \omega^2\}$. We also have that $\sum_{u \in N_n} b_u = 0$, which entails that for any codeword where $v_n = \omega$ there is an even amount of ω^2s. Similarly for when $v_n = \omega^2$, i.e. $a_n = 1$ and $b_n = 1$, we have that the constraint is satisfied if there is an odd amount of ω^2s.

□

When v_n's neighborhood consists of only leaves, then v_n computes its marginals by taking the pointwise product of the leaf-product produced by Algorithm 5. From the leaf-product proof, we know the leaf-product to be even-odd with respect to factors of type p, where $p \in \{\omega, \omega^2\}$. It follows from this that the marginal computation of v_n is equal to the global marginals.

Neighborhood of Internal Nodes

Lemma 4.4.6. *For any node v_n , where $\mathcal{N}_n$ consists entirely of internal nodes and N_z denotes the set of neighbors of v_n bearing the value $z \in \mathbb{F}_4$, the following holds true.*

1. $v_n \in \{0, \omega\} \implies |\mathcal{N}_\omega \cup \mathcal{N}_{\omega^2}| = 2k$, *where* $k \geq 0$.
2. $v_n \in \{1, \omega^2\} \implies |\mathcal{N}_\omega \cup \mathcal{N}_{\omega^2}| = 2k + 1$, *where* $k \geq 0$.

Proof. The constraint of v_n is given by $f_n = a_n + \sum_{j \in \mathcal{N}_n} b_j = 0$.

If $a_n = 0$, then $\sum_{j \in \mathcal{N}_n} b_j = 0$, which can only be true if there is an even number of ωs and ω^2s. Therefore, if $v_n \in \{0, \omega\}$, then the number of ωs and ω^2s must be equal to $2k$, where $k \geq 0$. If $a_n = 1$, then $\sum_{j \in \mathcal{N}_n} b_j = 1$, which can only be true if there is an odd number of ωs and ω^2s. Therefore, if $v_n \in \{1, \omega^2\}$, then the number of ωs and ω^2s must be equal to $2k + 1$, where $k \geq 0$.

□

When v_n's neighborhood consists of only internal nodes, it computes its marginals by taking the pointwise product of the internal-product produced by Algorithm 6. From the internal-product proof, we know the internal-product to be even-odd with respect to factors of type p, where $p \in \{p_\omega, p_{\omega^2}\}$. By proof of Lemma 4.4.6, it follows from this that the marginal computation of v_n is equal to the global marginals.

Combined Neighborhood

Lemma 4.4.7. *For any node v_n on a tree, with the set of leaf neighbors L_n and the set of internal neighbors I_n, the following holds true.*

$$f_n = a_n + \sum_{k \in L_n} b_k + \sum_{j \in I_n} b_j = 0 \tag{4.15}$$

1. $v_n \in \{0, \omega\} \implies \sum_{k \in L_n} b_k = \sum_{j \in I_n} b_j$
2. $v_n \in \{1, \omega^2\} \implies \sum_{k \in L_n} b_k \neq \sum_{j \in I_n} b_j$

Proof. Follows from the double binary $v_i = a_i + \omega b_i$, where $a_i, b_i \in \mathbb{F}_2$, and the constraint (4.15). □

In this scenario v_n computes its global marginals by taking the pointwise product of its channel information and the result of $dSX(internal_n, leaves_n)$. The operation of taking the pointwise product satisfies v_n's constraints trivially. What is left to show is that dSX produces a vector that satisfies the implications of Lemma 4.4.7.

$$leaves_{v_0} = \begin{pmatrix} E \\ F \\ G \\ H \end{pmatrix} \quad internal_{v_0} = \begin{pmatrix} Q \\ R \\ S \\ T \end{pmatrix} \tag{4.16}$$

$$dSX(internal_n, leaves_n) = \begin{pmatrix} EQ + FR \\ ER + FQ \\ GS + HT \\ GT + HS \end{pmatrix} \tag{4.17}$$

Lemma 4.4.7 is satisfied for the first and third entry, if they consist of products of 'sums of products' like those from Lemma 4.4.2.2, where they are of the same type. Lemma 4.4.7 is satisfied for the second and fourth entry, if they consist of products of 'sums of products' like those from Lemma 4.4.2.1, where they differ in type. By Lemma 4.4.2, we have that dSX in (4.16) and (4.18) satisfy Lemma 4.4.7.

$$g(v_0) = \begin{pmatrix} e_0(EQ + FR) \\ f_0(ER + FQ) \\ g_0(GS + HT) \\ h_0(GT + HS) \end{pmatrix} \tag{4.18}$$

We have now shown that Discriminative decoding computes the global marginals for any node on any tree. By showing Discriminative to compute the global marginals for trees, we show that it is an instance of the sum-product algorithm described in Chapter 2.

Chapter 5

Simulation Tool

In this chapter we present the simulation tool designed to simulate the embedded factor graph decoding from Chapter 3. and the discriminative decoding scheme from Chapter 4.

5.1 Message Passing on Graph Codes

The application implemented in association with the work of this thesis can simulate two types of message passing decoding on $\mathbb{F}_4$-additive codes in graph form – Embedded Decoding and Discriminative Decoding, as described in Chapter 3 and Chapter 4, respectively. Its initial design was aimed towards a pure simulation tool for embedded decoding on the quaternary symmetric channel. However, after the discoveries in Chapter 3 were made, the application gained functionality capable of validation, or repudiation of results. This aided the search for a message passing algorithm which could compute, as accurately as possible, the marginals from the global function. The application is written in Java and has no specific requirements other than Java 7 JRE.

Input and Output Upon running the software, the user is prompted to provide two values; path to a *code specification file* and the number of simulations she wishes to run, k. The message for each simulated transmission is the all-zero codeword. Results from each decoding appear in the terminal, displaying the *received transmission*, *decoded vector*, and a statement of whether or not the decoded vector is a member of the code.

1.	double p	transition probability of the QSC
2.	int $d \in \{1, 2\}$	type of decoding
3.	String $g \in \{t, c\}$	type of graph
4.	int n	number of nodes
5.	...	the next n lines of the file are rows of H^1

Figure 5.1: The format of the code specification file

Code Specification File This file provides necessary information regarding the simulated channel and the code on which we intend to decode. It should be a plain text file (.txt). The first two lines specify the *transition probabilities* of the channel, and which decoding method to be used. The remaining lines describe the code in terms of its graph. Figure 5.1 describes the required format of the code specification file. The transition probability p is the probability a bit is not flipped. Decoding methods are chosen by specifying 1 for embedded decoding, and 2 for discriminative decoding. If the graph is a tree, then the value of line 3. is t, otherwise c indicates the graph is cyclic.

5.2 Application Structure

Simulating message passing decoding, as a software implementation problem, requires a fulfillment of more details than compared to theoretical work on the same subject. The channel implementation represents a real world, probabilistic object that applies noise to transmissions according to some model. Accurately representing this model requires a random number generator. The field $\mathbb{F}_4$ and its operations addition and multiplication must be available to several components of the system. The code and the decoding algorithm must also be realized in a sensible manner. There are several approaches to finding a solution to this implementation problem. Here we give a brief account of the object-oriented solution to our simulation application in terms of its structure. We consider only the conceptual structure realized through package structure, the *base framework*, and the *decoder* package. A complete class digram may be found in Figure 5.5.

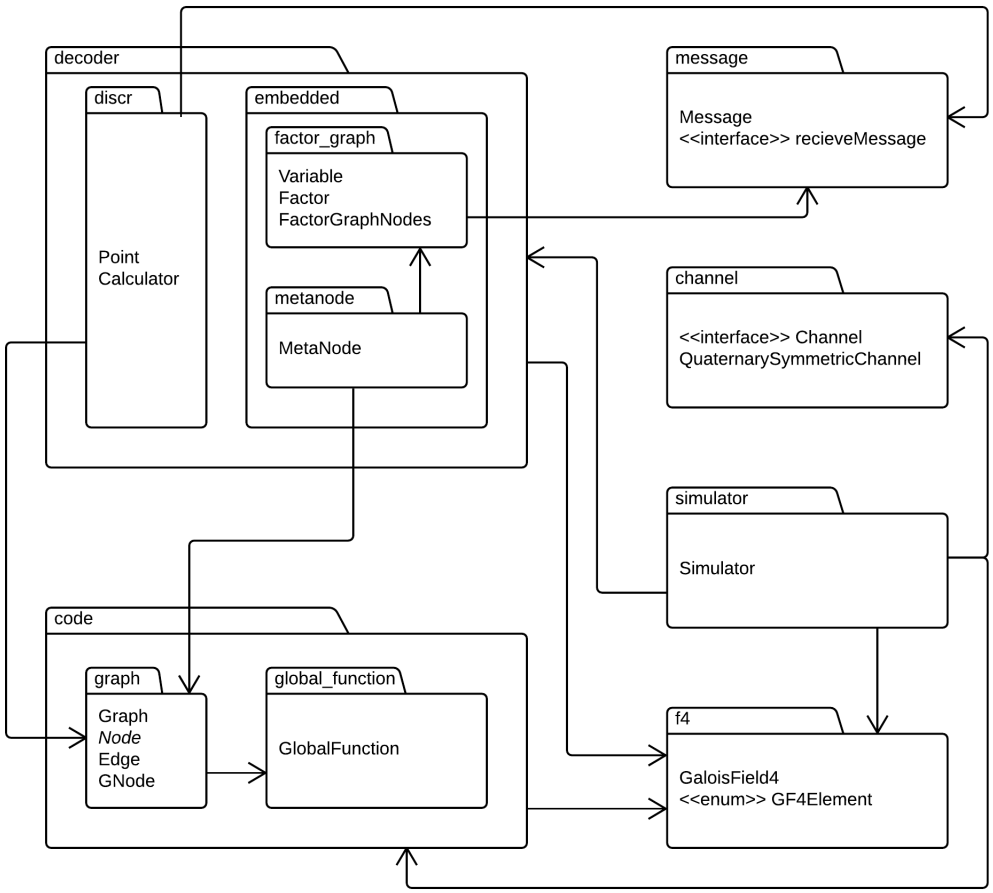


Figure 5.2: Package diagram

Conceptual Entities The packages of the implementation express the mathematical and logical entities of our domain. The field $\mathbb{F}_4$, the channel, the code, and the decoder are kept separated – these represent our communication system. Furthermore, we have a simulator package and a message package. In Figure 5.2 one may find a diagram depicting the dependency relationships between our packages and sub-packages. An arrow from one package to another indicates that the one package depends on a class in the other. If there are sub-packages, then the arrow indicates that the one package depends on at least one class from each sub-package.

Base Framework The base framework is the foundation of the simulation tool. It is the code that is independent of any specific channel model or message calculation algorithm. This includes a *Simulator* class, a *Channel* interface, a *GlobalFunction* class, a representation of $\mathbb{F}_4$, and a *Graph* class with its subcomponents; *Node*, *Edge*, and *GNode*. Figure 5.3 depicts the relationship between the components of the base framework. The Simulator sends transmissions through the Channel, provides the Graph with the output of the Channel to decode, and validates the decoding by asking the GlobalFunction. Two requirements must be met in order for the Simulator to be able to run simulations. Firstly, a channel model must be realized by implementing the Channel interface. Our answer to this is the *QuaternarySymmetricChannel* class, which may be found in the Figure 5.5. Secondly, the abstract class Node needs to be extended with a class that can perform the message calculation – the essential computation during message passing. We have realized this through the two classes, *Point* and *MetaNode*, in the *decoder* package. Point is the node associated with discriminative decoding and MetaNode is the node associated with embedded decoding.

Decoder Package There are two sub-packages in the decoder package, *discr* and *embedded*. These are *not* decoders themselves, but rather packages of classes necessary to realize different decoding schemes with the base framework introduced in the previous paragraph. The *discriminative* package contains two classes, *Point* and *Calculator*. Point is an extension of Node which uses a Calculator object to calculate messages to its neighbors. This package makes it possible to realize the scheme presented in Chapter 4. The *embedded* package contains an additional pair of sub-packages, *factor_graph* and *metanode*. The latter contains the class *MetaNode*, which is an extension of Node that realizes the meta nodes from the embedded decoding scheme of Chapter 3. The *factor_graph* package contains the abstract class *FactorGraphNode* and two

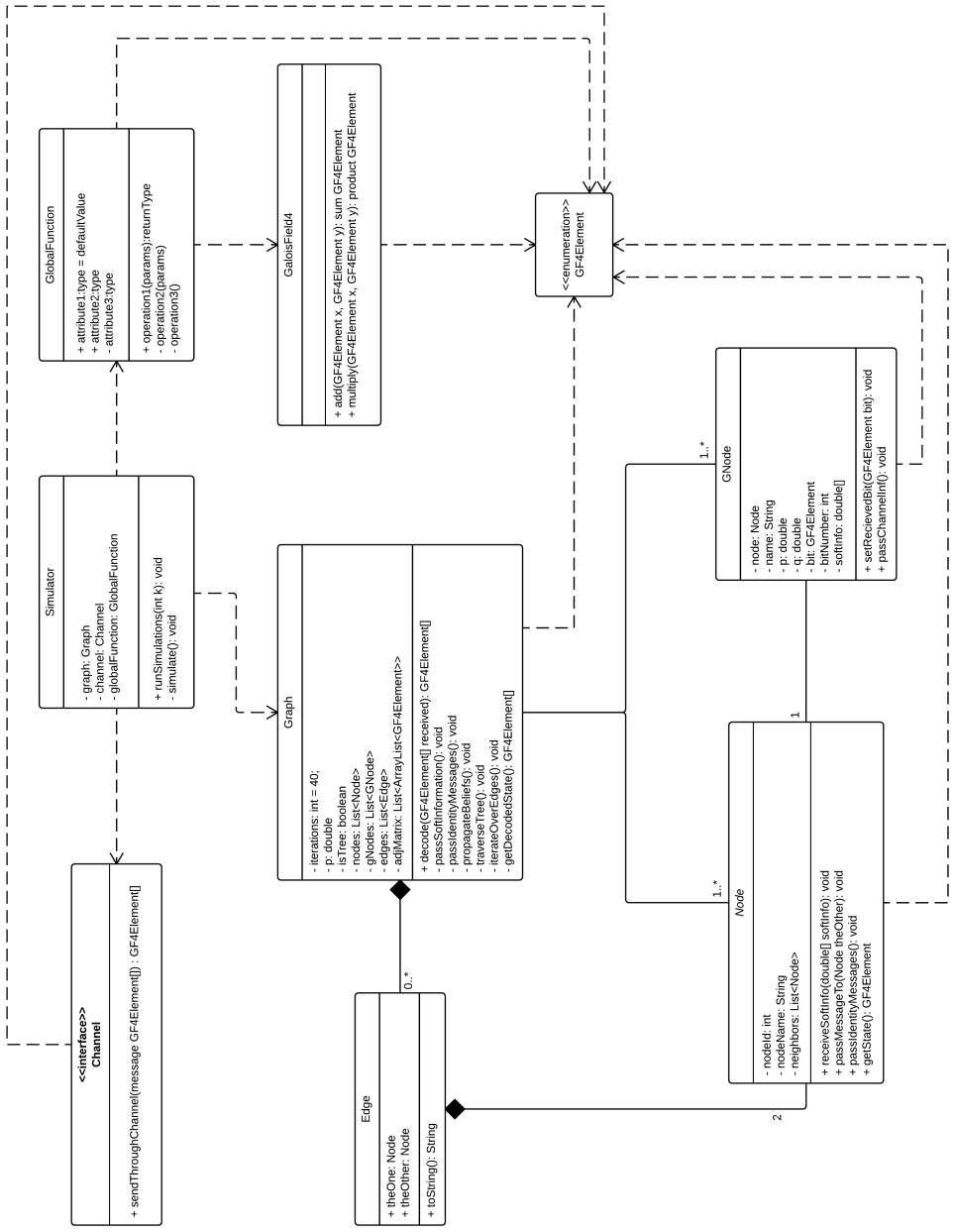


Figure 5.3: Base framework – class diagram.

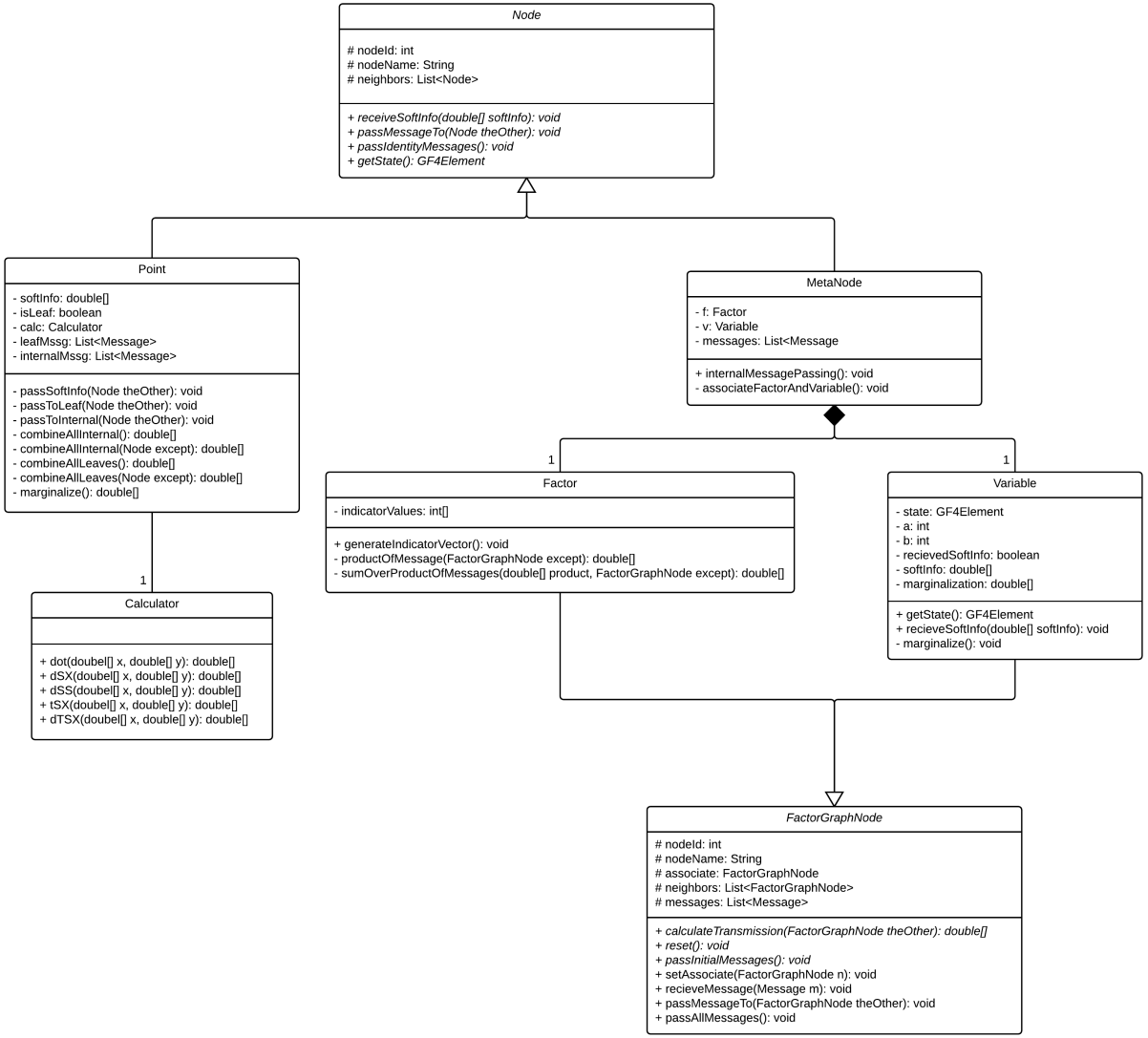


Figure 5.4: Node strategy.

classes extending it; *Factor* and *Variable*. The contents of the *embedded* package together with the base framework realize the embedded decoding of Chapter 3.

5.3 Behavior and Key Features

Our application was built to perform message passing decoding on graph codes using two types of message passing schemes. In this section we consider the behavior of our implementation which allows for this functionality. We begin by presenting essential behavior invariant with respect to message calculation, i.e. the behavior of the central objects in the base framework. This includes simulation and decoding methods found in Simulator and Graph, respectively. Thereafter, we look at the message passing behavior of Point and MetaNode, the two extensions of Node.

5.3.1 The Simulator

We begin by describing the Simulator as its behavior is invariant with respect to decoding scheme.

A Simulation The sequence diagram in Figure 5.6 illustrates the sequence of collaboration with other classes required for a single simulation caused by calling *runSimulation()* on a Simulator object. The Simulator begins by sending the all-zero codeword through the Channel – *sendThroughChannel(codeWord)*. Once it has received output from the Channel it passes it along to the Graph for decoding – *decode(received)*. After decoding, the Simulator checks with the GlobalFunction of the code to verify whether or not the decoded vector is a codeword.

5.3.2 A Decoding

The first thing Graph does when it is called upon to decode a received transmission is to pair the i-th entry in the received transmission with the i-th GNode and prompt each GNode to pass along soft information to the i-th Node. In order for belief propagation to occur, the Graph calls upon each Node to pass identity messages to their respective neighborhoods. Depending

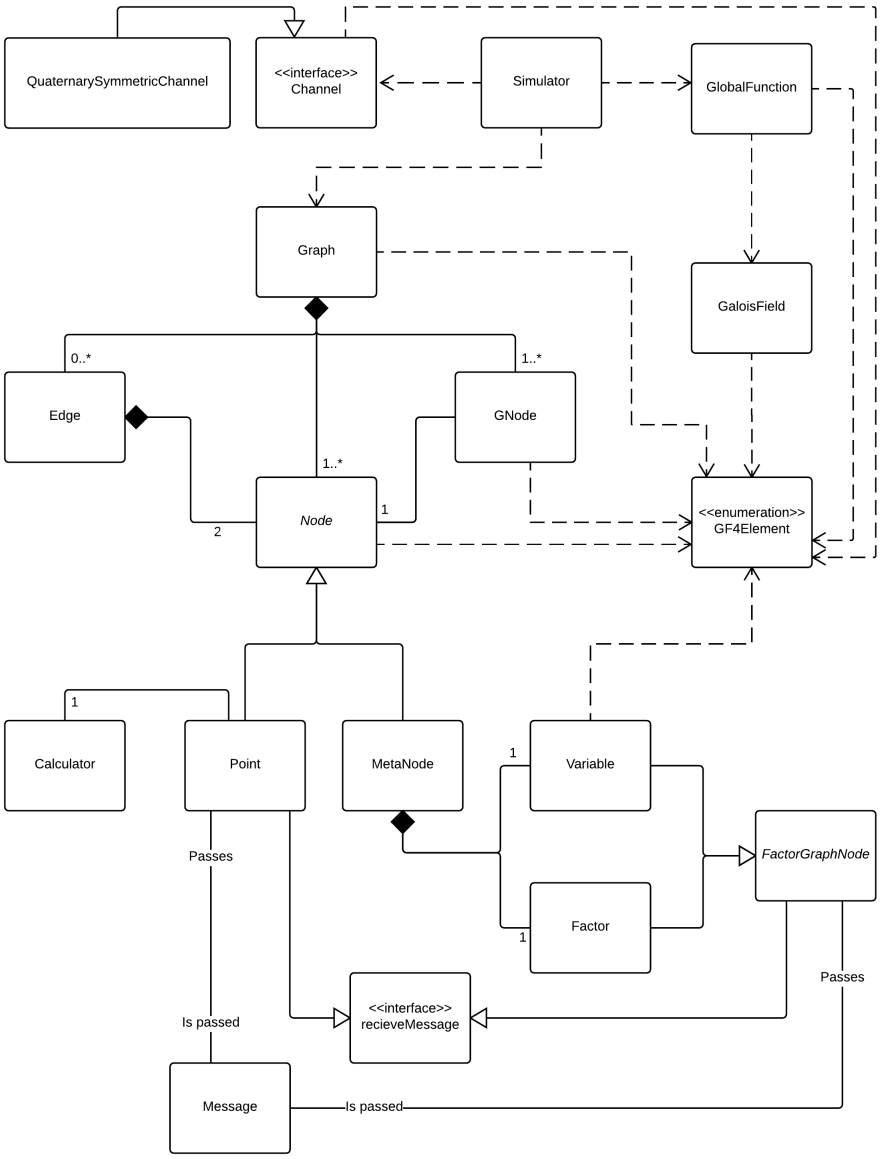


Figure 5.5: Complete class diagram

on whether or not Graph is a tree, beliefs are propagated through calling *traverseTree()* or *iterateOverEdges()*. Finally, before returning the decoded codeword to the Simulator, Graph asks each Node for its *state*.

Belief propagation by *iterateOverEdges()* involves message passing by iterating over all edges of the graph until messages have been passed in both directions on each edge a predetermined number of times. This is initialized by passing identity messages along each edge.

The method *traverseTree()* involves passing messages in two stages, first from the leaves to the root[2], then from the root to the leaves. The former stage is implemented with a recursive method *passMessagesFromLeaves*($\lfloor \frac{n}{2} \rfloor$), which elicits message passing from a child node to a parent node in post-order. The latter, *passMessagesToLeaves*($\lfloor \frac{n}{2} \rfloor$), elicits message passing from a parent node to all of its children in pre-order. The message passing is done after the stages are complete.

5.3.3 Passing a Message

Embedded Decoding

Nodes that belong to the class *MetaNode* pass messages between each other by means of their embedded factor graph nodes, *Factor f* and *Variable v*. Consider the scenario where the MetaNode M_i is elicited to pass a message to M_j. First f_i and x_i pass messages to each other, thereafter they pass to their respective neighbors embedded in M_j, i.e. f_i passes to x_j and x_i to f_j. The embedded Factor f calculates its messages by performing the sum-product computation on its truth table, as described in Chapter 3. The Variable v computes its messages by multiplying the beliefs of its neighbors, as described in Chapter 2.

Discriminative Decoding

Describing the message passing procedure of a Point node is a little more involved than describing that of the MetaNode. The Point node differentiates between passing a message to a leaf and passing a message to an internal node, and treats messages from these types of differently as well. Furthermore, the Point requires the use of four types of vector multiplication in order to

[2]The root is chosen to be the $\lfloor \frac{n}{2} \rfloor$-th node

calculate messages – for this it uses a Calculator object. The Calculator provides functionality for performing four binary vector operations; the point product, and three others which we have named *dSS(u,v)*, *dSX(u,v)*, and *tSX(u,v)* – as described in Chapter 4.

If the transmitting node is itself a leaf, then it merely passes along its own soft information as a message. We will not consider this case here. On the other hand, if the sender is an internal node, then it must consider whether or not its recipient is a leaf. This is because Point handles messages from leaf nodes and internal nodes separately. Therefore, in order to pass a message it must compute two products; the product of leaf-messages and the product of internal-messages.

Products of Messages The product of the leaf-messages is computed by calling either *combineAllLeaves()* or *combineAllLeaves(Node theOther)*. If the recipient is a leaf node, then the leaf-product is computed by calling the latter method. This method excludes the beliefs of the recipient so as to avoid echoing back its own information. The computations performed in both methods are the same, the only difference being the exclusion of a belief.

Similarly for internal nodes, if the recipient is an internal node, then the internal-product is computed by calling *combineAllInternals(Node theOther)*. Otherwise, we use the method *combineAllInternal()*. Both of these return *null* if there are no internal-messages to consider.

If the internal-product is returned *null*, then Point transmits the result of calling *calculator.dSS(leaves, softInfo)* on its Calculator. Otherwise, it combines the soft information with the two products in two steps. First, it computes m = *calculator.dSX(leaves, softInfo*, thereafter it transmits the output of *calculator.dSS(m, internals)*

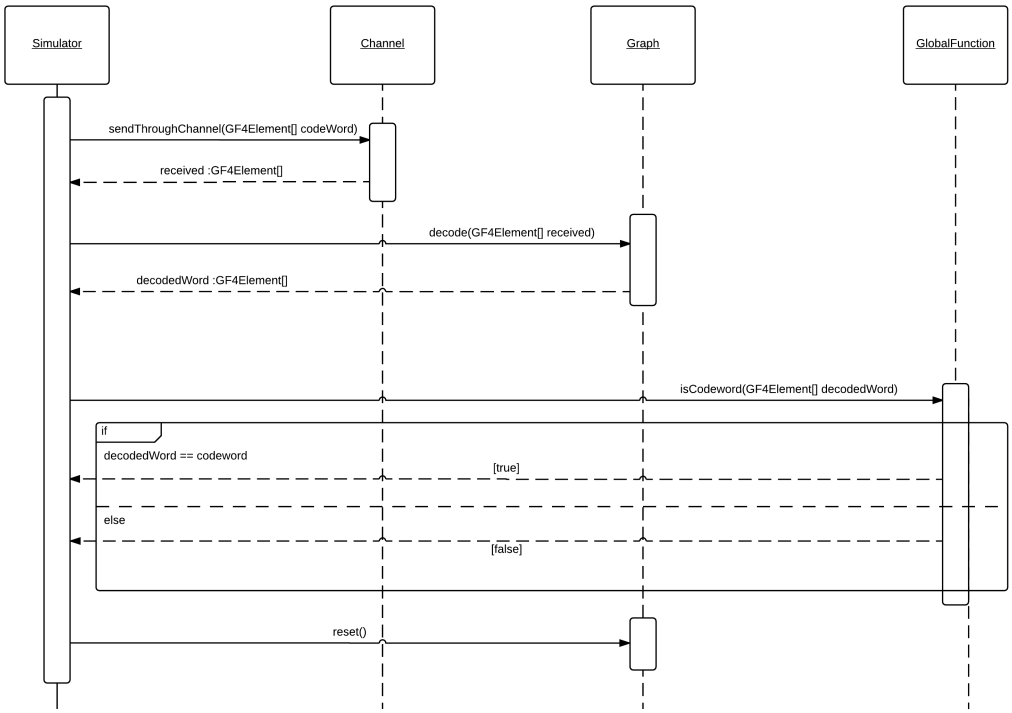


Figure 5.6: Sequence diagram of runSimulation() in Simulator.class.

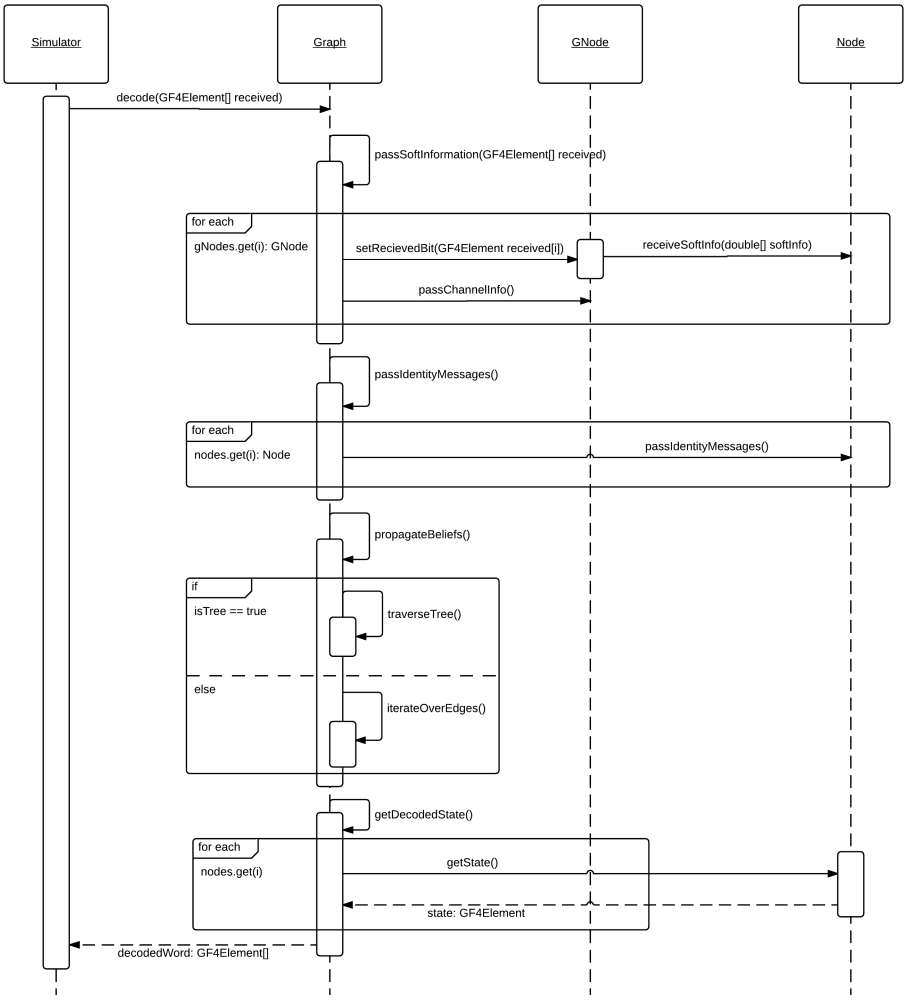


Figure 5.7: Sequence diagram of decode(transmission) in Graph.class.

Chapter 6

Analysis

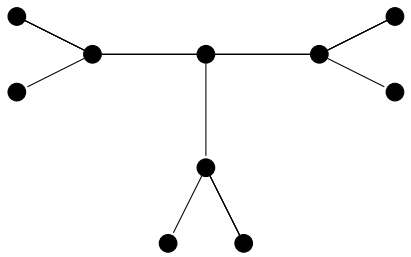


Figure 6.1: Tree.

From Chapter 4, we know our algorithm to be an instance of the sum-product algorithm on self-dual $\mathbb{F}_4$-additive codes in graph form. The sum-product algorithm is known to be an exact method for computing marginals on trees. Marginal computation on graphs containing cycles, however, is not guaranteed to be exact, nor to converge towards particular approximations [16, 23]. Here we provide a brief report of data produced by running simulations of discriminative decoding on trees and graphs with cycles. Each node was provided with a noisy vector taken from a uniformed distribution. Results from the simulation were compared with the global marginals. Note, simulations were executed with the quaternary symmetric channel with similarly successful results, however, due to time constraints we do not present those result.

6.1 Marginals of Trees

We simulated message-passing decoding on the tree in Figure 6.1. A comparison of the computed marginals and the marginals collected from the global

function may be found in Table 6.1. These values are as to be expected from Chapter 4, some values may differ in the last couple of decimal points. We attribute these differences to the occasional inaccuracies that might arise during floating-point multiplication and addition with Java, as these operations are not associative [2, 1].

-	Global Marginals	Decoded Marginals
v_0	$\begin{pmatrix} 0.3043016475642586 \\ 0.3692815693246852 \\ 0.09690964937579223 \\ 0.22950713373526396 \end{pmatrix}$	$\begin{pmatrix} 0.3043016475642584 \\ 0.36928156932468525 \\ 0.09690964937579225 \\ 0.22950713373526402 \end{pmatrix}$
v_1	$\begin{pmatrix} 0.5206226697492123 \\ 0.06492453379902476 \\ 0.23176947430133385 \\ 0.18268332215042904 \end{pmatrix}$	$\begin{pmatrix} 0.5206226697492125 \\ 0.06492453379902477 \\ 0.2317694743013339 \\ 0.18268332215042904 \end{pmatrix}$
v_2	$\begin{pmatrix} 0.41469390782761706 \\ 0.3093493097822275 \\ 0.17085329572062002 \\ 0.10510348666953549 \end{pmatrix}$	$\begin{pmatrix} 0.4146939078276171 \\ 0.3093493097822274 \\ 0.1708532957206199 \\ 0.10510348666953546 \end{pmatrix}$
v_3	$\begin{pmatrix} 0.46592311660130026 \\ 0.17752884243021078 \\ 0.11962408694693685 \\ 0.23692395402155209 \end{pmatrix}$	$\begin{pmatrix} 0.46592311660130026 \\ 0.17752884243021078 \\ 0.11962408694693684 \\ 0.2369239540215521 \end{pmatrix}$
v_4	$\begin{pmatrix} 0.17925281525999412 \\ 0.1487609303962149 \\ 0.44457502293200774 \\ 0.22741123141178324 \end{pmatrix}$	$\begin{pmatrix} 0.17925281525999406 \\ 0.1487609303962149 \\ 0.44457502293200785 \\ 0.22741123141178327 \end{pmatrix}$
v_5	$\begin{pmatrix} 0.13739489147236222 \\ 0.27822138880036196 \\ 0.19061885418384675 \\ 0.39376486554342915 \end{pmatrix}$	$\begin{pmatrix} 0.13739489147236217 \\ 0.27822138880036196 \\ 0.19061885418384666 \\ 0.39376486554342904 \end{pmatrix}$
v_6	$\begin{pmatrix} 0.06562772233681938 \\ 0.4195028782331188 \\ 0.2623860233193896 \\ 0.2524833761106722 \end{pmatrix}$	$\begin{pmatrix} 0.06562772233681935 \\ 0.41950287823311894 \\ 0.2623860233193896 \\ 0.25248337611067223 \end{pmatrix}$

Table 6.1: Marginals of Tree in Figure 6.1.

6.2 Marginals of Graphs with Cycles

For graphs with cycles we ran simulations and compared the decoding decisions of the simulation to that of the global function. We measured the decoding accuracy in terms of bit-errors and word-errors, where any difference between the global decision and the decoded decision was counted as an error. For example, assume the global decision determines that a transmitted codeword was $(0, 0, \omega, \omega)$, while the decoded decision determines the codeword to be $(1, 0, \omega^2, \omega)$. In this case we would count the word error-rate to be 1 and the bit error-rate to be 2.

Decoding accuracy was measured on two graphs with cycles simulation scenarios. In the first situation, we measured decoding performance on the graph in Figure 6.2 under the variation of the number of decoding iterations. In the second, we varied of the number of short cycles in the graph by running simulations on the graphs in Figure 6.3.

Variable: Decoding Iteration. In Figures 6.4 and 6.5 one may find plots of error-rates from simulations of message-passing decoding on the 10-node cycle graph in Figure 6.2. The x-axes represents the number of iterations we allowed each individual decoding to have. The y-axes represent the total number of errors from 10.000 decodings. The y-axis in Figure 6.4 represents bit error-rates, and in Figure 6.5 it represents the word error-rate. We see from both plots that the error-rate decreases when the number of decoding iterations increases.

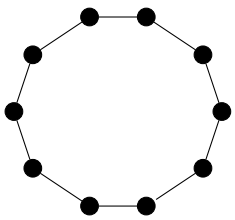


Figure 6.2: Circle graph with 10 nodes.

Variable: Cycles. In Figures 6.6 and 6.7 one may find plots of error-rates from simulations run on the codes represented by the graphs in Figure 6.3. Each data-point corresponds to a graph in Figure 6.3. The x-axes represent the number of 3-node cycles present in the graph, while the y-axes represent

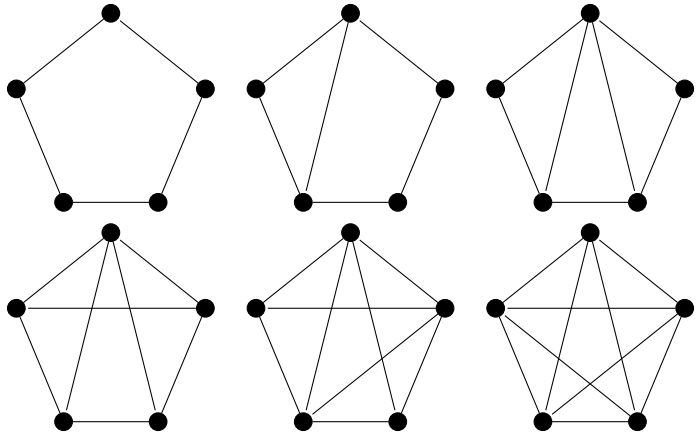


Figure 6.3: Codes of length 5 with increase of triangles.

the total error-rates from 100.000 decodings. Each of the 100.000 simulations were performed with 100 decoding iterations.

In these simulations we see a significant increase in error-rate as the number of short cycles increases. Particularly, from the third graph to the fourth, this extra leap is to be expected due to the difference in minimum distance of the codes these graphs represent. The top three graphs in Figure 6.3 have minimum distance $d = 3$, while the bottom three have a minimum distance $d = 2$.

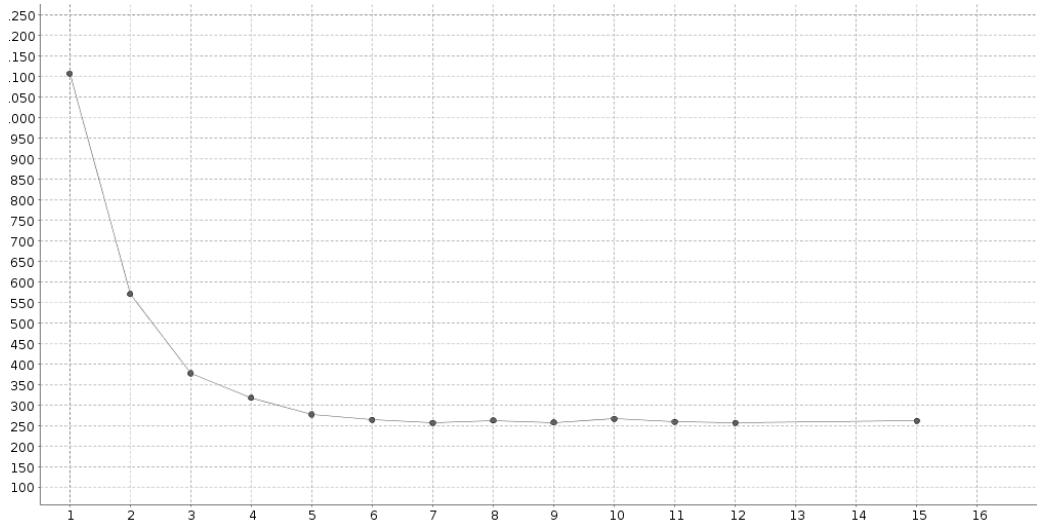


Figure 6.4: Bit error-rate. 8-circle

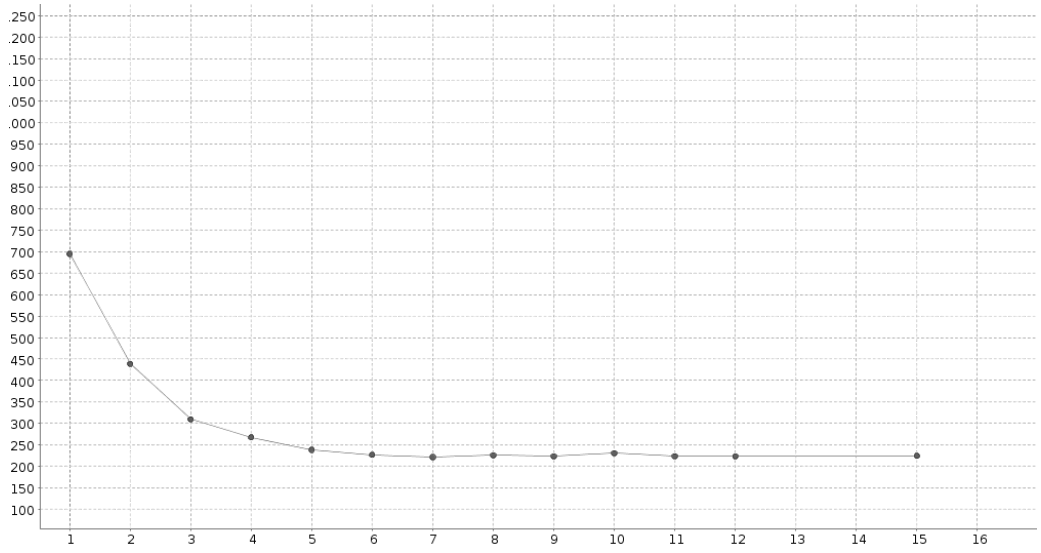


Figure 6.5: Word error-rate. 8-circle

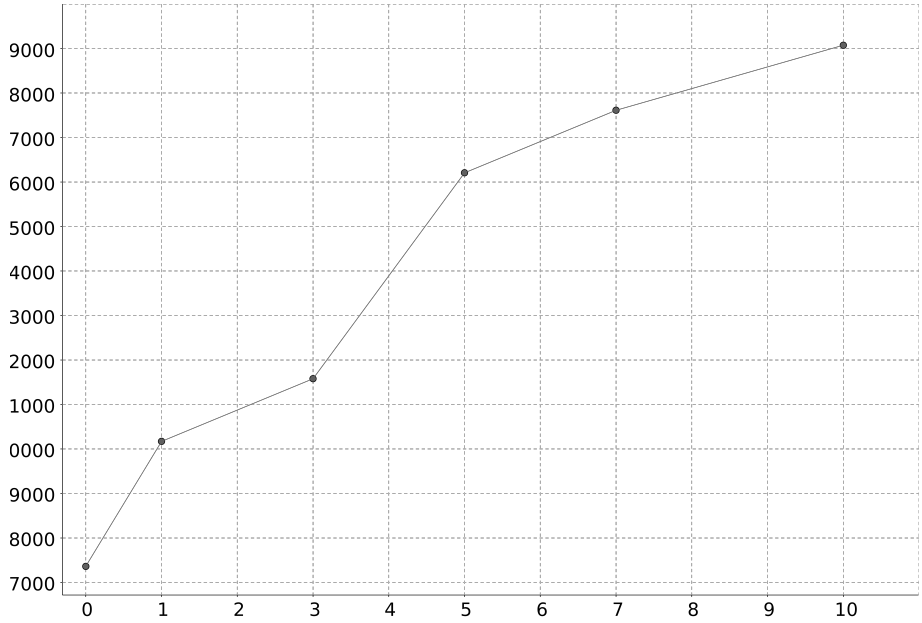


Figure 6.6: Bit error-rate.

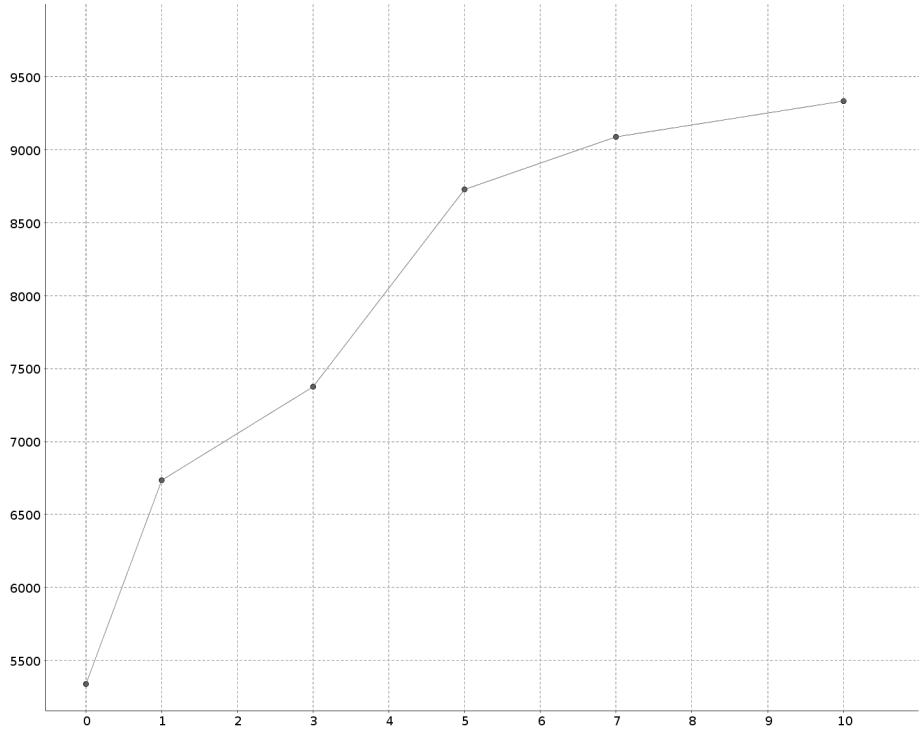


Figure 6.7: Word error-rate.

Chapter 7

Summary

7.1 Conclusions

The original purpose of this master thesis was to verify, by means of software implementation and empirical analysis, a dynamic message-passing decoding scheme on F4-additive codes, self-dual with respect to the Hermitian inner product, described by Parker et al. [17] in an unpublished manuscript. These codes have been shown to be useful for representing quantum error-correcting codes [5].

The dynamic decoding scheme [17] for these codes was based on performing the sum-product algorithm on factor graphs implicitly represented by the graph form (referred to in this thesis as embedded factor graphs). Additionally, the manuscript proposed the use of *local complementation* as a means for dynamically circumventing problems when message-passing on graphs with cycles. Local complementation (LC) on a node $v \in G = (V, E)$, gives us the graph $G * v = (V, E')$ such that for all x and y in the neighborhood of v: $(x, y) \in E \iff (x, y) \notin E'$ — edges not incident on v in E are the same in E' [6, 10]. However, as we show in Chapter 3, we found the method of decoding on embedded factor graphs to be unsound. The proceeding work was aimed at finding a replacement method for the embedded factor graph decoding.

We developed in Chapter 4 the Discriminative decoding procedure, which performs message-passing decoding on graphs constructed from the adjacency matrix of self-dual $\mathbb{F}_4$-additive codes in graph form. Its local operations may collectively be described as various recursive applications of the vector

products introduced in this thesis — $dSS(u, v)$, $dSX(u, v)$, $tSX(u, v)$ — and the pointwise product. We saw that messages sent by leaves and internal nodes required the use of different vector products, hence the name *Discriminative decoding*. Furthermore, we showed that the method computes the global marginals for any self-dual $\mathbb{F}_4$-additive code represented by a tree in graph form. This also implies that Discriminative decoding is an instance of the sum-product algorithm for the special case of self-dual $\mathbb{F}_4$-additive codes in graph form.

In Chapter 5, we described a decoding simulation tool developed for empirical analysis and hypothesis testing during the development of our decoding procedure. This tool can simulate both Embedded decoding and Discriminative decoding over the Quaternary Symmetric Channel. Its modular design makes it easy to extend with another channel model or graph representation, and it may serve as aid to future development of the dynamic decoding scheme.

The development of the Discriminative decoding procedure is the main contribution of this thesis. We argue that it serves as a good foundation for dynamic decoding [17] on self-dual $\mathbb{F}_4$-additive codes represented by graphs. An especially interesting class of graphs are "nested cliques" [6], which we discuss in Section 7.2.

7.2 Future Work

7.2.1 Dynamic Decoding with Local Complementation

Several self-dual $\mathbb{F}_4$-additive codes with high minimum distance have been shown [8] to be represented by graphs with highly regular "nested clique" structures [6, Definition 4.1], where length $n \leq 12$. It would be very interesting to perform simulations of Discriminative decoding on them. These codes are desirable not only because they are significantly stronger than the ones we decoded on in this thesis, but also because nodes of their graph representations have the smallest neighborhoods possible without compromising the high minimum distance [7, page 17]. Unfortunately, the computation of marginals on "nested clique" graphs may be disturbed by the many short cycles they contain.

It is known that two codes are equivalent, up to re-labeling, if their graphs are equivalent with respect to local complementation [9, Definition 4, Theorem 11]. For future work, we propose that the dynamic component of the Embedded

decoding scheme described by Parker et al. be coupled with the Discriminative decoding scheme for empirical analysis by means of the simulation tool from Chapter 5. This involves conceiving a *local* heuristic such that nodes may decide based upon messages from their neighbors when it is necessary to initiate local complementation on themselves.

7.2.2 Decoding with the General Procedure

Although, it can be beneficial to distinguish between leaf messages and internal messages due to the context in which Discriminative decoding may be used, it is possible to decode without this distinction. We propose Algorithms 7, 8, 9, and 10, as description of a potential general algorithm for decoding on self-dual $\mathbb{F}_4$-additive codes, where discrimination between messages from leaves and internal nodes is absent. We conjecture it may be proven that message-passing decoding using these algorithms also exactly computes the global marginals for trees. Extending the simulation tool from Chapter 5. with a graph that uses these procedures for message-passing would be interesting. Furthermore, coupling it with a dynamic LC component from the previous section could be fruitful enquiry.

Algorithm 7 Pass a message.

Let $s = (e, f, g, h)$ be the soft information of the sender.

```
1: procedure passMessage(Node recipient)
2:     if Neighborhood.size() == 1 then
3:         message ← (e, g, f, h)
4:         recipient.receiveMessage(message)
5:     else
6:         inboxProduct ← inboxProduct(recipient)
7:         message ← dSS(inboxProduct, s)
8:         recipient.receiveMessage(message)
9:
```

Algorithm 8 Compute marginals.

Let $s = (e, f, g, h)$ be the soft information from the channel.

1: **procedure** MARGINALIZE
2: inboxProduct $\leftarrow$ inboxProduct()
3: marginals $\leftarrow \cdot(inboxProduct, s)$
4:

Algorithm 9 Compute the inbox-product of all messages, except for Node recipient.

M is the set of all received messages. $R \subseteq M$ is the set of messages from the recipient.

1: **function** INBOXPRODUCT(Node recipient)
2: $M' \leftarrow M \setminus R$
3: $product \leftarrow m'_0$
4: **for all** $m'_i \in M'$, where $i > 0$ **do**
5: $product \leftarrow dSX(m'_i, product)$
6:
return *product*

Algorithm 10 Compute the inbox-product of all messages.

M is the set of all messages from neighbors.

1: **function** INTERNALPRODUCT
2: $product \leftarrow m_0$
3: **for all** $m_i \in M$, where $i > 0$ **do**
4: $product \leftarrow dSX(m_i, product)$
5:
return *product*

Bibliography

[1] Java language specifications: Chapter 15.17 multiplicative operators. `http://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.17`. Last accessed: May 30th 2016, 1200 am.

[2] Java language specifications: Chapter 15.18 additive operators. `http://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.18`. Last accessed: May 30th 2016, 1200 am.

[3] S. B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, 27(6):509–516, June 1978.

[4] A. Bouchet. Graphic presentations of isotropic systems. *Journal of Combinatorial Theory, Series B*, 45(1):58 – 76, 1988.

[5] A. R. Calderbank, E. M. Rains, P. W. Shor, and N. J. A. Sloane. Quantum error correction via codes over GF(4). 1996.

[6] L. E. Danielsen. On self-dual quantum codes, graphs, and boolean functions. Master's thesis, Department of Informatics, University of Bergen, Norway, 2005.

[7] L. E. Danielsen. *On Connections Between Graphs, codes, Quantum States, and Boolean Functions*. PhD thesis, University of Bergen, Norway, 2008.

[8] L. E. Danielsen and M. G. Parker. Spectral orbits and peak-to-average power ratio of boolean functions with respect to the {I,H,N}ˆn transform. *CoRR*, abs/cs/0504102, 2005.

[9] L. E. Danielsen and M. G. Parker. On the classification of all self-dual additive codes over GF(4) of length up to 12. *J. Comb. Theory, Ser. A*, 113(7):1351–1367, 2006.

[10] H. de Fraysseix. Local complementation and interlacement graphs. *Discrete Mathematics*, 33(1):29 – 35, 1981.

[11] R. Diestel. *Graph theory*. Graduate texts in mathematics. Springer, New York, 3rd electronic edition edition, 2005.

[12] J. B. Fraleigh. *A First Course in Abstract Algebra*. Pearson Education, 7th edition, 2014.

[13] R. G. Gallager. *Low-Density Parity-Check Codes*. PhD thesis, 1963.

[14] W. C. Huffman and V. Pless. *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 2003. Cambridge Books Online.

[15] J. G. Knudsen. *On Iterative Decoding of High-Density Parity-Check Codes Using Edge-Local Complementation*. PhD thesis, University of Bergen, Norway, 2010.

[16] F. R. Kschischang, B. J. Frey, and H. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Trans. Information Theory*, 47(2):498–519, 2001.

[17] M. G. Parker. Dynamic message-passing decoding on simple graphs for the quaternary symmetric channel.

[18] E. M. Rains and N. J. A. Sloane. Self-Dual Codes. *ArXiv Mathematics e-prints*, July 2002.

[19] T. Richardson and R. Urbanke. *Modern Coding Theory*. Cambridge University Press, New York, NY, USA, 2008.

[20] W. E. Ryan and S. Lin. *Channel codes: classical and modern*. Cambridge Univ. Press, Leiden, 2009.

[21] D. Schlingemann. Stabilizer codes can be realized as graph codes. *Quantum Information & Computation*, 2(4):307–323, 2002.

[22] T. M. Thompson. *From Error-Correcting Codes Through Sphere Packings to Simple Groups*. The Mathematical Association of America, 1983.

[23] J. S. Yedidia. Message-passing algorithms for inference and optimization. *Journal of Statistical Physics*, 145(4):860–890, 2011.