Belief Propagation in Self-Dual $\mathbb{F}_4$-Additive Codes Utilizing Local Complementation

Åsmund Hammer

Master's thesis in Software Engineering at

Department of Computing, Mathematics and Physics,
Western Norway University of Applied Sciences

Department of Informatics,
University of Bergen

June 2019

Western Norway
University of
Applied Sciences

Abstract

In this thesis we show how one can decode self-dual additive codes over $\mathbb{F}_4$ using belief propagation. We then develop an extension to the algorithm with steps of local complementation. We show empirical evidence that the extension using local complementation achieves lower error rates for the same soft information compared with the algorithm without the extension. We discuss how the algorithmic parameters impact runtime and decoding performance.

Acknowledgements

I would like to thank my supervisors Pål Ellingsen, Constanza Susana Riera, and Matthew Geoffrey Parker for their invaluable insight and support. Thanks to my friends and family for their support and encouragement.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

1.1 Background

Digital communication has become an increasingly important concept for today's society. Data needs to be transmitted over a vast number of channels, and stored in various formats such as hard drives or flash memory. In all these scenarios the data is susceptible to corruption due to environmental noise. For most high-speed wired networking where errors can occur, the cost of transmission is low, and any corrupted message can simply be re-transmitted. However, the higher the cost of transmission, the more important it is to get it right the first time. This is especially the case in space technology where transmissions can be costly both in travel time of the signals, and in the electrical cost of running the transmitters. Most media used for storing data are susceptible to component failures which can result in data corruption.

In scenarios like these, the use of error correcting codes [16] can be necessary as it provides a defence layer against bit errors, that is more efficient than simply repeating the same information over and over again. In error correcting codes, the idea is to add some redundancy so that messages with some errors can still be decoded and interpreted as the original message.

Low-density parity check (LDPC) codes [11], and turbo codes [4] are examples of such error correcting codes. These codes can be decoded using graph structures, where each node takes care of a bit of received information, assigns to it the likelihood that it is correct, and communicates it to its neighbouring nodes, which do the same. The goal is for the algorithm to converge towards what is believed to be the most likely codeword to have been transmitted. This is known as belief propagation, or sum-product message-passing [14].

1.2 Decoding in $\mathbb{F}_4$

In their manuscript "*Dynamic Message-Passing Decoding on Simple Graphs for the Quaternary Symmetric Channel*" [18], Matthew G. Parker *et al.* present ideas for applying practices from error correcting codes on to a non-binary system, the Galois Field $\mathbb{F}_4 = \{0, 1, w, w^2\}$, with the assumption that the codewords in $\mathbb{F}_4$ are transmitted over a quaternary symmetric channel. Just as in binary channels, any $b \in \mathbb{F}_4$ transmitted will have a probability of being received as $a \in \mathbb{F}_4$, where $a \neq b$, a "bit" error. Parker *et al.* suggests that the belief propagation in self-dual $\mathbb{F}_4$-additive codes (graph codes) could be structured using the simple graphs associated to this type of codes [9]. These graphs are not necessarily bipartite (as those used in LDPC codes), and the

idea is that all nodes play a similar role instead of using distinct variable and check nodes. In the master thesis "*Message-Passing decoding on Self-Dual $\mathbb{F}_4$-additive codes*" by Hannah Hansen [12], decoding in $\mathbb{F}_4$ is explored and the decoding method *discriminative decoding* is introduced. The algorithm computes products, messages and marginals based on whether the nodes are leaves or internal nodes. The discriminative decoder is shown to compute exact marginals for trees and to be an instance of the sum-product algorithm. Though capable of decoding the marginals of trees exactly, the algorithm was shown to run into problems when performed on graphs containing short cycles. Since the graph codes that form trees have a small minimum edit distance $d \leq 2$, they are not optimal for decoding purposes. It is therefore desirable to improve the decoding performance on graphs containing cycles, as it has been shown that many self-dual $\mathbb{F}_4$-additive codes of higher distance have short cycles as in the case of nested-cliques [6][8]. In the future work section of [12], it is suggested to try decoding in a non-discriminate way where one does not differentiate between leaves and internal nodes, and some algorithms are suggested as to how it might be done. One of the goals of this thesis is to elaborate on how one can decode self-dual $\mathbb{F}_4$-additive codes in a non-discriminate way. In addition to this, in the previous work on message-passing in $\mathbb{F}_4$, it has been suggested to use local complementation in order to deal with decoding problems with regard to short-cycled graphs [18][12]. It is the goal of this thesis to outline a belief propagation algorithm for simple graphs, and extend it with steps of local complementation. With this, we can gather empirical data on the decoding performance with regards to short cycles, and general self-dual $\mathbb{F}_4$-additive codes. The study of self-dual $\mathbb{F}_4$-additive codes has been motivated by their interpretation as quantum stabilizer codes [6]. As the definitions of quantum codes is not necessary for the strict purpose of decoding in $\mathbb{F}_4$, we do not go into detail about them in this thesis.

1.3 Thesis Outline

Section 2 outlines some concepts of error correcting codes central to this thesis. In Section 3, we first develop algorithms for Belief Propagation for Simple Graphs (BPSG), based on previous work presented in [12]. Due to the problems this algorithm runs into when dealing with cycles, we develop the algorithm Iterative Local Complementation (ILC), that utilizes other graphs in the LC-orbit of the code. In Section 4, we compare the performance of BPSG and ILC, look at how their parameters impacts performance, and see how a single LC operation impacts decoding in specific graph-structures. We end the thesis by concluding and showing some prospects for future work.

2 Theorethical Background

In this section we give a brief summary of concepts from algebra and coding theory that are central to this thesis. Though our descriptions of these concepts are brief, there exists plenty of books and papers that can elaborate on the topics at hand. We begin by introducing groups and fields and continue with general coding-theory, graph-theory, and related decoding-algorithms. Self-dual $\mathbb{F}_4$-additive codes and their properties relating to *local complementation* are of particular interest in this thesis, and are presented in Sections 2.4 and 2.6.

2.1 Groups and fields

We will develop decoding algorithms for self-dual additive codes over the Galois Field $\mathbb{F}_4$. It is therefore necessary that the reader has a basic understanding of the the concepts of groups and fields. We provide a brief description of groups and rings based on their definition in [10].

Definition 2.1. *A group $\langle G, * \rangle$ is a set G closed under the binary operation $*$, such that the following three axioms are satisfied:*

g_1 : $\forall a, b, c \in G$, $(a * b) * c = a * (b * c)$. *($*$ is associative).*

g_2 : *There is an element $e \in G$ such that $e * x = x * e = x$ for all $x \in G$. (identity element e).*

g_3 : *For all elements $a \in G$, there exists an element $a' \in G$ such that $a * a' = a' * a = e$ (inverse a' of a)*

An *abelian group* is a group where $a * b = b * a$, i.e. $*$ is commutative.

Definition 2.2. *A ring $\langle R, +, \cdot \rangle$ is a set R with addition and multiplication defined on R such that the following axioms are satisfied:*

r_1**:** $\langle R, + \rangle$ *is an abelian group.*

r_2 : *Multiplication is associative.*

r_3 : *For all $a, b, c \in R$, $a \cdot (b+c) = (a \cdot b) + (a \cdot c)$ and $(a+b) \cdot c = (a \cdot c) + (b \cdot c)$, the left and right distributive laws.*

A ring R with multiplicative identity element $e = 1$, and where all non-zero elements have a multiplicative inverse in R is called a *division ring*. A division ring where multiplication is commutative is called a *field*. A *Finite*

field, or *Galois field* is a field with a finite set of elements. $GF(n)$ or $\mathbb{F}_n$ denotes a Galois field with n elements. For any prime integer p, $GF(p)$ is the field of integers $\mathbb{Z}_p$ with operations modulo p. We can construct the Galois field $GF(p^m)$ given that we have a irreducible polynomial $\pi(x)$ of degree m over $GF(p)$. [16]

Theorem 2.1. *Let $\pi(x)$ be an irreducible polynomial over $GF(p)$ that has degree m. The set of all polynomials in x of degree $< m$ and coefficients from $GF(p)$, with calculations performed modulo $\pi(x)$, forms a field of order p^m [16].*

In this thesis we will primarily focus on codes in the Galois field $\mathbb{F}_4$= $\{0, 1, \omega, \omega^2\}$. This finite field is defined to consist of all polynomials in x with binary coefficients and degree at most 1, with calculations performed modulo the irreducible polynomial $\pi(x) = x^2 + x + 1$ [16]. We use ω to denote a root of $x^2 + x + 1$ in $\mathbb{F}_4$. Then, $\omega^2 = \omega + 1$ and all elements in $\mathbb{F}_4$ can be written as $a + b\omega$, where $a, b \in \mathbb{F}_2$ and addition is binary (XOR).

2.2 Linear codes

A code C is a set of strings over a certain alphabet A. A codeword c is any one of the strings $c \in C$. A *block code* is a code where all codewords have the same length n. We denote the set of all n-length touples $c = (c_0, c_1, ..., c_{n-1})$, with $c_i \in A$, to be A^n. A block code C with n-length words is then a subset of A^n. The edit distance or *distance* between two codewords c and c' is the minimum amount of characters you have to change in c in order to get c'. The minimum edit distance of a code C is the minimum distance between any codewords $c, c' \in C$. A *linear code* C of dimension k is a linear subspace of $\mathbb{F}_q^n$. These codes can be described by a *generator matrix* $G \in \mathbb{F}_q^{k \times n}$ whose rows span C. These codes can also be described by a parity check matrix H such that $C = \{c \in \mathbb{F}_q^n \mid Hc^\top = 0\}$ [16], where $c^\top$ denotes the transpose of c such that $c^\top$ is a single column matrix of size $n \times 1$. We can therefore use the parity check matrix H to check whether a given string c is in C by checking if $Hc^\top = 0$. A *dual* code $C^\perp$ of C is a code consisting of all words orthogonal to the words in C. So $C^\perp = \{\hat{c} | \hat{c} \cdot c = 0, \forall c \in C\}$, where $\cdot$ is the dot product for vectors. A *self-dual* code C is then a code where all codewords are orthogonal to each other. An *additive* code C is a code where any sum of codewords is a codeword. We are particularly interested in self dual additive codes over $\mathbb{F}_4$, due to how they can be represented by graph structures [9]. This is outlined in Section 2.4

2.3 Graphs

A graph $G = (V, E)$ is a set V of n *vertices* (also called *nodes*) v_i for $i \in \{0 \ldots n-1\}$ together with a set $E \in V \times V$ of edges (v_i, v_j) representing that there is an edge between the vertices v_i and v_j. If there is an edge between two nodes v_i and v_j, they are connected and we call them *neighbours*. We call the set of all nodes having an edge with v_i the *neighbourhood* $\mathcal{N}_{v_i}$ of v_i. A graph with n vertices can be represented by an adjacency matrix Γ of size $n \times n$, where entry $\Gamma_{ij} = 1$ if there is an edge between v_i and v_j [7]. We can illustrate the graphs with circles representing vertices, and lines between the circles representing edges. The graph $G = (V, E)$ where $V = \{v_0, v_1, v_2, v_3\}$ and $E = \{(v_0, v_1), (v_0, v_2), (v_1, v_2), (v_2, v_3)\}$ is visualized in Figure 1.

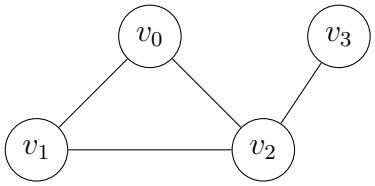


Figure 1: A simple graph

A *cycle* is a set of vertices and edges such that $\{(v_i, v_j), (v_j, v_h), ..., (v_x, v_i)\} \in E$. v_0, v_1 and v_2 forms a cycle in Figure 1. A graph that contains no cycles is called a *tree*. We refer to nodes that only have a single neighbour as *leaves*, while we refer to nodes that have more than a single neighbour as *internal nodes*. Figure 2 shows a tree with $\{v_0, v_1, v_2\}$ as internal nodes and $\{v_3, v_4, v_5, v_6\}$ as leaf nodes. Other graph structures of interest in this thesis

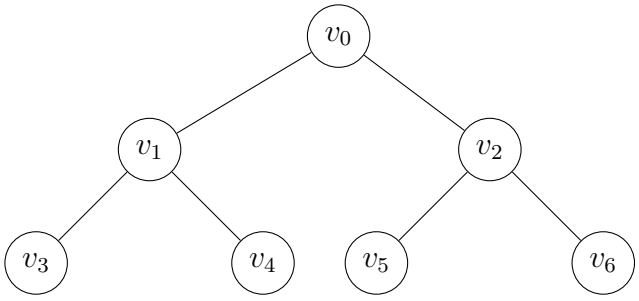


Figure 2: A tree containing 3 internal nodes and 4 leaves

are *cliques* and *nested-cliques*. A *clique* is a graph where all vertices connect to all other vertices. A triangle or a square with edges across the two diagonals are simple examples of cliques. *Nested-cliques* are graph-structures comprised of x cliques where each clique has y vertices, and each vertex

neighbours all other nodes in its local clique as well as one node in all other cliques. Figures 3 and 4 show a clique and a nested-clique respectively. The nested-clique of six vertices has two cliques forming a triangle, where each vertex also neighbours one vertex in the opposite triangle.

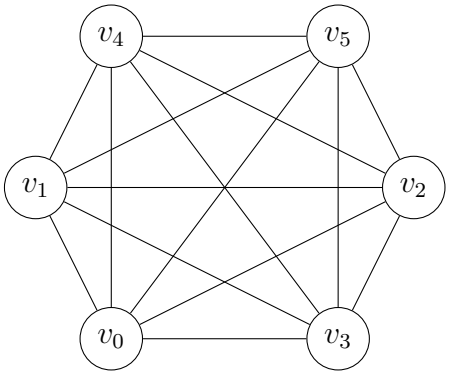


Figure 3: Six vertices with edges forming a clique

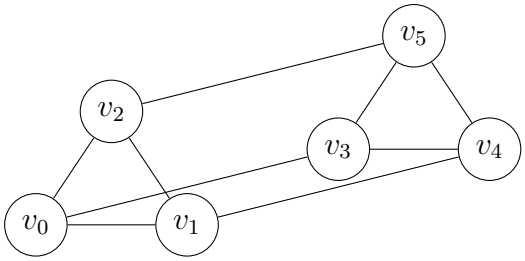


Figure 4: Six vertices with edges forming a nested-clique

2.4 Graph Codes

This thesis studies the decoding of graph codes using belief propagation. The study of these codes have been motivated by their interpretation as quantum stabilizer codes [6]. This thesis focuses strictly on the decoding of graph codes, and we leave out descriptions of their relation to quantum codes.

Definition 2.3. *The hermitian inner product [12] is defined as* $\star(\vec{u}, \vec{v})$: $\mathbb{F}_4^n \times \mathbb{F}_4^n \to \mathbb{F}_2$*:*

$$\vec{u} \star \vec{v} = \sum_{i=0}^{n-1} u_i v_i^2 + u_i^2 v_i \ (mod\ 2)$$

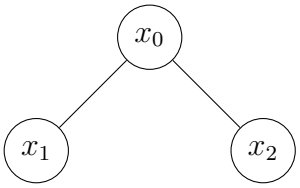


Figure 5: Graph corresponding to Γ

A code C generated by adding the rows of a generator matrix $G : \mathbb{F}_4^{n\times n}$ is an additive code over $\mathbb{F}_4$. The dual code is $C^\perp = \{\vec{u} \in C^\perp \mid \vec{u} \star \vec{c} = 0 \ \forall c \in C\}$. If $C = C^\perp$, then C is a self-dual $\mathbb{F}_4$-additive code with respect to the hermitian inner product.

Definition 2.4. *A graph code is a self-dual additive code over $\mathbb{F}_4$ that has a generator matrix of the form $C = \Gamma + \omega I$, where I is the identity matrix, and Γ is the adjacency matrix of a simple undirected graph [6].*

It has been shown that all graph codes are self-dual, and that every self-dual additive code over $\mathbb{F}_4$ is equivalent to a graph code [9]. Another feature of these self-dual $\mathbb{F}_4$-additive codes is that their generator matrix is equal to their parity check matrix.

Exmple. *For the self-dual $\mathbb{F}_4$-additive code C with parity check matrix H, it can be represented on the form $\Gamma + \omega I$ by:*

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

We can construct the corresponding graph from the adjacency-matrix Γ, which can be seen in Figure 5.

2.5 Sum-Product Algorithm

The sum-product algorithm (SPA) is used to solve problems that deal with large functions of many variables by factorizing the "global"-function into the product of smaller functions that are easier to use in computations. Algorithms such as Markov random fields, Bayesian network, and iterative decoding of LDPC and turbo codes can be shown to be instances of the sum-product algorithm.[14].

The SPA takes advantage of factorization of a global function on many variables. For the variables $x_0, x_1, ..., x_{n-1}$, where each x_i is in some alphabet A_i, the global function $g : A_0 \times A_1 \times ... \times A_n \to \mathbb{R}$ is a function that takes a configuration of the values of the variables, and outputs a value in $\mathbb{R}$. For each global function $g(x_0, x_1, ..., x_n)$, we can associate n marginal functions $g_i(x_i)$, where for $a \in A$, $g_i(a)$ is a the sum of all configurations of $g(x_0, x_1, ..., x_n)$ that has $x_i = a$. We call $g_i(x_i)$ the marginal for x_i. As an example, for $g : A_0 \times A_1 \times A_2$, the marginal $g_1(x_1)$ for x_1 can be calculated by

$$\sum_{\sim x_1} g(x_0, x_1, x_2) := \sum_{x_0 \in A_0} \sum_{x_2 \in A_2} g(x_0, x_1, x_2)$$

Where we use the notation $\sim x$ to symbolize summarizing over everything that is not x. In general, the summary over a variable x_i can be defined as:

$$g(x_i) := \sum_{\sim x_i} g(x_0, x_1, ..., x_n) \text{ [15]}$$

2.5.1 Global function marginal computation

Let C be a code with n codewords of length k in some alphabet A. Let each x_i contain a probability vector p_i of length $|A|$ corresponding to the probabilities of x_i being each of the characters in A. We can then calculate the probability of any x_i being $a_j \in A$ to be the sum of all configurations of $g(x_0, ..., x_{k-1})$ that has $x_i = a_j$.

As an example, consider the self-dual additive code over $\mathbb{F}_4$ represented by the generator and parity check matrix H:

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

This code has 8 codewords of length 3, which we can generate from the matrix H: $C = \{000, \omega 11, 1\omega 0, 10\omega, \omega^2\omega^2 1, \omega^2 1\omega^2, 0\omega\omega, \omega\omega^2\omega^2\}$

Let p_0, p_1, p_2 be the probability vectors we associate with each variable x_i:

$$p_i = \begin{pmatrix} P(x_i = 0) = c_i \\ P(x_i = 1) = d_i \\ P(x_i = \omega) = e_i \\ P(x_i = \omega^2) = f_i \end{pmatrix}$$

c_i corresponds to the probability that x_i is 0, and d_i, e_i, f_i corresponds to the probabilities of x_i being $1, \omega, \omega^2$ respectively. We can find the most likely

codeword from the global function by multiplying together the probabilities of the variables being the arrangements of the codewords. For example, the probability of the codeword being 000 will be $c_0c_1c_2$, and the probability that it is $0\omega\omega$ is $c_0e_1e_2$. If we want the probability of a single node x_i being 0 we can take the sum of all arrangements where $x_i = 0$. $P(x_0 = 0)$ is then the sum of two products as there are only two codewords with $x_0 = 0$. In these codewords, $x_0 = 0 \Rightarrow x_1 = x_2 = 0$ OR $x_1 = x_2 = \omega$. We get $P(x_0 = 0) = c_0c_1c_2 + c_0e_1e_2$. Using the same logic for $P(x_0 = 1), P(x_0 = \omega)$ and $P(x_0 = \omega^2)$ we get the entire marginal m_0 for x_0:

$$m_0 = \begin{pmatrix} c_0c_1c_2 + c_0e_1e_2 \\ d_0e_1c_2 + d_0c_1e_2 \\ e_0d_1d_2 + e_0f_1f_2 \\ f_0f_1d_2 + f_0d_1f_2 \end{pmatrix} = \begin{pmatrix} c_0(c_1c_2 + e_1e_2) \\ d_0(e_1c_2 + c_1e_2) \\ e_0(d_1d_2 + f_1f_2) \\ f_0(f_1d_2 + d_1f_2) \end{pmatrix}$$

Computing marginals directly from the global function can be done for small codes such as in this example, but the amount of computations needed in order to get the marginals of the nodes gets out of hand quickly as the codes get larger. Self-dual $\mathbb{F}_4$-additive codes with generator matrix of size $n \times n$ consists of 2^n codewords. This means that solving the marginals for a code with generator matrix of size $(n+1)\times(n+1)$ is twice as computationally heavy as for a code with generator matrix of size $n \times n$.

2.5.2 Factor graph decoding

A *factor graph* is a representation of the factorization of a global function in graph form. The factor graphs are structures that consists of variable nodes, and factor nodes forming a bipartite graph with connections only between a factor node and a variable node. Let's say we have a function $g(x_0, x_1, x_2, x_3, x_4)$ that can be factorized to:

$$f_0(x_0, x_1)f_1(x_1, x_2, x_3)f_2(x_3, x_4)$$

We can then construct a factor graph from the factorization, where each factor node is connected to its corresponding variable node. A factor graph corresponding to the factorization of g can be seen in Figure 6.

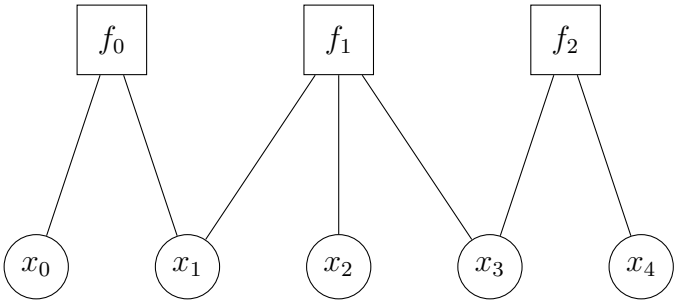


Figure 6: A factor graph with 3 factor nodes, and 5 variable nodes

One can use factor graphs for decoding codewords in a binary linear code C based on the parity check matrix H of C. This can be done by having the rows H represent factor nodes, and having the columns represent the variables, where having 1 in position h_{ij} means that there is an edge between factor node f_i and variable node x_j. The graph in Figure 6 corresponds to the parity check matrix:

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

Error correction can be done in the factor graph by having the variable nodes contain the probabilities or *soft information* $P(x_j = 0)$ and $P(x_j = 1)$, and the factor nodes contain logic for which arrangements of the variable nodes constitute a codeword. The decoding process can then be done by means of *message-passing*, also called *belief propagation*. Message-passing is done by having the nodes pass messages regarding the soft information to their neighbours, which propagate it throughout the graph. Factor nodes pass messages to the variable nodes containing their belief about the variable, and the variable nodes pass messages regarding what they believe about themselves. A key property in belief propagation is that, when nodes create a message for one of their neighbours, they do not take into consideration any message from that neighbour. This is done so that the neighbour in question does not get their original beliefs falsely amplified by confirmation from their neighbour. In Figure 6, if f_1 is to send a message to x_3, it will calculate it by using the messages from x_1 and x_2 together with the logic for what constitutes a codeword. If x_3 is to send a message to f_1 it will be the dot-product of its original soft information, and the probabilities from f_2. The message-passing can be done in different ways depending on whether the factor-graph is a tree. If it is a tree, we can simply select a node as root, propagate messages from the leaves and up to the root, and then propagate

messages back from the root to the leaves. This will calculate exact marginals for the variable nodes based on the soft information. However, for codes that contain cycles, we can not propagate messages in a simple forward-backward manner. Instead a *flooding scheme* can be applied to graphs with cycles. The flooding scheme uses some amount of *flooding iterations* where for each iteration, one message is passed in each direction for every edge in the graph. Decoding in this manner has been shown to be quite successful, but it can not be guaranteed that the scheme will converge towards the exact marginals [15].

2.6 Local Complementation

Local complementation (LC) is a graph-operation that alters the structure of the graph. Experiments of decoding binary linear codes using a combination of LC operations was conducted in the PhD thesis "*On iterative decoding of high-density parity check codes using edge-local complementation.*" by Joakim G. Knudsen[13]. The thesis showed that one could use LC to aid decoding by introducing diversity in the graph. In this section we describe LC in relation to self-dual $\mathbb{F}_4$-additive codes.

Given a node v with the set of neighbours $\mathcal{N}_v$, performing LC on v, is for every pair of neighbours $(x_i, x_j) \in \mathcal{N}_v$, checking whether the edge (x_i, x_j) exists. If the edge exists, it is removed, and if it does not, it is added to the graph. An illustration of an LC operation can be seen in Figure 7. There, x_0 neighbours the nodes x_1, x_2, x_3. Performing LC on x_0 has the effect that, since there is no edge (x_1, x_2), it is added. The edges (x_1, x_3) and (x_2, x_3) exist and are therefore removed. The corresponding parity check matrix H for the graph code corresponding to G looks as follows:

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

We can perform LC directly on these parity check matrices by performing a series of matrix-operations. Doing $LC_v(H)$ can be done by three operations. (i): add row v to all rows $i \in \mathcal{N}_v$. (ii): swap ω^2 with 1 in column v. (iii): swap ω^2 with ω in all columns $i \in \mathcal{N}_v$. We can see that by doing $LC_0(H)$ we get H', representing a graph code corresponding to G' in Figure 7.

$$H = \begin{pmatrix} \omega & 1 & 1 & 1 \\ 1 & \omega & 0 & 1 \\ 1 & 0 & \omega & 1 \\ 1 & 1 & 1 & \omega \end{pmatrix} \; (i) \rightarrow \begin{pmatrix} \omega & 1 & 1 & 1 \\ \omega^2 & \omega^2 & 1 & 0 \\ \omega^2 & 1 & \omega^2 & 0 \\ \omega^2 & 0 & 0 & \omega^2 \end{pmatrix} \; (ii) \rightarrow \begin{pmatrix} \omega & 1 & 1 & 1 \\ 1 & \omega^2 & 1 & 0 \\ 1 & 1 & \omega^2 & 0 \\ 1 & 0 & 0 & \omega^2 \end{pmatrix}$$

$$(iii) \rightarrow \begin{pmatrix} \omega & 1 & 1 & 1 \\ 1 & \omega & 1 & 0 \\ 1 & 1 & \omega & 0 \\ 1 & 0 & 0 & \omega \end{pmatrix} = LC_0(H) = H'$$

It has been shown that any graph codes C and C' are equivalent given that C' has been obtained by a series of LC operations on C [9]. We refer to the set of all graphs obtainable by a series of LC operations on G including G itself, as the *LC-orbit* of G. All graphs in an LC-orbit are considered to be *LC-equivalent*. Another property of LC is that applying LC twice on node v in the graph G results in the original graph G. In other words, $LC_v(LC_v(G)) = G$.

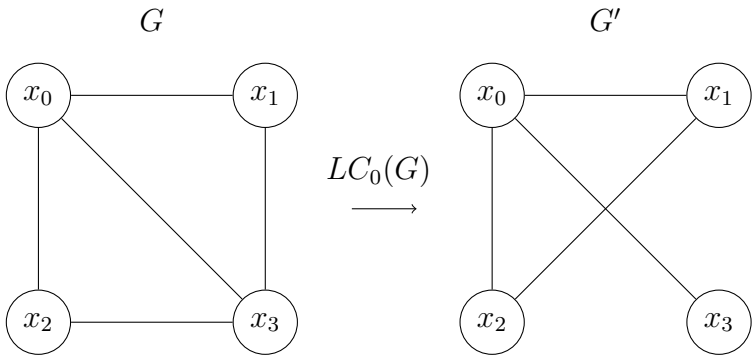


Figure 7: Performing $LC_0(G)$, resulting in G'

2.7 Additive white gaussian noise

In order to sample data of the performance of our decoding algorithms, we need to simulate a noisy channel. For this we use an *additive white gaussian noise* (AWGN) channel to act as noise on our code-bits. Using binary shift key modulation [3], each bit is transformed into a signal $t_i = (2c_i - 1)\sqrt{E_b}$ where E_b is the bit energy. This means that 0 has the signal $-\sqrt{E_b}$ and 1 has signal $-\sqrt{E_b}$. We can add to this signal some noise n form a normal distribution $\mathcal{N}(0, \sigma^2)$, with $\sigma^2 = N_0/2$ where N_0 represents noise. The

final received signal for bit i would be the bit-signal with the added noise $r_i = t_i + n_i$. Assuming $P(c_i = 0) = P(c_i = 1) = \frac{1}{2}$, we get the probability function:

$$P(c_i = 1|r_i) = \frac{1}{1 + e^{-2\sqrt{E_b}r_i/\sigma^2}} \text{ [17]}$$

Which of course gives:

$$P(c_i = 0|r_i) = 1 - P(c_i = 1|r_i)$$

For any given noise N_0 and bit-energy E_b we can then calculate the two probabilities by getting n_i from a corresponding normal distribution $\mathcal{N}(0, \sigma^2)$. A normal way to sample n_i from a normal distribution in software is to use the Box-Muller method [5].

3 Design and Implementation

For developing a decoding algorithm for self-dual $\mathbb{F}_4$-additive codes or *graph codes*, we extend on the previous work of Hansen in the thesis "Dynamic message-passing decoding on simple graphs for the quarternary symmetric channel"[12]. Her thesis explores two different decoding algorithms, and finds that a *discriminative decoding scheme* is successful for computing marginals exactly for trees. The algorithm computes messages and marginals differently based on whether the nodes are leaves or internal nodes, hence the name *discriminative*. Though successful at decoding in graphs that are trees, and graphs that have few cycles, the algorithm struggles to correct errors when there are cycles, particularly short cycles such as local cliques in the graph. In the future work section of Hansen's thesis, it is suggested to try decoding using a general procedure that does not discriminate between internal nodes and leaf nodes. In this section we define belief propagation for simple graphs (BPSG) based on the algorithms from the future work section of [12]. We begin by reducing the marginals of nodes to a series of vector products, and show how we can get those marginals by means of mesage-passing (BPSG). Due to BPSG having the same issues as discriminative decoding, in that it struggles to correct errors in graphs containing cycles, we develop an extension to BPSG using iterative local complementation (ILC). We then show an example where we go though each step of the belief propagation using ILC.

3.1 Marginal computation in graph codes

For the decoding of graph codes using BPSG, we use three vector products. The normal dot product as well as two other products that were introduced in Hansen's thesis [12]. These products are defined as follows:

Definition 3.1. *The dot product is defined as* $\cdot(u,v) : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$*:*

$$\cdot(u,v) = \begin{pmatrix} u_0v_0 \\ u_1v_1 \\ \vdots \\ u_{n-1}v_{n-1} \end{pmatrix}$$

Definition 3.2. *We define the divided-straight-straight product as* $dSS(u,v) : \mathbb{R}^4 \times \mathbb{R}^4 \to \mathbb{R}^4$ *to be the following product:*

$$dSS(u,v) = \begin{pmatrix} u_0v_0 + u_1v_1 \\ u_2v_2 + u_3v_3 \\ u_0v_1 + u_1v_0 \\ u_2v_3 + u_3v_2 \end{pmatrix}$$

Definition 3.3. *We define the divided-straight-cross product as* $dSX(u,v) : \mathbb{R}^4 \times \mathbb{R}^4 \to \mathbb{R}^4$ *to be the following product:*

$$dSX(u,v) = \begin{pmatrix} u_0v_0 + u_1v_1 \\ u_1v_0 + u_0v_1 \\ u_2v_2 + u_3v_3 \\ u_3v_2 + u_2v_3 \end{pmatrix}$$

These vector products comes from the method used to reduce the marginals of a graph code to chains of those products. Some important properties of these vector products that we will use is how they are affected by the vector $(1010)^\top$.

$$\text{let } u = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}, v = \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}$$

$$dSX(u,v) = dSX(v,u) = \begin{pmatrix} 1a + 0b \\ 0a + 1b \\ 1c + 0d \\ 0c + 1d \end{pmatrix} = \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}$$

$$dSS(u,v) = dSS(v,u) = \begin{pmatrix} 1a + 0b \\ 1c + 0d \\ 1b + 0a \\ 1d + 0c \end{pmatrix} = \begin{pmatrix} a \\ c \\ b \\ d \end{pmatrix}$$

This means that u is the identity element for $\langle \mathbb{R}^4, dSX \rangle$, and $dSS(u, v)$ swaps the two middle elements b and c of v. As an example of how one can compute marginals, take the graph code generated by the matrix:

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

From this matrix we find the codewords:

$$C = \{0000, 100\omega, 10\omega0, 00\omega\omega, 1\omega00, 0\omega0\omega, 0\omega\omega0, 1\omega\omega\omega, \omega111, \omega^2 11 \\ \omega^2, \omega^2 1\omega^2 1, \omega 1\omega^2\omega^2, \omega^2\omega^2 11, \omega\omega^2 1\omega^2, \omega\omega^2\omega^2 1, \omega^2\omega^2\omega^2\omega^2\}$$

The adjacency matrix Γ from $H = \Gamma + \omega I$ gives the graph seen in Figure 8.

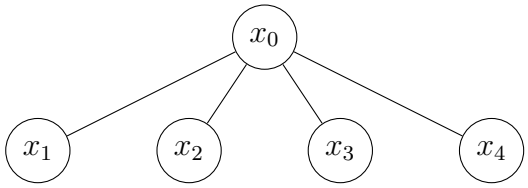


Figure 8: Graph G corresponding to parity check matrix H

Let the soft information in the nodes be denoted $p_i = (c_i, d_i, e_i, f_i)^\top$ The marginals for x_0 can be calculated directly from the code words C and will be:

$$m_0 = \begin{pmatrix} c_0(c_1(c_2c_3 + e_2e_3) + e_1(c_2e_3 + e_2c_3)) \\ d_0(c_1(c_2e_3 + e_2e_3) + e_1(c_2c_3 + e_2e_3)) \\ e_0(d_1(d_2d_3 + f_2f_3) + f_1(d_2f_3 + f_2d_3)) \\ f_0(d_1(d_2f_3 + f_2d_3) + f_1(d_2d_3 + f_2f_3)) \end{pmatrix}$$

Let $q = c_2c_3 + e_2e_3$, $r = e_2c_3 + c_2e_3$, $s = d_2d_3 + f_2f_3$, and $t = f_2d_3 + d_2f_3$. Let $g_j = dSS((1010)^\top, p_j) = (c_j, e_j, d_j, f_j)^\top$. We have that $(q, r, s, t)^\top = dSX(g_2, g_3)$. We can then reduce the expression to:

$$m_0 = \begin{pmatrix} c_0(c_1q + e_1r) \\ d_0(c_1r + e_1q) \\ e_0(d_1s + f_1t) \\ f_0(d_1t + f_1s) \end{pmatrix}$$

Let $A = c_1q + e_1r$, $B = c_1r + e_1q$, $C = d_1s + f_1t$, and $D = d_1t + f_1s$. $(A, B, C, D)^\top = dSX(g_1, (q, r, s, t)^\top)$. We can again reduce the expression to:

$$m_0 = \begin{pmatrix} c_0A \\ d_0B \\ e_0C \\ f_0D \end{pmatrix} = p_0 \cdot (A, B, C, D)^\top$$

We can then express the entire marginal as a product using the dot product, and dSX:

$$m_0 = p_0 \cdot dSX(g_1, dSX(g_2, g_3))$$

Expanding g_j using its definition as $dSS((1010)^\top, p_j)$, we have:

$$m_0 = p_0 \cdot dSX(dSS((1010)^\top, p_1), dSX(dSS((1010)^\top, p_2), dSS((1010)^\top, p_3)))$$

The marginals for x_1 using the global function will look like this:

$$m_1 = \begin{pmatrix} c_1(c_0(c_2c_3 + e_2e_3) + d_0(c_2e_3 + e_2c_3)) \\ d_1(e_0(d_2d_3 + f_fe_3) + f_0(d_2f_3 + f_2d_3)) \\ e_1(c_0(d_2d_3 + f_2f_3) + d_0(d_2f_3 + f_2d_3)) \\ f_1(e_0(d_2f_3 + f_2d_3) + f_0(d_2d_3 + f_2f_3)) \end{pmatrix}$$

Using the same q, r, s, t as in the marginal for x_0, we can reduce the marginal to:

$$m_1 = \begin{pmatrix} c_1(c_0q + d_0r) \\ d_1(e_0s + f_0t) \\ e_1(c_0r + d_0q) \\ f_1(e_0t + f_0s) \end{pmatrix}$$

Using $E = c_0q + d_0r$, $F = e_0s + f_0t$, $G = c_0r + d_0q$, $H = e_0t + f_0s$, we have that $(E, F, G, H)^\top = dSS(x_0, (q, r, s, t)^\top)$. We can then reduce the marginal to:

$$m_1 = \begin{pmatrix} c_1E \\ d_1F \\ e_1G \\ f_1H \end{pmatrix} = p_1 \cdot (E, F, G, H)^\top$$

Again, we can express m_1 as a product of $\cdot$, dSS, and dSX:

$$m_1 = p_1 \cdot dSS(p_0, dSX(g_2, g_3))$$

$$= p_1 \cdot dSS(p_0, dSX(dSS((1010)^\top, p_2), dSS((1010)^\top, p_3)))$$

BPSG is based on this reduction of the marginals into products using $\cdot$, dSS, and dSX. In order to compute the marginals from the example, x_2 and x_3 can send $g_2 = (c_2, e_2, d_2, f_2)$ and $g_3 = (c_3, e_3, d_3, f_3)$ respectively to x_0. After x_0 receives these messages, it can combine them into a message-product mp using dSX. It can then combine it with its own information by doing $dSS(p_0, mp)$, which it can send to x_1. x_1 can then compute its marginals by taking the dot product of its own soft information and the message from x_0. Similarly, x_0 computes its marginal by creating a message product mp by applying dSX to all its received messages and then compute $p_0 \cdot mp$.

In general, any node x_i can compute its marginals by having the message product $mp = (1010)^\top$ and then apply $dSX(mp, m)$ for all its received messages m. The node can then take the dot-product of its own soft information and the message product in order to get its final marginal. When any node x_i is selected to send a message to a neighbour x_j, it can create the message by having $mp = (1010)^\top$ and apply $dSX(mp, m)$ to all messages m excluding any message from the recipient x_j. It can then finalize the message by taking the dSS-product of its own soft information and the message products.

When using this reduction of marginals to a product of $\cdot$, dSX and dSS, unlike in the discriminative decoder in [12], nodes do not need to compute different products based on whether the messages are from leaves or internal nodes.

3.2 Belief propagation for simple graphs

Belief propagation for simple graphs (BPSG) operates on the simple graph generated by the adjacency matrix Γ from a given graph codes parity check matrix $H = \Gamma + \omega I$. In this graph, the nodes representing characters in the code are able to receive and store soft information $p \in \mathbb{R}^4$ from a noise channel representing the probabilities of a character being $0, 1, \omega$ or ω^2. The nodes are also able to receive messages $m \in \mathbb{R}^4$ from its neighbouring nodes. The nodes also need to be able to compute its own marginals and create messages for its neighbours based on their own soft information, and their own received messages. Just as in general factor graph decoding in $\mathbb{F}_2$, we want the message for a given recipient r to be calculated without taking into account any message from r itself, as to avoid false amplification of its own belief, therefore the nodes needs to be able to keep track of which message comes from which node.

When a node is selected to create a message to one of its neighbours, it computes a message-product by taking the dSX-product of its received

messages and then takes the dSS-product of its own soft information and this message-product. This can be seen in Algorithm 1. Note that we define these algorithms differently than those found in the future work section of [12], such that they more closely resemble the reduction of marginals in Section 3.1.

Algorithm 1: Computing a message

Let $M \setminus r$ be the set of all received messages excluding any message r from the recipient. Let p be the soft information of the computing node.

Function *createMessage(Node recipient)*
  $messageProduct \leftarrow (1010)^\top$
  for *each* $m \in M \setminus r$ **do**
    $messageProduct \leftarrow dSX(m, messageProduct)$
  end
  $messageProduct \leftarrow dSS(p, messageProduct)$
  return messageProduct

We initiate the messageProduct with the vector $(1010)^\top$ as it is the identity element for $\langle \mathbb{R}^4, dSX \rangle$. In the example from Section 3.1, we saw that leaf nodes should send their soft information with the values for 1 and ω swapped. Since leaf nodes can only receive and send messages with its parent, no message will be taken into account in the message product and it will remain $(1010)^\top$. They will therefore send the message $dSS((a, b, c, d)^\top, (1010)^\top) = (a, c, b, d)^\top$. As seen in the edge-case for leaves, unlike the discriminative decoder in [12], nodes do not need to know whether themselves or their neighbours are leaves.

When a node is selected to compute its marginals after y amount of flooding iterations, it runs Algorithm 2. All received messages are taken into account in the dSX-product and the result is then multiplied with the variables' soft information using the dot-product. Again the *messageProduct*

can be initialized with $(1010)^\top$.

Algorithm 2: Computing a marginal

Let p be the soft information of the computing node. Let M be the set of all received messages.

Function *marginal()*
  $messageProduct \leftarrow (1010)^\top$
  for ***each*** $m \in M$ **do**
    $messageProduct \leftarrow dSX(m, messageProduct)$
  end
  $marginal \leftarrow \cdot(p, messageProduct)$
  return *marginal*

With the two algorithms described we can perform decoding on simple graphs with a flooding scheme where for each flooding iteration one message is passed in each direction for each edge in the graph as seen in Algorithm 3.

Algorithm 3: Flooding the graph y times

Let n be the number of nodes in the graph, y be the amount of flooding iterations, and E be the set of edges in the graph.

Function *flood(y)*
  for *i in* $\{0 \ldots y\}$ **do**
    for ***each*** $(x_a, x_b) \in E$ **do**
      $x_a.giveMessage(x_b.createMessage(x_a))$
      $x_b.giveMessag(x_a.createMessage(x_b))$
    end
  end

A complete decoding using $BPSG$ can be performed by selecting y flooding iterations and running $flood(y)$. The resulting marginals can then be obtained by running $x_i.marginal()$ for all nodes in the graph. We refer to flooding the graph y times, and then computing the marginals as $BPSG(y)$.

3.3 Marginals in trees and general graphs using BPSG

Due to BPSG being directly based on marginal computation in trees, it is able to compute the exact marginals of graph codes that form trees. Take the graph code generated by the matrix H:

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

We can associate to the graph code generated by H the tree in Figure 9

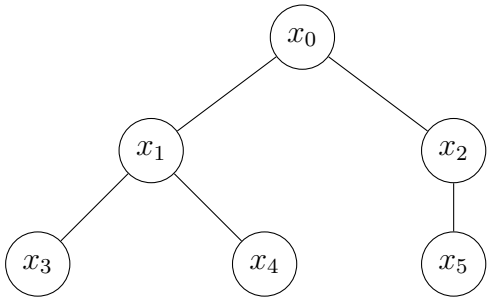


Figure 9: Graph G corresponding to parity check matrix H

Suppose the codeword $w11000$ is transmitted and the decoder has the following soft information to work with:

$$p_0 = \begin{pmatrix} 0.2 \\ 0.2 \\ 0.5 \\ 0.1 \end{pmatrix}, p_1 = \begin{pmatrix} 0.2 \\ 0.2 \\ 0.1 \\ 0.5 \end{pmatrix}, p_2 = \begin{pmatrix} 0.2 \\ 0.5 \\ 0.2 \\ 0.1 \end{pmatrix},$$

$$p_3 = \begin{pmatrix} 0.5 \\ 0.2 \\ 0.2 \\ 0.1 \end{pmatrix}, p_4 = \begin{pmatrix} 0.5 \\ 0.2 \\ 0.2 \\ 0.1 \end{pmatrix}, p_5 = \begin{pmatrix} 0.5 \\ 0.2 \\ 0.2 \\ 0.1 \end{pmatrix}$$

In this example there is a "bit" error in x_1, where p_1 has "ω^2" as the most likely character instead of "1". Using implementation of the global function and BPSG in java to decode this soft information, we get the results in Table 1.

Marginal	Global function	BPSG(50)
m_0	$\begin{pmatrix} 0.18806656744689498 \\ 0.08929779461507242 \\ 0.6670274658368284 \\ 0.05560817210120418 \end{pmatrix}$	$\begin{pmatrix} 0.1880665674468949 \\ 0.08929779461507238 \\ 0.6670274658368286 \\ 0.05560817210120415 \end{pmatrix}$
m_1	$\begin{pmatrix} 0.3995850809543138 \\ 0.47995309610787895 \\ 0.020114553736526403 \\ 0.10034726920128086 \end{pmatrix}$	$\begin{pmatrix} 0.3995850809543138 \\ 0.479953096107879 \\ 0.02011455373652641 \\ 0.10034726920128086 \end{pmatrix}$
m_2	$\begin{pmatrix} 0.20457312948180226 \\ 0.6961168989311324 \\ 0.06629684751725072 \\ 0.033013124069814645 \end{pmatrix}$	$\begin{pmatrix} 0.2045731294818022 \\ 0.6961168989311325 \\ 0.06629684751725072 \\ 0.03301312406981464 \end{pmatrix}$
m_3	$\begin{pmatrix} 0.6282415550444232 \\ 0.08018761556848418 \\ 0.2512966220177694 \\ 0.04027420736932306 \end{pmatrix}$	$\begin{pmatrix} 0.6282415550444235 \\ 0.0801876155684842 \\ 0.2512966220177694 \\ 0.04027420736932306 \end{pmatrix}$
m_4	$\begin{pmatrix} 0.6282415550444232 \\ 0.08018761556848418 \\ 0.2512966220177694 \\ 0.04027420736932306 \end{pmatrix}$	$\begin{pmatrix} 0.6282415550444235 \\ 0.0801876155684842 \\ 0.2512966220177694 \\ 0.04027420736932306 \end{pmatrix}$
m_5	$\begin{pmatrix} 0.6961168989311325 \\ 0.06629684751725072 \\ 0.20457312948180228 \\ 0.03301312406981465 \end{pmatrix}$	$\begin{pmatrix} 0.6961168989311325 \\ 0.0662968475172507 \\ 0.2045731294818022 \\ 0.03301312406981464 \end{pmatrix}$

Table 1: Marginal computation comparison between the global function and BPSG(50)

Just as Hansen found in her thesis for the discriminate decoder [12], we find that BPSG calculates the exact marginals for trees, with minor differences in some of the last digits, which we attribute to the fact that floating point operations in java are not associative [2][1]. We now look at how BPSG perform in some graphs that contain cycles such as circles and nested-cliques. In binary factor graph decoding, one can not expect to compute the exact marginals in factor graphs that contain cycles [15], and we expect the same to be the case for BPSG. Consider the two graphs G_0 and G_2 in Figures 10 and 11. Since the neighbourhood of x_0 is the same in both G_0 and G_2, $\omega 1001$ is a codeword in both these graph codes.

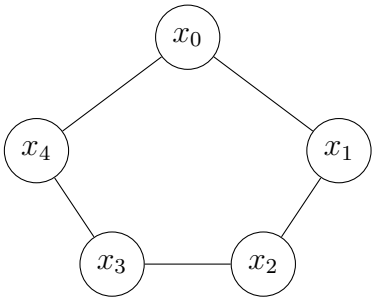


Figure 10: Graph G_0: 5 nodes in a circle

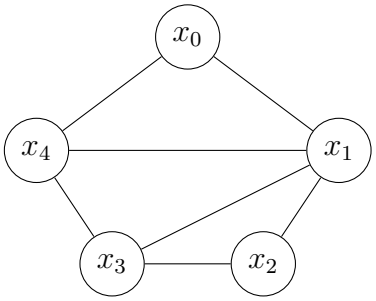


Figure 11: Graph G_2: 5 nodes in a circle, with two short-cuts

We initiate the soft information to correspond to the codeword $\omega 1001$ with an error in node x_1:

$$p_0 = \begin{pmatrix} 0.133 \\ 0.133 \\ 0.6 \\ 0.133 \end{pmatrix}, p_1 = \begin{pmatrix} 0.133 \\ 0.133 \\ 0.133 \\ 0.6 \end{pmatrix}, p_2 = \begin{pmatrix} 0.6 \\ 0.133 \\ 0.133 \\ 0.133 \end{pmatrix},$$

$$p_3 = \begin{pmatrix} 0.6 \\ 0.133 \\ 0.133 \\ 0.133 \end{pmatrix}, p_4 = \begin{pmatrix} 0.133 \\ 0.6 \\ 0.133 \\ 0.133 \end{pmatrix},$$

Applying the global function and BPSG on G_0 using this soft information we get the results in Table 2:

Marginal	Global function	BPSG(50)
m_0	$\begin{pmatrix} 0.07312178213933611 \\ 0.07312178213933611 \\ 0.6988456499765301 \\ 0.15491078574479755 \end{pmatrix}$	$\begin{pmatrix} 0.07695303937884447 \\ 0.07375303814479427 \\ 0.6478579206045338 \\ 0.20143600187182745 \end{pmatrix}$
m_1	$\begin{pmatrix} 0.07312178213933611 \\ 0.5238837343408642 \\ 0.07312178213933611 \\ 0.3298727013804636 \end{pmatrix}$	$\begin{pmatrix} 0.1366037895821299 \\ 0.46631764496518324 \\ 0.05206313350300684 \\ 0.34501543194968004 \end{pmatrix}$
m_2	$\begin{pmatrix} 0.6988456499765303 \\ 0.15491078574479755 \\ 0.0731217821393361 \\ 0.0731217821393361 \end{pmatrix}$	$\begin{pmatrix} 0.6478579206045341 \\ 0.20143600187182728 \\ 0.07695303937884451 \\ 0.07375303814479423 \end{pmatrix}$
m_3	$\begin{pmatrix} 0.6988456499765302 \\ 0.07312178213933611 \\ 0.07312178213933611 \\ 0.15491078574479758 \end{pmatrix}$	$\begin{pmatrix} 0.7069528644035826 \\ 0.0931845251948374 \\ 0.11126770459942226 \\ 0.08859490580215763 \end{pmatrix}$
m_4	$\begin{pmatrix} 0.07312178213933611 \\ 0.6988456499765302 \\ 0.15491078574479755 \\ 0.07312178213933611 \end{pmatrix}$	$\begin{pmatrix} 0.09318452519483739 \\ 0.7069528644035826 \\ 0.08859490580215763 \\ 0.11126770459942228 \end{pmatrix}$

Table 2: Marginal computation comparison between the global function and BPSG(50) on G_0

As can be seen in Table 2, most marginals are similar to the global function, and in x_1 where the error was, BPSG is able to correct it. If we introduce more cycles to the graph by using G_2, x_1 where the error is present is connected to all the other nodes. As such, x_1's beliefs will be propagated at a higher rate than the soft information in the other nodes. We can see the marginals from G_2 using the same soft information in Table 3.

Marginal	Global function	BPSG(50)
m_0	$\begin{pmatrix} 0.07312178213933612 \\ 0.15491078574479758 \\ 0.6988456499765303 \\ 0.07312178213933612 \end{pmatrix}$	$\begin{pmatrix} 0.12694561638057028 \\ 0.13658841348661804 \\ 0.5912766965854168 \\ 0.1451892735473949 \end{pmatrix}$
m_1	$\begin{pmatrix} 0.07312178213933611 \\ 0.5238837343408642 \\ 0.07312178213933611 \\ 0.3298727013804636 \end{pmatrix}$	$\begin{pmatrix} 0.262446458057229 \\ 0.3054431901693206 \\ 0.07657578303948656 \\ 0.3555345687339639 \end{pmatrix}$
m_2	$\begin{pmatrix} 0.6988456499765303 \\ 0.07312178213933608 \\ 0.07312178213933608 \\ 0.15491078574479752 \end{pmatrix}$	$\begin{pmatrix} 0.591276696740994 \\ 0.14518927354089334 \\ 0.12694561633285587 \\ 0.1365884133852568 \end{pmatrix}$
m_3	$\begin{pmatrix} 0.6988456499765302 \\ 0.0731217821393361 \\ 0.15491078574479758 \\ 0.07312178213933611 \end{pmatrix}$	$\begin{pmatrix} 0.593182879773234 \\ 0.12952668383538146 \\ 0.14051630039121693 \\ 0.1367741360001676 \end{pmatrix}$
m_4	$\begin{pmatrix} 0.07312178213933611 \\ 0.6988456499765302 \\ 0.07312178213933611 \\ 0.15491078574479755 \end{pmatrix}$	$\begin{pmatrix} 0.12952668380198096 \\ 0.5931828797181119 \\ 0.1367741360289675 \\ 0.14051630045093963 \end{pmatrix}$

Table 3: Marginal computation comparison between the global function and BPSG(50) on G_2

As can be seen in Table 3, BPSG is not able to correct the error in x_1 using G_2. Whether we use G_1 or G_2 makes a difference as to whether we will be able to decode the received message using BPSG. Since $G_2 = LC_0(LC_2(G_0))$, G_2 and G_0 are LC-equivalent. In the marginals from the global function, one can verify that the probabilities has been augmented according to the rules of LC. When applying $LC_0(G_0) = G_1$, we swap 1 and ω^2 in x_0, and swap ω and ω^2 in the neighbours of x_0, namely x_1 and x_4. Then, by applying $LC_2(G_1) = G_2$, we permute the same probabilities for LC on x_2 and its neighbours, x_1 and x_3. The differences of the marginals from the global function on G_1 and G_2 can be attributed to these exact permutations.

As has been showcased in this example, using different graphs in the same LC-orbit can improve the performance of BPSG. In the example of G_0 and G_2, G_0 contains less cycles than G_2 and so x_1 had its own information propagated back to itself at a lower rate. However, if the soft information in x_1 was correct, it would have propagated correct information through the

graph at a higher rate.

3.4 Extending BPSG with LC operations

Due to the problems short cycles cause when decoding using the BPSG as showcased in the previous section, we turn to *Local Complementation* (LC) whose properties were described in Section 2.6. The goal is to improve BPSG by using the many different codes in the LC-orbit of the graph code. We want to utilize these different graph-representations in order to avoid passing soft information through the same short cycles. We propose an LC-extension that provides logic for converting graphs and soft information based on the properties of LC, and that applies BPSG on each graph-instance in the LC-orbit. When dealing with these different graph-representations, it is needed to keep track of the permutations the soft information has gone through in order to convert it back to the original graph code selected for error correction.

3.4.1 LC algorithms

We want an algorithm that, given the node v and Graph G containing associated soft information and messages in each node, computes $G' = LC_v(G)$. The graph needs to be changed according to the LC-rules, and the soft information in the nodes needs to be changed accordingly. The soft information needs to be swapped with regards to how the code C relates to the equivalent code C' in its LC-orbit. In the code C' generated by $LC_v(G)$, ω^2 and 1 will be swapped in position v, and ω^2 and ω will be swapped in position $i \in \mathcal{N}_v$. We outline an algorithm for this in Algorithm 4, where the graph along with soft information is changed.

Algorithm 4: Performing $LC_v(G)$

Function *LC(v)*
- marginalToSoftInformation()
- clearMessages()
- softChange(v)
- graphChange(v)

The way we combine the information from BPSG applied to the different graph-representations, is by replacing the soft information of all nodes $v \in V$ of G with the marginals of that node. This means that after y flooding iterations using BPSG, the information is propagated to the next graph-representation. In order to make sure that the nodes do not take into

Algorithm 5: Replacing soft information with current marginals

Let P be the set of i vectors where $p_i \in \mathbb{R}^4$ corresponds to the soft information of node x_i.

Function *marginalsToSoftInformation()*
  for ***each*** $i \in \{0 \ldots n-1\}$ **do**
    $p_i = x_i$.marginal()
  end

Algorithm 6: Clearing messages

Function *clearMessages()*
  for ***each*** $i \in \{0 \ldots n-1\}$ **do**
    x_i.clearMessages()
  end

Algorithm 7: Swapping soft information according to LC_v

Let P be the set of i vectors where $p_i \in \mathbb{R}^4$ corresponds to the soft information of node x_i. Let $\mathcal{N}_v$ be the set of indices neighbouring v.

Function *softChange(v)*
  swap1$\omega^2(p_v)$
  for ***each*** $i \in \mathcal{N}_v$ **do**
    swap$\omega\omega^2(p_i)$
  end

consideration messages from nodes that they may no longer be connected to, we also clear all their messages when performing LC. We can then use this function $LC(v)$ in a sequence where after each LC operation, we perform y flooding iterations with BPSG. One way to setup a sequence of LC operations in order to go through many graph-representations is to apply LC iteratively to all nodes nodes x_0 to x_{n-1}, and then loop around by applying LC to x_0 again. When using this decoding scheme, we can convert the code back to the original by applying LC to the same nodes in reverse order. After we have gone through many flooding iterations on many graph-forms and when the graph has been returned to its original form, we can find the resulting marginals by checking the soft information in each node. We call this process *iterative local complementation* (ILC).

Algorithm 8: Adjusting graph according to LC_v

Let $G = (V, E)$ be the current graph representation. Let $\mathcal{N}_v$ be the set of nodes neighbouring v.

Function *graphChange(v)*
  for ***each*** $(n, m) \in \{(n, m) \mid n, m \in \mathcal{N}_v, n \neq m\}$ **do**
    if $(n, m) \in E$ **then**
      $E \leftarrow E \setminus (n, m)$
    end
    else
      $E \leftarrow E \cup (n, m)$
    end
  end

Algorithm 9: Complete decoding process applying LC iterativeley

Let n be the number of nodes in the graph, let y be the amount of flooding iterations per graph-representation, and let z be total the amount of graph-representations we want to use.

Function *iterativeLC(y, z)*
  $flood(y)$
  for *i in* $\{0 \ldots z\}$ **do**
    LC(i mod n)
    $flood(y)$
  end
  for *i in* $\{z \ldots 0\}$ **do**
    LC(i mod n)
  end

The algorithm $iterativeLC(y, z)$ shows the whole decoding process using ILC. First, y flooding iterations is applied to the initial graph G representing the code. Then $LC_i(G)$ is applied z times for $i \in \{0 \ldots z \pmod{\text{n}}\}$ doing y flooding iterations for each of the graph representations. In the end, the graph is returned to its original form, and the nodes contain their marginals in their soft information.

3.5 Step-by-step belief propagation using ILC

In order to get a closer look at the effects of ILC, we can do a running example on the graph code with generator and parity check matrix:

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

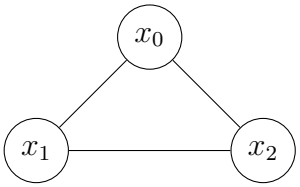


Figure 12: Small graph

From this matrix we can construct the graph G in Figure 12. In order to run ILC on this graph, we first need to chose the amount of LC iterations and flooding iterations we want to use. One LC iteration with two flooding iterations each should suffice empirically for the purpose of going through the different steps of the algorithm. We can use the generator matrix in order to obtain the codewords:

$$C = \{000, \omega 11, 1\omega 1, 11\omega, \omega^2\omega^2 0, \omega^2 0\omega^2, 0\omega^2\omega^2, \omega\omega\omega\}.$$

Assume the codeword $1\omega 1$ is transmitted over a noisy channel, and the resulting soft information at the recipient is:

$$\begin{pmatrix} P(x_0 = 0) = 0.05 \\ P(x_0 = 1) = 0.7 \\ P(x_0 = \omega) = 0.1 \\ P(x_0 = \omega^2) = 0.15 \end{pmatrix}, \begin{pmatrix} P(x_1 = 0) = 0.05 \\ P(x_1 = 1) = 0.1 \\ P(x_1 = \omega) = 0.7 \\ P(x_1 = \omega^2) = 0.15 \end{pmatrix}, \begin{pmatrix} P(x_2 = 0) = 0.25 \\ P(x_2 = 1) = 0.25 \\ P(x_2 = \omega) = 0.25 \\ P(x_2 = \omega^2) = 0.25 \end{pmatrix}$$

These probabilities are given to the respective nodes x_0, x_1 and x_2. Following from the beginning of Algorithm 9 $iterativeLC(y, z)$, we first flood the graph $y = 2$ times. In the first flooding iteration, 6 messages is computed: $m_{x_0 \to x_1}$, $m_{x_1 \to x_0}$, $m_{x_0 \to x_2}$, $m_{x_2 \to x_0}$, $m_{x_1 \to x_2}$, and $m_{x_2 \to x_1}$. Let p_i be the vector containing the soft information for node x_i. Since the nodes have not yet received messages from their neighbours, the messages will be calculated by $dSS(p_i, (1010)^\top)$. Since $dSS((abcd)^\top), (1010)^\top) = (acbd)^\top$, the messages

will be the senders' soft information with the probabilities for $P(x_i = 1)$ and $P(x_i = \omega)$ swapped. These messages in the first flooding iteration can be seen in Table 4.

Message	Vector
$m_{x_0 \to x_1}, m_{x_0 \to x_2}$	$\begin{pmatrix} 0.05 \\ 0.1 \\ 0.7 \\ 0.15 \end{pmatrix}$
$m_{x_1 \to x_0}, m_{x_1 \to x_2}$	$\begin{pmatrix} 0.05 \\ 0.7 \\ 0.1 \\ 0.15 \end{pmatrix}$
$m_{x_2 \to x_0}, m_{x_2 \to x_1}$	$\begin{pmatrix} 0.25 \\ 0.25 \\ 0.25 \\ 0.25 \end{pmatrix}$

Table 4: Messages in the first flooding iteration

In the second flooding iteration, the messages can be computed based on the messages from the previous iteration. For example, the message $m_{x_0 \to x_1}$ will be computed by $dSS(p_0, dSX(m_{x_2 \to x_1}, (1010)^\top)$. We know that $(1010)^\top$ is the identity element for dSX, so the message will simply be $dSS(p_0, m_{x_2 \to x_1})$. Since x_1 is the recipient, the message from x_1 to x_0 is not used, as to not send back the original beliefs of x_1. The messages for this second iteration can be seen in Table 5.

Message	Vector
$m_{x_0 \to x_1}$	$dSS(p_0, m_{x_2 \to x_0}) = \begin{pmatrix} 0.1875 \\ 0.0625 \\ 0.1875 \\ 0.0625 \end{pmatrix}$
$m_{x_1 \to x_0}$	$dSS(p_1, m_{x_2 \to x_1}) = \begin{pmatrix} 0.0375 \\ 0.2125 \\ 0.0375 \\ 0.2125 \end{pmatrix}$
$m_{x_0 \to x_2}$	$dSS(p_0, m_{x_1 \to x_0}) = \begin{pmatrix} 0.4925 \\ 0.0325 \\ 0.07 \\ 0.03 \end{pmatrix}$
$m_{x_2 \to x_0}$	$dSS(p_2, m_{x_1 \to x_2}) = \begin{pmatrix} 0.1875 \\ 0.0625 \\ 0.1875 \\ 0.625 \end{pmatrix}$
$m_{x_1 \to x_2}$	$dSS(p_1, m_{x_0 \to x_1}) = \begin{pmatrix} 0.0125 \\ 0.5125 \\ 0.01 \\ 0.21 \end{pmatrix}$
$m_{x_2 \to x_1}$	$dSS(p_2, m_{x_0 \to x_2}) = \begin{pmatrix} 0.0375 \\ 0.2125 \\ 0.0375 \\ 0.02125 \end{pmatrix}$

Table 5: Messages in the second flooding iteration

After this final flooding iteration, we now start with the steps of local complementation. $LC(0)$ is performed, and we follow the corresponding steps of Algorithm 4. First, the marginals of the nodes are computed using the messages from the final flooding iteration. The marginals are computed according to Algorithm 2, where first a message-product is computed from the messages, and then the final marginal is the dot-product of the nodes original beliefs and the message-product. For example, $x_0.marginal()$ will be computed by $\cdot(p_0, dSX(m_{x_2 \to x_0}, m_{x_1 \to x_0}))$. These marginals can be seen in Table 6. The marginals in this table are normalized and rounded in order to make them more readable.

marginal	Vector
$x_0.marginal()$	$\begin{pmatrix} 0.03 \\ 0.76 \\ 0.05 \\ 0.16 \end{pmatrix}$
$x_1.marginal()$	$\begin{pmatrix} 0.04 \\ 0.16 \\ 0.55 \\ 0.25 \end{pmatrix}$
$x_2.marginal()$	$\begin{pmatrix} 0.08 \\ 0.85 \\ 0.2 \\ 0.05 \end{pmatrix}$

Table 6: Marginals after 2 flooding iterations on the first graph

In order to keep the information gathered from these flooding iterations in the next graph-representation, the soft information in the three nodes are replaced with these marginals. The graph can then be changed according to the rules of LC. Since x_1, and x_2 are the only neighbours of x_0 and they have an edge between them, it is removed. The new graph-representation G' of this code can be seen in Figure 13, and its corresponding generator and parity check matrix will be:

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

By swapping 1 and ω^2 in position 0, and swapping ω and ω^2 in positions 1 and 2 in C, we obtain the codewords C' corresponding to the matrix H':.

$$C = \{000, \omega 11, 1\omega 1, 11\omega, \omega^2\omega^2 0, \omega^2 0\omega^2, 0\omega^2\omega^2, \omega\omega\omega\}.$$
$$C' = \{000, \omega 11, \omega^2\omega^2 1, \omega^2 1\omega^2, 1\omega 0, 10\omega, 0\omega\omega, \omega\omega^2\omega^2\}.$$

This means that sending the codeword $1\omega 1$ using the code C, is equivalent to sending the codeword $\omega^2\omega^2 1$ using the code C' in terms of re-labeling. Due to this, the soft information in the nodes has to be changed according to the same rules as seen in $softChange(0)$. x_0 swaps the probabilities for 1 and ω^2, while x_1 and x_2 swaps the probabilities for ω and ω^2. The new soft infromation in the nodes of G' is shown in Table 7.

Soft information	Vector
p_0	$\begin{pmatrix} 0.03 \\ 0.16 \\ 0.05 \\ 0.76 \end{pmatrix}$
p_1	$\begin{pmatrix} 0.04 \\ 0.16 \\ 0.25 \\ 0.55 \end{pmatrix}$
p_2	$\begin{pmatrix} 0.08 \\ 0.85 \\ 0.05 \\ 0.2 \end{pmatrix}$

Table 7: New soft information in second graph-representation

After these changes, the flooding iterations for the new graph can begin. Just as in the previous graph, the message calculations in the first flooding iteration will be the soft information of the sender, with the probabilities for $x_i = 1$ and $x_i = \omega$ swapped. The messages for the first flooding iteration in the second graph can be seen in Table 8

Message	Vector
$m_{x_0 \to x_1}, m_{x_0 \to x_2}$	$\begin{pmatrix} 0.03 \\ 0.05 \\ 0.16 \\ 0.76 \end{pmatrix}$
$m_{x_1 \to x_0}$	$\begin{pmatrix} 0.04 \\ 0.25 \\ 0.16 \\ 0.55 \end{pmatrix}$
$m_{x_2 \to x_0}$	$\begin{pmatrix} 0.08 \\ 0.05 \\ 0.85 \\ 0.2 \end{pmatrix}$

Table 8: Messages in the first flooding iteration of the second graph

In the second flooding iteration of the second graph, since both x_1 and x_2 have not received a message from any other node than x_0, they will not be

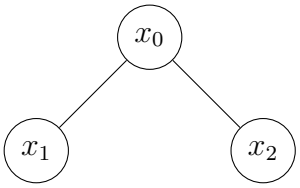


Figure 13: Small graph G' after $LC(0)$ in G

able to compute an updated message. They will therefore again only be able to compute a message based on their own soft information. x_0 can create new messages for x_1 and x_2 based on the message from the opposite node. The message $m_{x_0 \to x_1}$ will be calculated by $dSS(p_0, m_{x_2 \to x_0})$, and similarly $m_{x_0 \to x_2} = dSS(p_0, m_{x_1 \to x_0})$. The updated messages calculated by x_0 can be seen in Table 9

Message	Vector
$m_{x_0 \to x_1}$	$\begin{pmatrix} 0.0104 \\ 0.1945 \\ 0.0143 \\ 0.656 \end{pmatrix}$
$m_{x_0 \to x_2}$	$\begin{pmatrix} 0.0412 \\ 0.426 \\ 0.0139 \\ 0.1491 \end{pmatrix}$

Table 9: New messages from x_0 in the second flooding iteration of the second graph

After this, the code C' can be reverted back to the original C by re-applying $LC(0)$. The progress gathered from the final flooding iterations needs to be kept, and we replace the nodes' soft information with the marginals of the nodes. Similarly to in the first graph, the marginal for x_0 will be $x_0.marginal() = \cdot(p_0, dSX(m_{x_2 \to x_0}, m_{x_1 \to x_0}))$. Since x_1 and x_2 only has a single neighbour, their marginals will be calculated by:
$x_1.marginal() = \cdot(p_1, m_{x_0 \to x_1})$, and $x_2.marginal() = \cdot(p_2, m_{x_0 \to x_2})$. They can be seen in Table 10.

marginal	Vector
$x_0.marginal()$	$\begin{pmatrix} 0.001 \\ 0.009 \\ 0.031 \\ 0.959 \end{pmatrix}$
$x_1.marginal()$	$\begin{pmatrix} 0.001 \\ 0.079 \\ 0.009 \\ 0.911 \end{pmatrix}$
$x_2.marginal()$	$\begin{pmatrix} 0.008 \\ 0.915 \\ 0.002 \\ 0.075 \end{pmatrix}$

Table 10: Marginals after 2 flooding iterations on the second graph

These marginals correspond to the codewords in C', and indicates that the codeword $\omega^2\omega^2 1$ was sent. Since we are interested in the results for the original code C, we can re-apply $LC(0)$. The soft information will be swapped according to the same rules and can be seen in Table 11.

marginal	Vector
$x_0.marginal()$	$\begin{pmatrix} 0.001 \\ 0.959 \\ 0.031 \\ 0.009 \end{pmatrix}$
$x_1.marginal()$	$\begin{pmatrix} 0.001 \\ 0.079 \\ 0.911 \\ 0.009 \end{pmatrix}$
$x_2.marginal()$	$\begin{pmatrix} 0.008 \\ 0.915 \\ 0.075 \\ 0.002 \end{pmatrix}$

Table 11: Final results of the decoding using 1 LC with 2 flooding iterations per graph

Now that the soft information has been converted so that it corresponds to the original code, the algorithm is finished and the the soft information

contains the final results. The algorithm concludes correctly that the most likely codeword sent was $1\omega 1$.

3.6 Marginal convergence in ILC

Though BPSG computes exact marginals for trees, this is not the case for ILC. This is due to ILC introducing cycles to the graph through a series of LC operations. An important property of ILC is how the soft information in the nodes are updated after the set amount of flooding iterations in the particular graph-instance. Due to the soft information being continuously updated in in every graph of the LC-orbit, the marginals tends to continue to converge to what is believed to be the correct character. Suppose you have a node x_i with probability vector $p_i = (0.5, 0.2, 0.2, 0.2)^\top$, and that after the given amount of flooding iterations, the marginal is $m_i = (0.7, 0.1, 0.1, 0.1)^\top$. The soft information is then updated to be the marginal, and after permuting the graph according to the rules of LC, new flooding iterations are performed. Suppose the new marginals is then $(0.8, 0.66, 0.66, 0.66)^\top$. If every graph instance continues to agree on the correctness of x_i, the marginal will converge towards $(1, 0, 0, 0)^\top$. As an example, consider the decoding of the 12-node nested-clique where we have the starting soft information corresponding to the codeword $\omega 11110001000$, where there are errors in the last two characters:

$$p_0 = \begin{pmatrix} 0.2 \\ 0.2 \\ 0.7 \\ 0.1 \end{pmatrix}, p_1 = \begin{pmatrix} 0.2 \\ 0.7 \\ 0.2 \\ 0.1 \end{pmatrix}, p_2 = \begin{pmatrix} 0.2 \\ 0.7 \\ 0.2 \\ 0.1 \end{pmatrix},$$

$$p_3 = \begin{pmatrix} 0.2 \\ 0.7 \\ 0.2 \\ 0.1 \end{pmatrix}, p_4 = \begin{pmatrix} 0.2 \\ 0.7 \\ 0.2 \\ 0.1 \end{pmatrix}, p_5 = \begin{pmatrix} 0.7 \\ 0.2 \\ 0.2 \\ 0.1 \end{pmatrix}$$

$$p_6 = \begin{pmatrix} 0.7 \\ 0.2 \\ 0.2 \\ 0.1 \end{pmatrix}, p_7 = \begin{pmatrix} 0.7 \\ 0.2 \\ 0.2 \\ 0.1 \end{pmatrix}, p_8 = \begin{pmatrix} 0.2 \\ 0.7 \\ 0.2 \\ 0.1 \end{pmatrix},$$

$$p_9 = \begin{pmatrix} 0.7 \\ 0.2 \\ 0.2 \\ 0.1 \end{pmatrix}, p_{10} = \begin{pmatrix} 0.2 \\ 0.7 \\ 0.2 \\ 0.1 \end{pmatrix}, p_{11} = \begin{pmatrix} 0.2 \\ 0.7 \\ 0.2 \\ 0.1 \end{pmatrix}$$

For sake of simplicity, we will focus on the marginals in the nodes where there are errors, namely x_{10} and x_{11}. We will compare the marginals of ILC to the marginals of BPSG using the same amount of flooding iterations. Suppose we use 2 flooding iterations per graph encountered through LC. After the first two flooding iterations, before doing any LC, the marginals will be:

Marginal	BPSG(2)	ILC(2,0)
m_{10}	$\begin{pmatrix} 0.1922199455682555 \\ 0.585615013777599 \\ 0.15423322671412748 \\ 0.06793181394001813 \end{pmatrix}$	$\begin{pmatrix} 0.1922199455682555 \\ 0.585615013777599 \\ 0.15423322671412748 \\ 0.06793181394001813 \end{pmatrix}$
m_{11}	$\begin{pmatrix} 0.1904239742615757 \\ 0.586887996166403 \\ 0.15411987473794236 \\ 0.06856815483407909 \end{pmatrix}$	$\begin{pmatrix} 0.1904239742615757 \\ 0.586887996166403 \\ 0.15411987473794236 \\ 0.06856815483407909 \end{pmatrix}$

Table 12: Marginal comparison between BPSG(2) and ILC(2,0)

Naturally, the marginals are exactly the same, as both algorithms have gone through the same amount of flooding iterations. Continuing with one LC operation, we have:

Marginal	BPSG(4)	ILC(2,1)
m_{10}	$\begin{pmatrix} 0.2002982132016834 \\ 0.589572709063172 \\ 0.1470194139671689 \\ 0.06310966376797568 \end{pmatrix}$	$\begin{pmatrix} 0.27398765594694685 \\ 0.5726650817495365 \\ 0.1170500735861681 \\ 0.036297188717348584 \end{pmatrix}$
m_{11}	$\begin{pmatrix} 0.20011886896497225 \\ 0.5894657668392841 \\ 0.1471500425203516 \\ 0.06326532167539196 \end{pmatrix}$	$\begin{pmatrix} 0.26379294272939935 \\ 0.5873303293171007 \\ 0.11230223346682447 \\ 0.03657449448667541 \end{pmatrix}$

Table 13: Marginal comparison between BPSG(4) and ILC(2,1)

After an additional 2 flooding iterations on a new graph, ILC has begun increasing the probability that x_{10} and x_{11} were "0" considerably more than doing 2 additional iterations on the same original graph. Flowing with another LC iteration:

Marginal	BPSG(6)	ILC(2,2)
m_{10}	$\begin{pmatrix} 0.20405829722336438 \\ 0.5901281541724192 \\ 0.14458347809574534 \\ 0.061230070508471145 \end{pmatrix}$	$\begin{pmatrix} 0.7391668163793695 \\ 0.21729674216928524 \\ 0.03963708454249967 \\ 0.003899356908845561 \end{pmatrix}$
m_{11}	$\begin{pmatrix} 0.2040551707252924 \\ 0.5895653657082193 \\ 0.14496994891242299 \\ 0.06140951465406544 \end{pmatrix}$	$\begin{pmatrix} 0.6838777042735805 \\ 0.2885051043997645 \\ 0.024061219895666768 \\ 0.0035559714309880983 \end{pmatrix}$

Table 14: Marginal comparison between BPSG(6) and ILC(2,2)

After applying 2 flooding iterations to 3 different graphs, ILC already finds x_{10} and x_{11} to most likely be "0". By continuing with many more LC iterations, we get the results in Table 15. The marginals taken directly from the global function can be seen in Table 16.

Marginal	BPSG(100)	ILC(2,50)
m_{10}	$\begin{pmatrix} 0.20549182235723512 \\ 0.5903403022080188 \\ 0.1436441746066074 \\ 0.06052370082813874 \end{pmatrix}$	$\begin{pmatrix} 1.0 \\ 0.0 \\ 0.0 \\ 0.0 \end{pmatrix}$
m_{11}	$\begin{pmatrix} 0.20563136749259817 \\ 0.589454971077216 \\ 0.14420116502324684 \\ 0.06071249640693915 \end{pmatrix}$	$\begin{pmatrix} 1.0 \\ 0.0 \\ 0.0 \\ 0.0 \end{pmatrix}$

Table 15: Marginal comparison between BPSG(100) and ILC(2,50)

In the scenario presented, ILC was able to decode the soft information into the correct codeword, where as BPSG did not. In the example, both errors were located in the same clique, propagating wrong information in short cycles between them. By using different graphs in the LC-orbit, we get rid of cycles that may prohibit the nodes from converging towards a codeword.

Marginal	Global Function
m_{10}	$\begin{pmatrix} 0.4573283016391904 \\ 0.3923735235945031 \\ 0.1027593263498699 \\ 0.04753884841643655 \end{pmatrix}$
m_{11}	$\begin{pmatrix} 0.4573283016391904 \\ 0.39237352359450334 \\ 0.10275932634986977 \\ 0.0475388484164364845 \end{pmatrix}$

Table 16: Marginals from the global function

4 Analysis and Assessment

In this section we show the results of decoding using ILC on larger graphs using more LC and flooding iterations. We primarily compare the two methods, BPSG and ILC. We generate soft information by passing words through our channel model described in Section 4.1, and feeding the same soft information to our two decoding methods. We are particularly interested in checking the performance of these algorithms on the strong graph codes with high minimum edit distance, which usually forms nested-clique structures [6]. We adress the runtimes of ILC, BPSG and compare them to the runtime of the global function. Though single-clique graph codes are not good codes (being LC-equivalent to a tree with distance $d = 2$), we end the section by looking at how performing LC on a clique, thereby turning it into a tree, affects the decoding performance.

4.1 Channel model

In order to be able to analyze the performance of BPSG and its extension ILC, we need to be able to simulate a noisy quadratic channel. For the binary symmetric channel, one can use the method described in Section 2.7. In order to apply the properties of AWGN to the quarternary channel, we replace $\{0, 1, \omega, \omega^2\}$ with $\{00, 01, 10, 11\}$ respectively. Each character in $\mathbb{F}_4$ is then defined as two bits $b_0, b_1 \in \mathbb{F}_2$. These two bits is then simulated to be sent through a noisy channel using the AWGN channel as described in Section 2.7. For the given noise N_0 and bit energy E_b, the bits b_0, b_1 is then encoded to to r_0, r_1 respectively. We then get for any $\mathbb{F}_4$-character two

probability vectors:

$$b_0 = \begin{pmatrix} P(b_0 = 0|r_0) \\ P(b_0 = 1|r_0) \end{pmatrix} b_1 = \begin{pmatrix} P(b_1 = 0|r_1) \\ P(r_1 = 1|r_1) \end{pmatrix}$$

From this, we construct a length 4 probability-vector representing any character $x \in \mathbb{F}_4$ as

$$x = \begin{pmatrix} P(x = 0) \\ P(x = 1) \\ P(x = \omega) \\ P(x = \omega^2) \end{pmatrix} = \begin{pmatrix} P(b_0 = 0|r_0) + P(b_1 = 0)|r_1) \\ P(b_0 = 0|r_0) + P(b_1 = 1)|r_1) \\ P(b_0 = 1|r_0) + P(b_1 = 0)|r_1) \\ P(b_0 = 1|r_0) + P(b_1 = 1)|r_1) \end{pmatrix}$$

The error rates gathered from simulating transmission and decoding of codewords in $\mathbb{F}_4$ is plotted with $10 \cdot log_{10}(\frac{E_b}{N_0})$ along the x-axis, and log_{10}(error rate) along the y-axis. The error rates we sample are the "bit" error rate and the word error rate.

4.2 Comparing methods

The following sections show comparisons between BPSG and the extension ILC using varying amounts of flooding and LC iterations. In order to make the comparison of different decoding methods as fair as possible, we make sure that all methods operate based on the same instances of soft information sampled from our channel model. We sample 2000 instances of soft information from our channel model per value of $\frac{E_b}{N_0}$, usually such that each data point $10 \cdot log_{10}(E_b/N_0) \in \{-0.5, -0.45, ..., 1.45, 1.5\}$. This same soft information is then fed to the different decoders and we sample log_{10}(bit error rates) and log_{10}(word error rates) showing how successful the 2000 decoding were. The more decoding we do per level of noise, the more the results are averaged out, and the smoother the resulting curves will be. We primarily use 2000 decodings per level of noise due to computational limitations. Due to how similar the charts for bit error rates and word error rates are, we usually only show the word error rates as to avoid seemingly duplicate charts.

4.3 Impact of flooding iterations for BPSG

When measuring the performance of BPSG, we need to select a set number of flooding iterations. In order to compare BPSG fairly with ILC, we should select y amount of flooding iterations such that BPSG performs as good as possible. The codes we will primarily focus on is nested-clique structures,

where BPSG struggles due to the many short cycles. In these graphs, without doing any LC, the nodes quickly confirm their original beliefs due to their soft information being amplified through short cycles. In other words, doing more flooding iterations may not particularly improve the results for BPSG, but only reaffirm the original information. Figures 14 and 15 shows how flooding iterations impact the decoding performance of BPSG. For the 6-node nested-clique, the resulting decoded words does not improve after 10 flooding iterations. For the 20-node nested-clique, the performance does not improve past 5 flooding iterations, likely due to the many short cycles in this larger graph. Since there is no indication that using more flooding iterations decreases the performance, it is better to do too many iterations than too few.

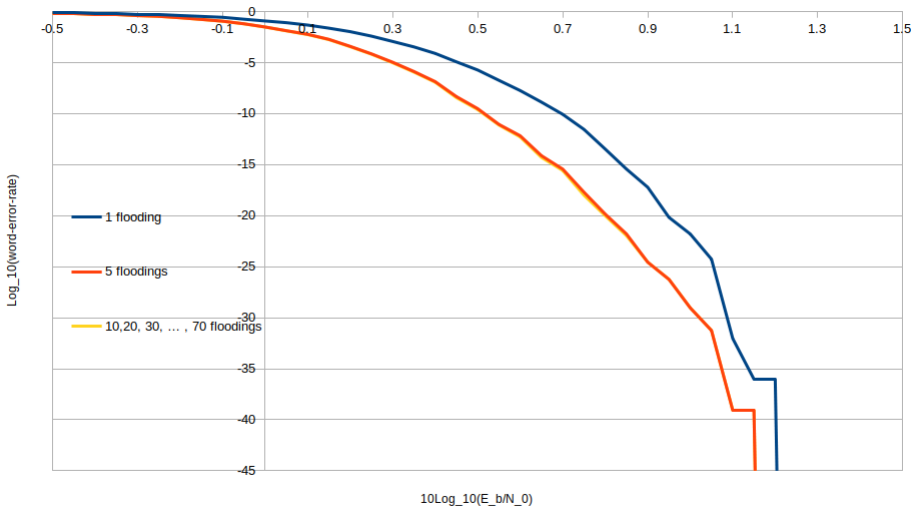


Figure 14: Word error rates for the 6-node nested-clique using BPSG

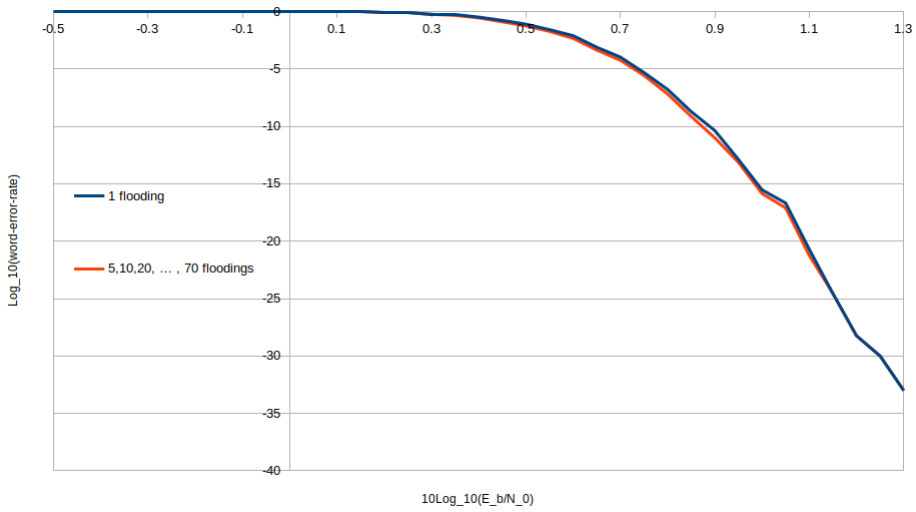


Figure 15: Word error rates for the 20-node nested-clique using BPSG

4.4 Decoding using ILC

When testing the performance of ILC and comparing it to BPSG, we need to decide how many LC iterations and flooding iterations should be performed. Due to dealing with graphs containing cycles, there is no clear indicator as to when decoding is complete. With the results from the previous section, 50 flooding iterations should be more than enough for BPSG. For ILC, we need to select both z LC iterations and y flooding iterations, such that the total amount of floodings across all graph-representations is $y \cdot z$. For the initial runs of ILC, we selected 50 LC operations with 10 floodings each. We later do an analysis of how the number of floodings and LC operations impact the decoding performance of ILC. Decoding on the 6-node nested-clique as seen in Figure 4 using $BPSG(50)$ and $iterativeLC(50, 10)$ gave the results in Figures 16 and 17.

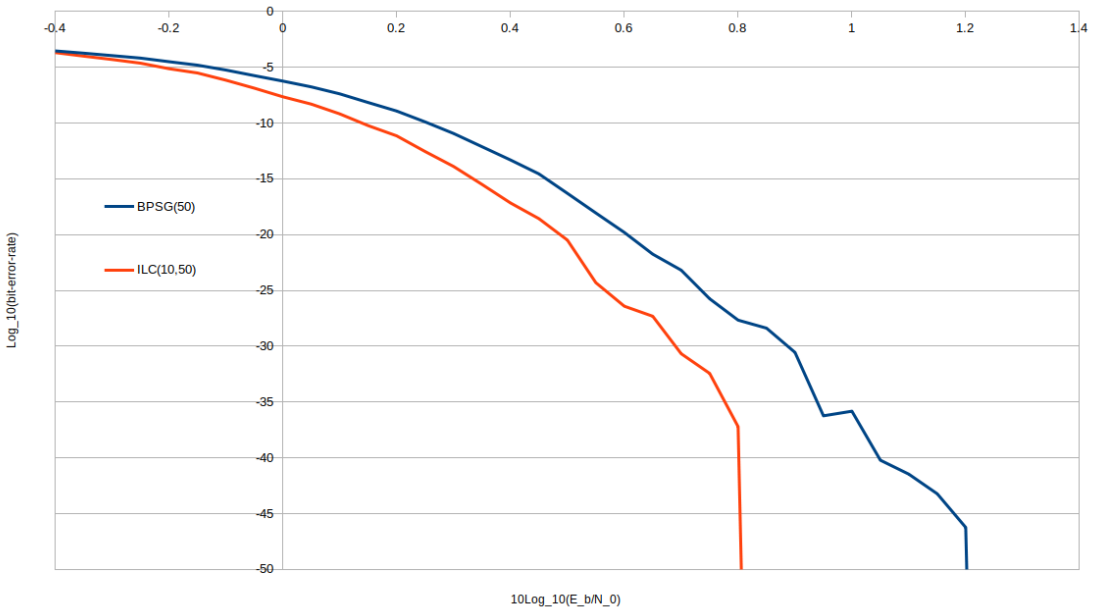


Figure 16: Bit error rates for the 6-node nested-clique, comparting BPSG(50) and ILC(10,50)

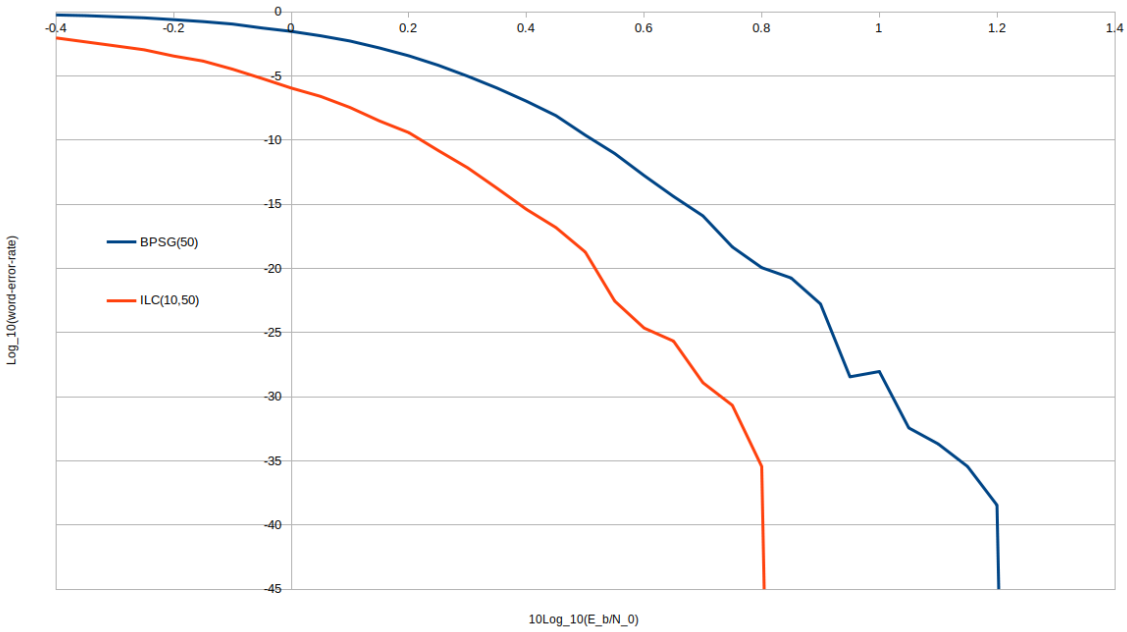


Figure 17: Word error rates for the 6-node nested-clique, comparting BPSG(50) and ILC(10,50)

One can clearly see the improvement for ILC for all signal-to-noise ratios < 1.2. The LC-decoder reaches zero error rate for all $10 \cdot log_{10}(\frac{E_b}{N_0}) \geq 0.85$, where as BPSG does not achieve zero error rates until $10 \cdot log_{10}(\frac{E_b}{N_0}) \geq 1.25$. The results show that local complementation can be used in order to improve the decoding of nested-clique structures.

4.5 Performance impact of LC iterations

In addition to the 6-node nested-clique, ILC have been tested on larger graphs such as a 12-node nested-clique and a 20-node nested-clique. The word error rates of ILC compared to BPSG for these graphs can be seen in Figures 18 and 19. The data has been sampled using the same 50 LC iterations with 10 flooding each.

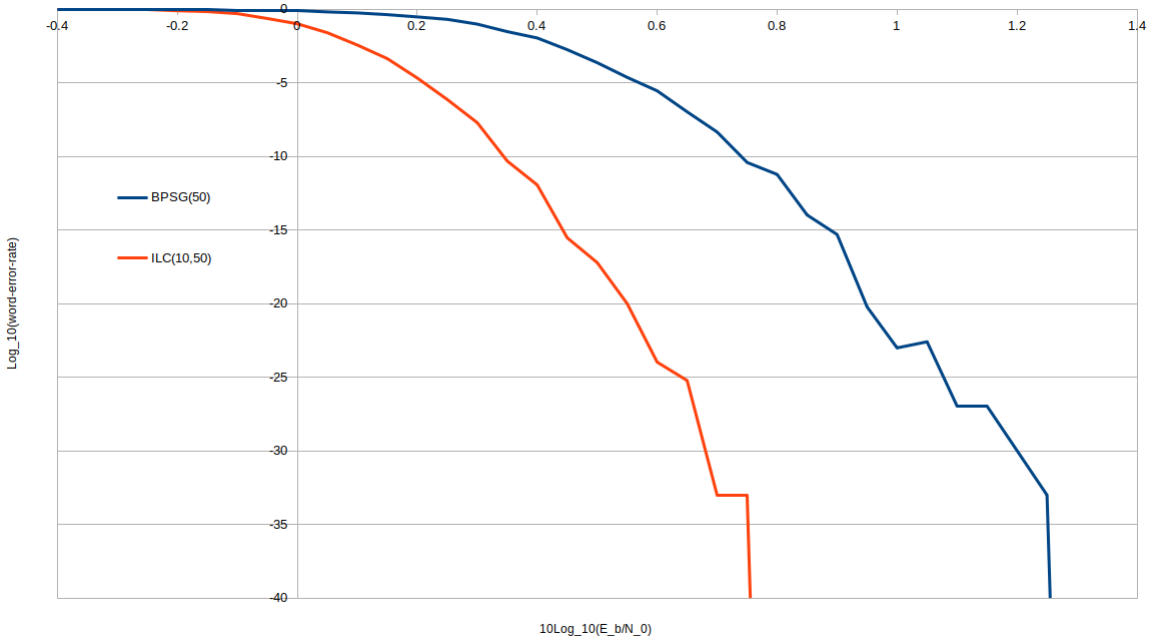


Figure 18: Word error rates for the 12-node nested-clique, comparting BPSG(50) and ILC(10,50)

Though the 12-node nested-clique shows great improvements for ILC, the improvements seems to diminish on the 20-node nested-clique. We would argue that the 50 LC operations does not utilize enough graphs in the LC-orbit of the 20-node graph. Figure 20 shows how the error correcting performance of the 20-node graph is increased as the amount of LC iterations increases. A particular point of interest in Figure 20, is where the version doing 150

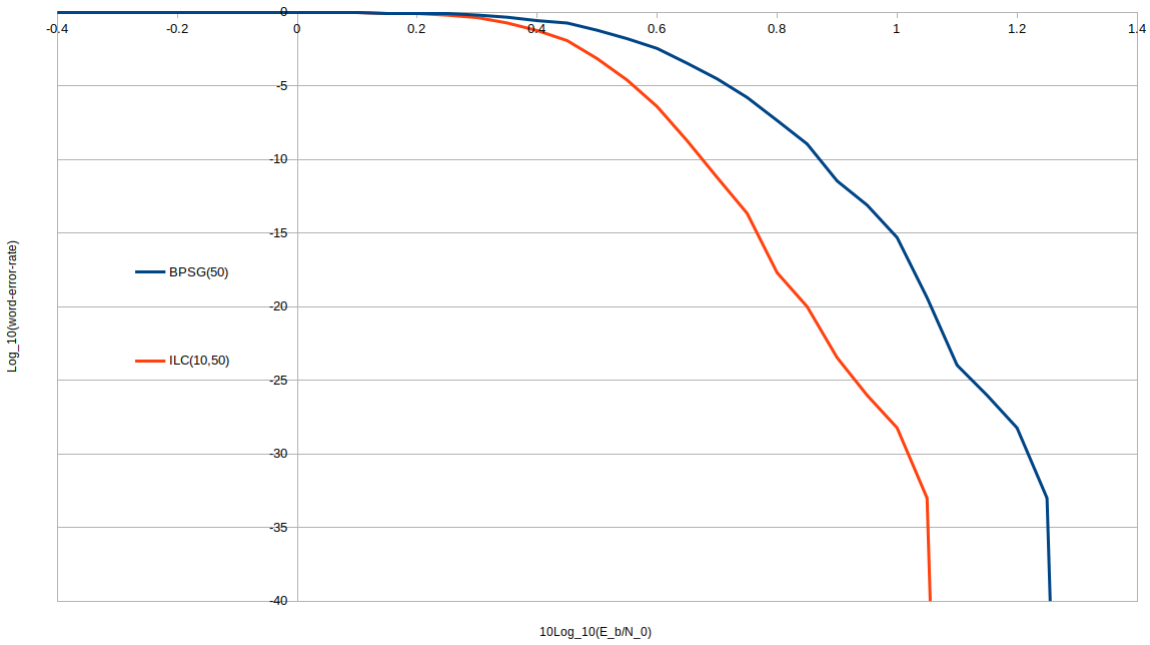


Figure 19: Word error rates for the 20-node nested-clique, comparting BPSG(50) and ILC(10,50)

LC performs better than the version that does 200. This might tie into how we indiscriminately use every graph encountered when iteratively applying LC. One way it could be explained, is if the 50 extra graphs utilized has the problematic nodes more heavily connected in short cycles. In Section 4.8 we look into how the different graphs in the LC-orbit of nested-cliques performs and how one can potentially avoid using graphs that has more cycles than the original nested-clique. Doing an increased number of LC iterations naturally comes at an increased computational cost. The computation times of the different methods on the 20-node nested-clique can be seen in Figure 21. Each data-point represents the average computation time of a single decoding in seconds. These results indicate that the computation cost increases linearly with the amount of LC iterations. In Section 4.7 we show the runtime upper bounds for BPSG and ILC.

We have seen that the performance of ILC depends on the size of the graph and the corresponding number of LC iterations. Though the error rates can be improved using more LC iterations, it also increases the computational cost.

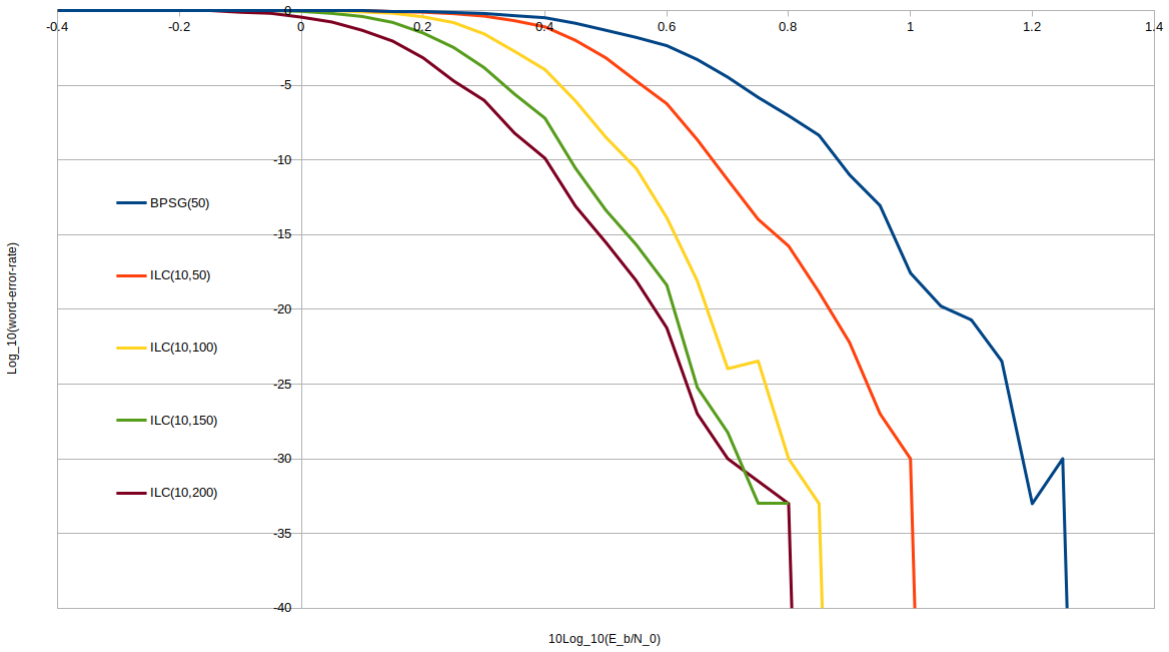
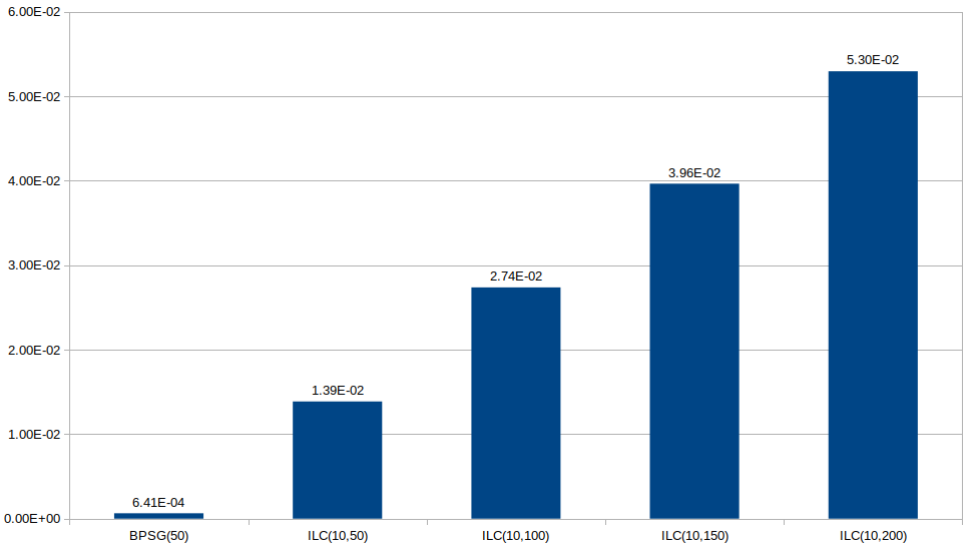


Figure 20: Word error rates for the 20-node nested-clique, comparing LC iterations

4.6 Performance impact of flooding iterations

In order to get a better understanding of ILC, it is interesting to look at how the amount of floodings per LC iteration impacts the error correction performance. In the early stages of development, several numbers of flooding iterations were selected in the 5-50 range. Since the different values for flooding iterations seemed to have little impact on the performance, 10 flooding iterations was used in order to sample the early results. Due to the nature of how the amount of floodings per LC iteration impacts runtimes, we now measure its impact more accurately. Figure 22 has been created by doing 50 LC operations with 1-16 flooding iterations on the 12-node nested-clique. Again, the BPSG performs 50 flooding iterations on a single graph.

We see in Figure 22 that even though 1 flooding iteration does not improve the decoding performance, only doing 2 flooding iterations shows to perform almost as good as doing 4 and more. Performing 8 and 16 flooding iterations per LC operation yields almost the exact same error rates as doing 4, only with some almost indistinguishable variations. Doing more than 4 flooding iterations shows to be redundant, and only having to do 2-4 flooding iterations can significantly improve the runtime. Keep in mind that



!h

Figure 21: Average runtimes for a single decoding on the 20-node nested-clique in seconds.

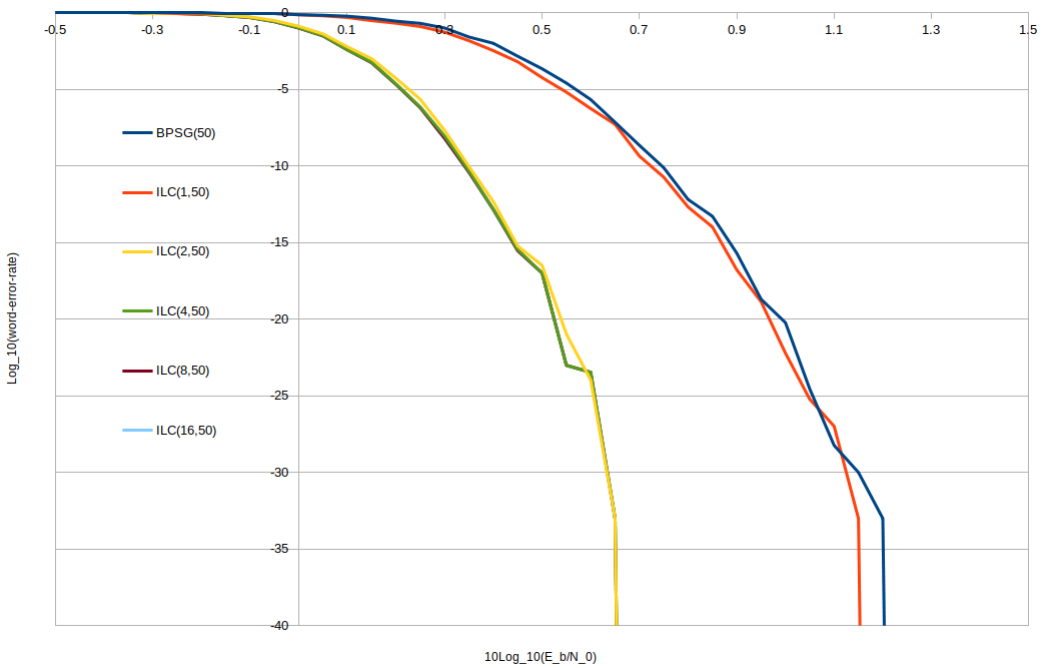


Figure 22: Word error rates for the 12-node nested-clique, comparing flooding iterations

these results are for the 12 node nested-clique, though similar results can be observed for the 6 and 20 node nested-clique, graph structures other than nested-cliques may require higher amounts of flooding iterations based on how far messages has to be propagated in order for all nodes to have received data from the entire graph. For the nested-clique structures, all nodes are a maximum of 2 edges away from all other nodes in the graph. Nodes have distance $d = 1$ to the nodes in its own clique, and has distance $d = 1$ to one of the nodes in all other cliques. As such, the distance between any two nodes in a nested-clique is $d \leq 2$. This means that after two flooding iterations, all nodes will have received soft information from all other nodes. This could be the reason using two flooding iterations per graph in ILC shows great improvements over BPSG.

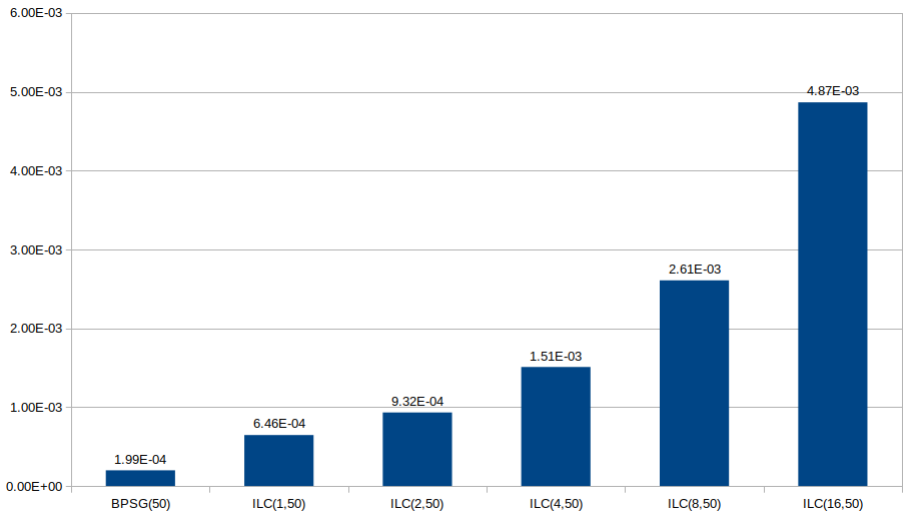


Figure 23: Average runtimes for a single decoding on a 12-node nested-clique in seconds

The runtimes from this experiment can be seen in Figure 23. Again, each number is the average runtime for a single decoding in seconds. We see that one can save quite a bit of computation time by selecting a smaller amount of flooding iterations without having to sacrifice decoding performance.

4.7 Runtime analysis

Let $G = (V, E)$ be the graph representing the self-dual $\mathbb{F}_4$-additive code. The runtime of ILC and BPSG is dependent on the methods they use. Both uses the same functionality for computing messages, marginals and flooding the graph. In the worst case scenario, $computeMessage()$ will have to consider

messages from all other $|V|$ nodes and the run time is $O(|V|)$. $marginal()$ has to consider the maximum of $|V|$ messages and therefore has the same upper bound. The runtime of $flood(y)$ depends on whether we consider the amount of flooding iterations y constant or non-constant. The flooding computes two messages per edge in the graph and has the runtime $O(y2|V| \cdot |E|) = O(y|V||E|))$. BPSG does y flooding iterations and then samples the marginals of all the nodes in order to get the most likely codeword. The full runtime of BPSG is therefore $O(y|V||E|+|V|^2)$. ILC extends BPSG utilizing several additional algorithms. A complete list of all the runtimes can be seen in Table 17. The runtime of ILC again depends on whether we consider the amount of LC and flooding iterations constant or non-constant. Figure 22 shows why one might consider the amount of flooding iterations a small constant, however it is shown in Section 4.5 that the amount of LC iterations needs to increase for larger graphs, which makes the amount of LC operations an important factor. $iterativeLC(y, z)$ performs one $flood(y)$ and one $LC(v)$ per z LC iterations. A single iteration of $LC(v)$ look through a maximum of $|V|^2$ pairs of nodes and check whether they have an edge between them. It also needs to compute the marginals for all $|V|$ nodes. $LC(v)$ therefore has an upper bound of $O(|V|^2 + |V|^2) = O(|V|^2)$. Since $iterativeLC(x, y)$ does one $LC(v)$ and one $flood(y)$ each of the z LC iterations, the total upper bound for $iterativeLC(y, z)$ is $O(z(|V|^2 + y|V||E|))$.

Algorithm	Runtime complexity
$createMessage(recipient)$	$O(\|V\|)$
$marginal()$	$O(\|V\|)$
$flood(y)$	$O(\|V\|\|E\|)$, $O(y\|V\|\|E\|)$
$BPSG(y)$	$O(\|V\|\|E\| + \|V\|^2)$, $O(y\|V\|\|E\| + \|V\|^2)$
$marginalToSoftInformation()$	$O(\|V\|^2)$
$clearMessages()$	$O(\|V\|)$
$softChange(v)$	$O(\|V\|)$
$graphChange()$	$O(\|V\|^2)$
$LC(v)$	$O(\|V\|^2)$
$iterativeLC(y, z)$	$O(\|V\|^2 + \|V\|\|E\|)$, $O(z(\|V\|^2 + y\|V\|\|E\|))$

Table 17: Algorithm runtimes. Black text: y,z assumed constant. Blue text: y,z assumed non-constant.

Overall, ILC is slightly more computationally expensive than the BPSG, the difference in runtime is heavily dependent on the amount of z LC iterations. From Section 4.5 we know that one can expect to do many more LC

iterations than there are nodes or edges in the graph. It could in theory be possible to identify the optimal amount of z LC iterations as a function of V and E, which again stresses the importance of z in $iterativeLC(y, z)$. The notion of whether the iterative-LC decoder is worth the additional computation time will be based on how much noise there is on the quadratic channel and whether the additional computation time is problematic for the precise use-case. One of the main reasons we decode using belief propagation is that it is far less expensive then decoding directly from the global function. Using the global function, variables needs to calculate their marginals based on every codeword. Since a graph code of size $|V|$ has $2^{|V|}$ codewords, the total computation time decoding directly from the global function would be $|V|2^{|V|}$. Though ILC is more computationally expensive than BPSG, it is still far less expensive than to decode directly from the global function.

4.8 Decoding using one LC

In order to get a better understanding of how LC iterations impact decoding performance, we now take a look at decoding using just one LC iteration. Given a graph G, we apply $LC_0(G) = G'$ and do flooding iterations only on G' before converting it back to the original code. Using this scheme, we can look at the performance difference between decoding on trees and decoding on cliques. Consider the graph of six nodes with edges forming a clique. We know that performing LC on any node of this graph will turn it into a tree as seen in Figure 24.

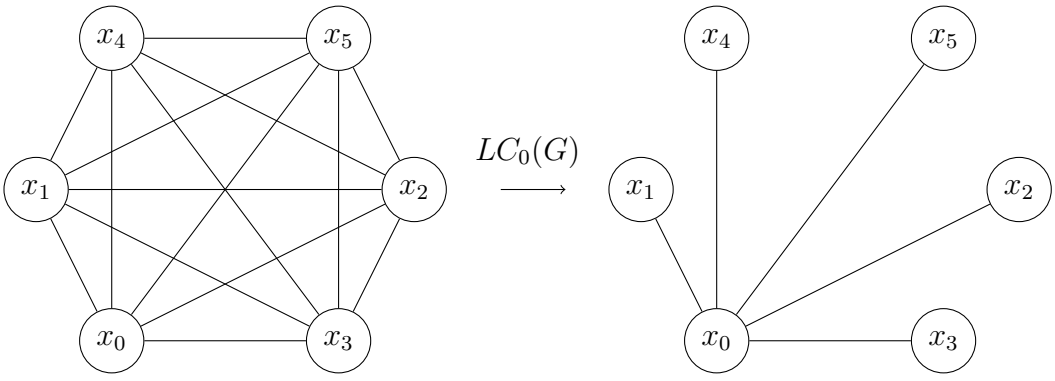


Figure 24: Performing LC on a 6-node clique

Due to the problems BPSG scheme runs into when dealing with cycles, we are interested in seeing if the error correction can be improved by using LC and decode in the corresponding tree. Using the graph code forming a 6-node clique we can compare doing 50 flooding iterations directly on the

clique to doing 50 floodings on the tree acquired by doing LC. Figure 25 shows the results of comparing the two methods for different signal-to-noise ratios. The direct use of BPSG achieves zero error rate for signal-to-noise ratios $10 \cdot log_{10}(\frac{E_b}{N_0}) \geq 1.2$, where as using LC and operating on the corresponding tree structure achieves zero error rate at $10 \cdot log_{10}(\frac{E_b}{N_0}) \geq 0.95$. We can again see that the decoding performance can be improved by operating on different graphs in the LC-orbit of the graph code.

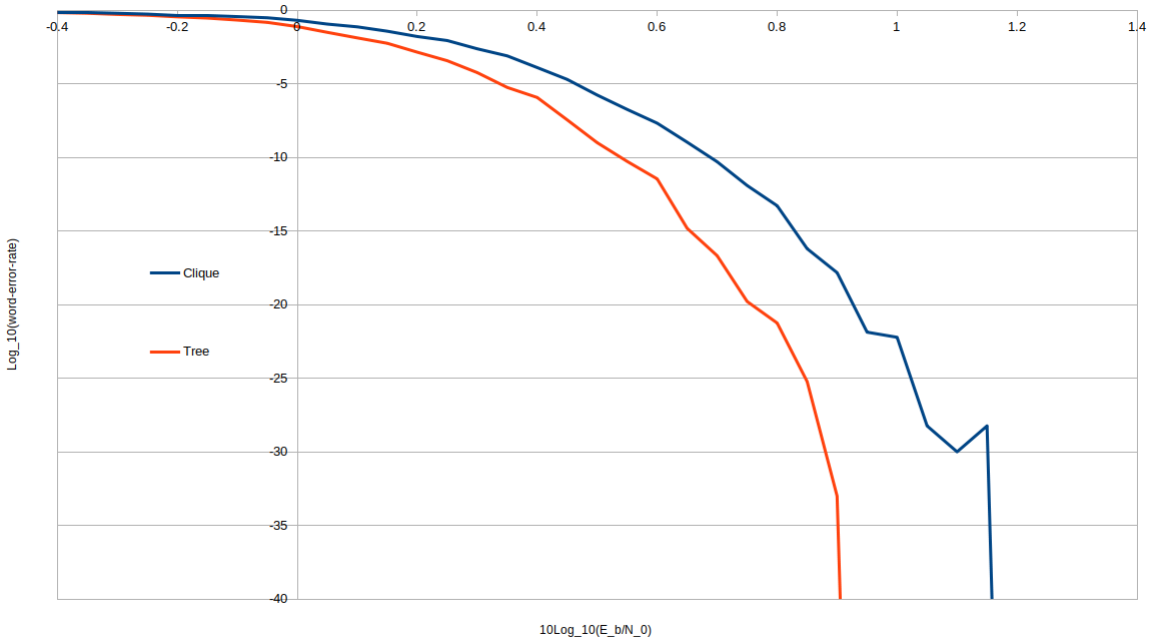


Figure 25: Word error rates for the 6-node clique, comparing decoding directly on the clique, to decoding on the tree acquired through LC.

When decoding on nested-clique graph codes we do not have the property that a single LC operation turns the graph into a tree. Instead, most LC operations on nested-cliques changes where the cycles are located on the graph. As expected, when decoding on a graph one LC operation away from the original nested-clique, we do not necessarily see better error rates. In some cases the LC operation may result in a graph containing more cycles than the original nested-clique, resulting in a decrease in the decoding performance.

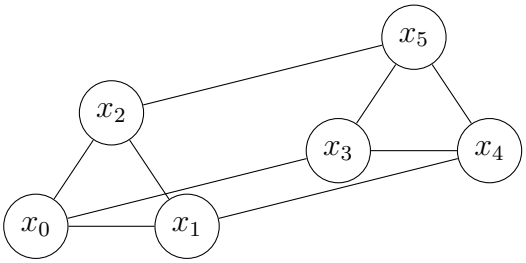


Figure 26: 6 Node Nested-clique

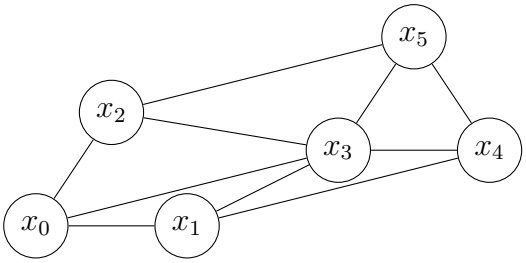


Figure 27: $LC_0(G)$ where G is the 6-node nested-clique

Consider the 6 node nested-clique G as seen in Figure 26. Performing $LC_0(G)$ on this graph creates the graph G' in Figure 27. It can be observed that G' contains one more edge than G and that the node x_3 has become connected to every other node in the graph. When we compare the decoding on G and G' using the same method as for the clique, the word error rates seen in Figure 28 show that G' has worse error rates than that of the original G.

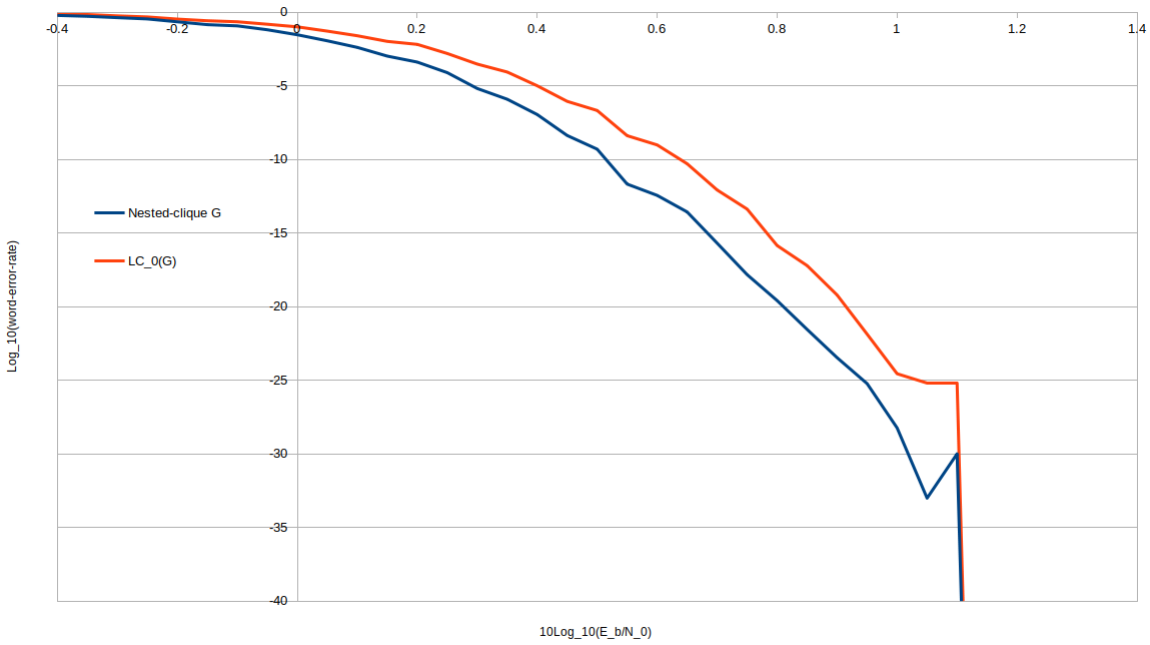


Figure 28: Word error rates for the 6-node nested-clique, comparing decoding directly on the nested-clique, to decoding on G' from $LC_0(G)$.

This means that using a single LC operation and decoding solely on that graph does not solve the problems with cycles for stronger nested-clique graph codes. An idea for potential improvement of the ILC is to find a method that does not indiscriminately use every graph that is encountered, but only operates on graphs containing the same amount of edges or less than the original graph.

5 Conclusion

We have shown how one can perform belief propagation in simple graphs (BPSG) directly on the graphs relating to self-dual $\mathbb{F}_4$-additive codes, without having to discriminate between leaves and internal nodes. In graphs containing many cycles, BPSG's ability to correct errors is shown to be poor. As a means to improve on the decoding on strong codes containing many short cycles, we have developed the extension iterative local complementation (ILC). Instead of performing all message-passing on the single graph related to the selected code, ILC uses many graphs in the LC-orbit of the code. By applying message-passing in many different graphs, ILC avoids passing messages through the same cycles. We have shown that ILC can successfully correct errors where BPSG can not. By sampling soft information from our channel model, and decoding it using BPSG and ILC, we have gathered empirical evidence that the decoding can be improved by performing belief propagation on several different graphs in the LC-orbit of the original graph. When it comes to the parameters for flooding iterations and LC iterations in ILC, the empirical evidence suggests that the amount of LC iterations needs to increase as the graphs gets larger. We do not find evidence for needing to increase flooding iterations in ILC for larger nested-cliques. Though ILC is shown to give better error rates than BPSG over a significant range of signal-to-noise ratios, it also comes at an increased computational cost. Though ILC is more computationally heavy then BPSG, it is still far less expensive than to decode directly from the global function. In conclusion, we have improved the decoding performance of belief propagation in $\mathbb{F}_4$, while still having lower runtime complexity than the global function.

6 Future Work

6.1 Selective LC

The decoding of graph codes using LC in this thesis is performed using an iterative decoding scheme where every graph that is encountered is used for belief propagation. In Section 4.8 it is shown that performing belief propagation on some graphs of an LC-orbit is less effective than others. It remains to be seen whether one can further improve the error rates by being more selective as to what graphs are utilized with belief propagation.

6.2 Estimate optimal LC iterations based on size of graph

In Section 4.5 we saw that the amount of LC iterations needed for good performance improvements over BPSG increased as the graphs grew larger. We saw that the amount of LC iterations providing satisfying improvements over BPSG could get larger than $|V|$. In order to be able to use ILC for any nested-clique G, it would be beneficial to have a function calculating an optimal amount of LC iterations based on $|V|$ and $|E|$.

6.3 Codes over $\mathbb{F}_9$

It would be interesting to see how one might perform belief propagation for codes over other finite fields than $\mathbb{F}_4$. It has been shown that every self-dual additive code over $GF(p^2)$ is equivalent to a graph code [7]. A natural continuation would be to study $\mathbb{F}_{3^2}$. This field is defined to consist of all polynomials in x with ternary coefficients and degree at most 1, with calculations performed modulo the irreducible polynomial $p(x) = x^2 + 1$ [16]. For self-dual $\mathbb{F}_9$-additive codes, we can not use the same definitions of dSX and dSS as they are only defined for $\mathbb{R}^4$. Instead, if decoding is to be performed similarly to the decoding of the graph codes in this thesis, new vector-products for vectors $u, v \in \mathbb{R}^9$ should be devised based on how one can reduce the marginals of these codes. The goal of studying other fields such as $\mathbb{F}_9$ would be to find similarities to $\mathbb{F}_4$ and see if one can generalize decoding for any field $\mathbb{F}_{p^2}$

6.4 Non self-dual codes

An interesting topic not looked into in this thesis is additive codes over $\mathbb{F}_4$ that are *not* self-dual. These codes are different in that their parity check matrix will differ from their generator matrix, and that the graphs associated to the parity check matrix will be directed. It would be interesting to see how one might perform belief propagation for such graphs.

References

[1] Java language specification, 15.17.1. multiplication operator *. `https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.17.1`. Accessed: 2019-05-27.

[2] Java language specification, 15.18.2. additive operators (+ and -) for numeric types. `https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.18.2`. Accessed: 2019-05-27.

[3] Sergio Benedetto and Ezio Biglieri. *Principles of digital transmission: with wireless applications*. Springer Science & Business Media, 1999.

[4] Claude Berrou, Alain Glavieux, and Punya Thitimajshima. Near shannon limit error-correcting coding and decoding: Turbo-codes. 1. In *Proceedings of ICC'93-IEEE International Conference on Communications*, volume 2, pages 1064–1070. IEEE, 1993.

[5] George EP Box. A note on the generation of random normal deviates. *Ann. Math. Stat.*, 29:610–611, 1958.

[6] Lars Eirik Danielsen. On self-dual quantum codes, graphs, and boolean functions. *arXiv preprint quant-ph/0503236*, 2005.

[7] Lars Eirik Danielsen. On connections between graphs, codes, quantum states, and boolean functions. 2008.

[8] Lars Eirik Danielsen and Matthew G Parker. Spectral orbits and peak-to-average power ratio of boolean functions with respect to the {I, H, N} n transform. In *International Conference on Sequences and Their Applications*, pages 373–388. Springer, 2004.

[9] Lars Eirik Danielsen and Matthew G Parker. On the classification of all self-dual additive codes over gf (4) of length up to 12. *Journal of Combinatorial Theory, Series A*, 113(7):1351–1367, 2006.

[10] John B Fraleigh. *A first course in Abstract Algebra*. Pearson, 2014.

[11] Robert Gallager. Low-density parity-check codes. *IRE Transactions on information theory*, 8(1):21–28, 1962.

[12] Hanna A Hansen. Dynamic message-passing decoding on simple graphs for the quaternary symmetric channel. Master's thesis, Bergen University College, University of Bergen, June 2016.

[13] Joakim Grahl Knudsen. On iterative decoding of high-density parity-check codes using edge-local complementation. 2010.

[14] Frank R Kschischang, Brendan J Frey, and H-A Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on information theory*, 47(2):498–519, 2001.

[15] Frank R Kschischang, Brendan J Frey, Hans-Andrea Loeliger, et al. Factor graphs and the sum-product algorithm. *IEEE Transactions on information theory*, 47(2):498–519, 2001.

[16] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error-correcting codes*. Elsevier, 1977.

[17] Matthew G Parker. Graph-based inference, networks and coding theory, autumn 2016, project. `http://www.ii.uib.no/~matthew/INF244/INF24416Project.pdf`. Accessed: 2019-05-27.

[18] Matthew G Parker. Dynamic message-passing decoding on simple graphs for the quaternary symmetric channel. Unpublished Manuscript, 2012.

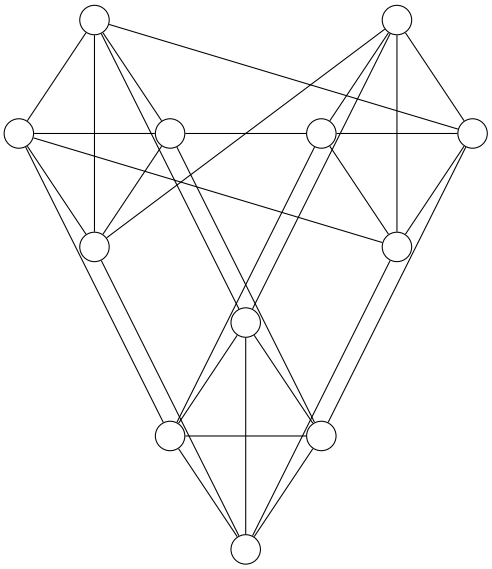


Figure 29: 12-node nested-clique