Doctoral Thesis

Multitree search decoding of linear codes

Author(s):
Ostojic, Maja

Publication Date:
2010

Permanent Link:
https://doi.org/10.3929/ethz-a-006391879

Rights / License:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.
Multitree Search Decoding of Linear Codes

A dissertation submitted to
ETH ZURICH

for the degree of
Doctor of Sciences ETH Zurich

presented by
MAJA OSTOJIC

dipl. El.-Ing. ETH
born August 6, 1976
citizen of Lenzburg AG

accepted on the recommendation of
Prof. Dr. Hans-Andrea Loeliger, examiner
Prof. Dr. Daniel J. Costello, Jr., co-examiner

2010
Abstract

Tree search algorithms have a long history in computer science. In the coding literature, tree search algorithms have traditionally been used for decoding convolutional codes. Convolutional codes are linear codes with a special structure. Classic tree search decoders (most notably the sequential decoder) search one code tree in which the bits are ordered sequentially.

We propose a multitree search decoder for arbitrary linear codes. We develop several algorithms for constructing code trees from the parity check matrix of a linear code. We propose algorithms that generate code trees, in which the bits appear in random order. We also show how to generate code trees specially designed to decode a given sequence received from a noisy channel. In such code trees, the ordering of the bits depends on the received sequence. Multiple code trees can be generated for each code and sequence. Specialized code trees for low-density parity check codes are also presented.

The different code trees are explored with a new search algorithm. The algorithm is similar to the M-algorithm for convolutional codes; it explores a code tree with limits on the breadth of the explored subtree. An evaluation function is used to decide which node to expand at each depth. We present an evaluation function for general linear codes and an improved evaluation function for low-density parity check codes. Both are optimized for the proposed search algorithm.

The proposed multitree search decoder achieves near optimal performance for short block codes.

For longer block lengths, we propose to use tree search decoding to
improve the standard sum-product decoder for low-density parity check codes. When the sum-product decoder fails to find a codeword, a tree search is used to decode a subset of bits. The channel messages for these bits are then replaced by the decisions found in the tree search in an additional sum-product decoding attempt. This can be repeated multiple times for different subsets of bits. The resulting decoder significantly outperforms the sum-product decoder.

Keywords: Linear codes, low-density parity check codes, tree search, branch and bound, informed search, depth-first search, A*-search, best-first search, convolutional codes, sequential decoding.
Kurzfassung


Der vorgestellte Dekodierer erreicht fast optimale Fehlerraten bei Co-

**Stichworte:** Lineare Codes, Low-Density-Parity-Check-Codes, Suchalgorithmen in Bäumen, Branch-and-Bound, Informierte Suche, $A^*$-Suche, Best-First-Search, Faltungs codes, Sequentieller Dekodierer.
Contents

Abstract iii

Kurzfassung v

1 Introduction 1
  1.1 Motivation .................................................. 1
  1.2 Contributions ................................................. 2
  1.3 Outline ....................................................... 3

2 Background Material 5
  2.1 Linear Codes .................................................. 5
    2.1.1 Low-Density Parity Check Codes ......................... 6
    2.1.2 Random \((p = 1/2)\)-Parity Check Codes ............. 7
  2.2 Model for Digital Transmission Over a Noisy Channel .... 7
    2.2.1 The Additive White Gaussian Noise Channel .......... 8
    2.2.2 Channel Decoding ......................................... 10

3 Codeword Estimation as a Tree Search Problem 13
  3.1 Outline ....................................................... 13
  3.2 Code Trees ................................................... 14
  3.3 Search Algorithms on Code Trees ............................ 15
    3.3.1 \(A^*\)-Search ............................................ 16
    3.3.2 Forward Sweep Search .................................... 17
  3.4 Multitree Search ............................................. 18

4 Code Trees for Linear Codes 21
  4.1 From A Parity Check Matrix To A Code Tree ............... 21
  4.2 Algorithms for Constructing Code Trees for General Lin- 24
    ear Codes ....................................................... 24
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.1</td>
<td>General-Code Tree Algorithm</td>
<td>24</td>
</tr>
<tr>
<td>4.2.2</td>
<td>General-Code Tree Algorithm With Reliability Ordering</td>
<td>25</td>
</tr>
<tr>
<td>4.3</td>
<td>Algorithms for Constructing Code Trees for Low-Density Parity Check Codes</td>
<td>25</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Low-Density Code Tree Algorithm</td>
<td>26</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Low-Density Code Tree Algorithm With Reliability Ordering</td>
<td>27</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Selection of Precomputed Low-Density Code Trees</td>
<td>29</td>
</tr>
<tr>
<td>4.4</td>
<td>Simulated Average Code Tree Properties</td>
<td>29</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Simulated Structure of General-Code Trees</td>
<td>30</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Simulated Structure of Low-Density Code Trees</td>
<td>31</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Comparison</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>Evaluation Functions</td>
<td>35</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>35</td>
</tr>
<tr>
<td>5.2</td>
<td>The Basic Evaluation Function</td>
<td>36</td>
</tr>
<tr>
<td>5.3</td>
<td>The Greedy Evaluation Function</td>
<td>37</td>
</tr>
<tr>
<td>6</td>
<td>Multitree Search Decoding with the Basic Evaluation Function</td>
<td>39</td>
</tr>
<tr>
<td>6.1</td>
<td>Simulation Procedure</td>
<td>39</td>
</tr>
<tr>
<td>6.2</td>
<td>Simulated Word Error Rates for a (3,6)-regular LDPC Code of Length n = 100</td>
<td>41</td>
</tr>
<tr>
<td>6.3</td>
<td>Simulated Word Error Rates for a (p = 1/2)-Code of Length n = 100</td>
<td>41</td>
</tr>
<tr>
<td>6.4</td>
<td>Comparison of Simulated Word Error Rates for Different Codes of Length n = 100</td>
<td>43</td>
</tr>
<tr>
<td>6.5</td>
<td>Simulated Word Error Rates for a (3,6)-regular LDPC Code of Length n = 200</td>
<td>45</td>
</tr>
<tr>
<td>7</td>
<td>Multitree Search Decoding of LDPC Codes with The Greedy Evaluation Function</td>
<td>49</td>
</tr>
<tr>
<td>7.1</td>
<td>Simulated Word Error Rates for a (3,6)-regular LDPC Code of Length n = 100</td>
<td>49</td>
</tr>
<tr>
<td>7.2</td>
<td>Simulated Word Error Rates for a (3,6)-regular LDPC Code of Length n = 200</td>
<td>50</td>
</tr>
<tr>
<td>8</td>
<td>Multitree Search Aided LDPC Decoding</td>
<td>55</td>
</tr>
<tr>
<td>8.1</td>
<td>Introduction</td>
<td>55</td>
</tr>
<tr>
<td>8.2</td>
<td>Simulated Tests With an Error-Free Tree Search Decoder</td>
<td>56</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>8.3 Single Bit Flipping Aided LDPC Decoding Algorithms</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>8.4 Tree Search Aided LDPC Decoding Algorithms</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>8.5 Simulation Results</td>
<td>67</td>
<td></td>
</tr>
<tr>
<td>9 Discussion and Conclusion</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>A The Sum-Product Decoder</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>A.1 Factor Graphs</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>A.2 The Sum-Product Algorithm</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>A.3 Iterative Channel Decoding</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>List of Figures</td>
<td>83</td>
<td></td>
</tr>
<tr>
<td>Bibliography</td>
<td>91</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Low-density parity check codes have been shown to asymptotically approach capacity for the additive white Gaussian noise channel under iterative decoding [1]. However, for short block lengths, the decoding loss due to the iterative decoder is considerable. Various near-optimal soft-decision decoding techniques have been recently developed for decoding short codes (an overview can be found in [2]). We propose the use of tree search for decoding short linear codes.

Tree search algorithms have a long history in computer science (an introduction can be found e.g. in [3]). In the coding literature, tree search algorithms have traditionally been used for decoding convolutional codes. The first such decoder was the sequential decoder (first introduced by Wozencraft in 1957 [4]), which was later significantly improved with the introduction of the Fano metric [5] and the stack algorithm [6]. While numerous variations on the sequential decoder have been developed since (see for instance [7]), the basic algorithm is an $A^*$-search algorithm with an evaluation function. A related tree search algorithm for decoding convolutional codes is the M-algorithm (a breadth-limited search algorithm). A comprehensive introduction to both convolutional codes and decoding algorithms for convolutional codes can be found in [8].
Tree search algorithms for decoding linear block codes have been proposed in [9], [10], [11] and [12]. Each of the proposed algorithms is a maximum-likelihood decoder ($A^*$-search on a code trellis). Because of the prohibitive worst-case complexity of the maximum-likelihood decoder, the proposed decoders work only for short block lengths and high signal-to-noise ratios. The authors of [9], [10] and [12] observed that the ordering of the received symbols in the trellis has a notable effect on the decoding complexity. Fewer nodes are visited during the search if symbols with high reliability appear first in the trellis. In [11], Kschischang et al introduced the cumulative distance function for linear block codes. They showed that the decoding complexity of $A^*$-search decoding is smaller if trellises with quickly growing cumulative distance functions are used. This result is analogous to the behavior of the sequential decoder for convolutional codes, which is known to work better for codes with quickly growing distance profile.

In this thesis, we develop suboptimal tree search decoders for linear block codes. The decoders achieve near-maximum-likelihood error rates for short codes with significantly lower complexity than maximum-likelihood decoders. We introduce a novel search algorithm which, unlike $A^*$-search, has known complexity (which is controlled by a free parameter). We show that multitree search outperforms single tree search. We also show how tree search decoders can be used to enhance the iterative decoder for low-density parity check codes when decoding longer codes, resulting in significantly smaller error rates.

1.2 Contributions

In this thesis, we present tree search algorithms for channel decoding of linear codes. Some of the main contributions are:

- We demonstrate that multitree search algorithms can achieve near-maximum-likelihood error rates for short linear block codes with much smaller complexity than maximum-likelihood decoders.

- A generic method for constructing efficiently searchable code trees for linear codes is presented. We present several algorithms which construct code trees from a linear code’s parity check matrix.
• We introduce a probabilistic method for sorting the received symbol sequence according to symbol reliability. We present an algorithm which uses the sorted channel sequence and a parity check matrix to construct code trees with reliability ordering specifically designed for decoding the given sequence.

• For low-density parity check codes, we develop specialized algorithms which optimize the local rate (and with it the distance profile) as well as the reliability ordering.

• We present an informed search algorithm for decoding linear codes. The number of nodes visited during the search can be tightly controlled by a free parameter. The presented algorithm avoids the comparison of nodes at different depths in the code trees.

• We present evaluation functions which are specifically designed for the presented search algorithm.

• We show that using the proposed algorithm to search multiple code trees in independent decoding attempts outperforms searching only one tree to decode the same received sequence.

• We show how a tree search decoder can be used to substantially improve the performance of the sum-product decoder for low-density parity check codes. Excellent results are achieved for medium to long block lengths.

### 1.3 Outline

This thesis is organized as follows. Chapter 2 briefly reviews the basics of linear codes and data transmission over a noisy channel. Chapter 3 introduces tree search algorithms and outlines how tree search can be used for decoding. In Chapter 4, we present several algorithms for constructing code trees from a parity check matrix and compare the average properties of the code trees for different codes. Evaluation functions for tree search decoding are presented in Chapter 5. Chapter 6 and 7 show some simulation results for different codes using different code trees and evaluation functions. Finally, Chapter 8 presents a multistage sum-product decoder, where tree search decoding is used to enhance the sum-product
decoder. A discussion of the presented work is given in Chapter 9. A brief review of the sum-product decoder is given in Appendix A.
Chapter 2

Background Material

In this chapter, we briefly review linear codes and the model for digital transmission over a noisy channel. The purpose of this chapter is to introduce the notation used in this thesis.

2.1 Linear Codes

We use the notation common in the coding literature where a codeword is a row vector.

Definition 2.1. (Linear code) An \((n, k)\) linear block code \(C\) over a field \(\mathbb{F}\) is a \(k\)-dimensional subspace of \(\mathbb{F}^n\).

The following terms will be used extensively in the thesis.

- **(Generator matrix)** A generator matrix for a linear code \(C\) is a \(k \times n\) matrix \(G\) over \(\mathbb{F}\) such that
  \[ C = \{uG : u \in \mathbb{F}^k\}. \]

- **(Parity check matrix)** A parity check matrix for an \((n, k)\) linear code \(C\) is a \(n \times m\) matrix \(H\) where \(m \geq n - k\) and
  \[ C = \{x \in \mathbb{F}^n : Hx^T = 0\}. \]
• (Rate) The rate \( r \) of a linear code \( C \) is defined as
\[
r \triangleq \frac{1}{n} \log_{\mathbb{F}} |C| = \frac{k}{n}.
\]

• (Hamming weight) The Hamming weight \( w_H(x) \) of a vector \( x \in C \) is the number of nonzero entries in \( x \).

• (Hamming distance) The Hamming distance \( d_H(x_1, x_2) \) between two codewords \( x_1 \) and \( x_2 \) in a linear code \( C \) is defined as
\[
d_H(x_1, x_2) = w_H(x_1 - x_2).
\]

• (Minimum distance) The minimum distance \( d_{\text{min}} \) of a linear code \( C \) is the smallest Hamming distance between any two distinct codewords in \( C \):
\[
d_{\text{min}}(C) = \min_{x_i, x_j \in C: x_i \neq x_j} d_H(x_i, x_j).
\]

In this thesis we will only consider binary linear block codes, i.e., \( \mathbb{F} = \mathbb{Z}_2 \). In the context of code trees (see Section 3.2), we will also use the following terms.

• (Partial codeword) A partial codeword of a linear code \( C \) of length \( n \) is
\[
x_S = \{ x_i : i \in S, S \subset \{1, \ldots, n\}, x \in C \}.
\]

• (Information bit) An information bit is a free parameter in the solution set to \( Hx^T = 0 \), where \( H \) is a parity check matrix of a binary linear code.

### 2.1.1 Low-Density Parity Check Codes

Low-density parity check codes were first presented in [13]. There is a large body of work on low-density parity check (LDPC) codes, a comprehensive introduction can be found in [14]. We only summarize the definitions needed in this thesis.

A low-density parity check code is a code which has at least one parity check matrix which is sparse, i.e., consists mostly of zeroes.
Definition 2.2. (Regular LDPC code) A \((w_{\text{col}}, w_{\text{row}})\)-regular LDPC code has a parity check matrix \(H\) where each row has Hamming weight \(w_{\text{row}}\) and each column has Hamming weight \(w_{\text{col}}\).

Consider a \((3, 6)\)-regular LDPC code of length \(n = 500\) and rate \(r = 1/2\). Only 1.2\% of the of the matrix coefficients are nonzero for a full rank parity check matrix.

In this thesis, we use regular LDPC codes whose parity check matrix is drawn randomly. In some cases, we want a parity check matrix to satisfy an additional constraint, i.e., that the factor graph of the parity check matrix contains no cycles of length 4 (see Appendix A.1). This condition is fulfilled if for any row, the column indices of the nonzero entries in the row have at most one index in common with the column indices of the nonzero entries of any other row.

Such a parity check matrix can be obtained from a random parity check matrix by removing the 4-cycles from the given matrix. We identify the pairs of rows which violate the condition and do some random swaps of the coefficients in the matrix if they remove the constraint violation (and don’t introduce a new constraint violation), until the matrix fulfills the constraint. The matrix obtained defines a different code.

2.1.2 Random \((p = 1/2)\)-Parity Check Codes

A (binary) random \((p = 1/2)\)-parity check code is an unstructured code obtained as follows: each coefficient in the parity check matrix is drawn independently to be one with probability \(p = 1/2\) and zero otherwise. We will use the shorter term “\((p = 1/2)\)-code” instead of “random \((p = 1/2)\)-parity check code” in the following.

2.2 Model for Digital Transmission Over a Noisy Channel

Assume we wish to transmit some data from a sender to a receiver over an unreliable channel. The data reconstructed at the receiver should be
identical to the data sent at the receiver with high probability. We use error correcting codes to minimize the probability of error.

A generic communication system is shown in Figure 2.1. The information symbols $u_1, \ldots, u_k$ are first encoded into a redundant sequence $x_1, \ldots, x_n$ to permit error correction at the receiver. This sequence is sent over the noisy channel. The received sequence $y_1, \ldots, y_n$ is then processed by a decoder, which computes either an estimate $\hat{u}_1, \ldots, \hat{u}_k$ of the transmitted information symbols or announces a decoding failure.

In the following sections we first describe the channel model used in this thesis and then state the channel decoding problem.

### 2.2.1 The Additive White Gaussian Noise Channel

The channel model used in this thesis is the binary input additive white Gaussian noise channel (AWGN channel).

**Definition 2.3. (AWGN channel)** An binary-input AWGN channel accepts input symbols $\tilde{X}_k \in \{+1, -1\}$. The output symbol is $Y_k = \tilde{X}_k + Z_k$, where $Z_k$ is a zero-mean Gaussian random variable with variance $\sigma^2$. If a sequence $\tilde{X}_1, \tilde{X}_2, \ldots$ is sent over an AWGN channel then $Z_1, Z_2, \ldots$ are i.i.d. (the channel is memoryless).

If we use a code over $\mathbb{Z}_2$, we need to first map the channel input bits before we can transmit them over the AWGN channel. The mapper is the function $f : \{0, 1\} \mapsto \{-1, 1\}$:

$$f(X_k) = \tilde{X}_k = \begin{cases} +1 & \text{if } X_k = 0 \\ -1 & \text{if } X_k = 1 \end{cases}$$
2.2. Model for Digital Transmission Over a Noisy Channel

Figure 2.2 shows a model for the AWGN channel including a mapper.

For an AWGN channel with variance $\sigma^2$ and a received sequence $y = (y_1, y_2, \ldots)$ transmitted over the channel, the following derived terms will be used later in the thesis.

- **(Signal-to-noise ratio)** The signal-to-noise ratio (SNR) of the channel is
  \[ \text{SNR} = 10 \log_{10} \left( \frac{1}{\sigma^2} \right). \]

- **(Log-likelihood ratio)** The log-likelihood (LLR) ratio of symbol $y_k$ is
  \[ \text{LLR}(y_k) = \log \left( \frac{p(y_k | x_k = 0)}{p(y_k | x_k = 1)} \right) = \frac{2y_k}{\sigma^2}. \]

- **(Reliability)** The reliability $\theta$ of a received symbol $y_k$ is the absolute value of the LLR, i.e.
  \[ \theta(y_k) = |\text{LLR}(y_k)| = \left| \frac{2y_k}{\sigma^2} \right|. \]

- **(Hard decision)** The hard decision of a symbol is
  \[ \hat{x}_{k, \text{HD}} = \arg\max_{x_k} \log p(y_k | x_k). \]

We will also use the following terminology when a codeword from a linear code $C$ with parity check matrix $H$ was sent over the AWGN channel and a sequence $y$ was received.

- **(Satisfied parity check equation)** A parity check equation $h$ (a row of the parity check matrix $H$) is satisfied if the equation
  \[ h \hat{x}_{\text{HD}}^T = 0 \]
  is satisfied ($\hat{x}_{\text{HD}}$ is the vector of hard decisions).

- **(Maximum-log-likelihood of a parity check)** The maximum log-likelihood of a parity check $h$ with support $s(h)$ is
  \[ (ML)(h) = \max_{h \hat{x}^T = 0} \sum_{i \in s(h)} \log p(y_i | x_i) \]
2.2.2 Channel Decoding

We refine the decoder model from Figure 2.1. Figure 2.3 shows a more detailed model for the decoder. It consists of a codeword estimator and an encoder inverse. Given a statistical channel model, codeword estimation is a statistical detection problem. The error rate of the channel decoder is equal to the error rate of the codeword estimator, since the inverse encoder is a deterministic one-to-one mapping.

In the following we will neglect the inverse encoder and refer to channel decoding as codeword estimation.

**Definition 2.4. (MAP)** The maximum-a-posteriori (MAP) codeword estimator (or decoder) is defined as

\[
\hat{X} = \arg\max_{X \in C} \log p(X|Y) = \arg\max_{X \in C} \log p(X, Y)
\]

The MAP decoder minimizes the block error rate \(P(\hat{X} \neq X)\). A related decoding rule is the maximum-likelihood decision rule.

**Definition 2.5. (ML)** The maximum-likelihood (ML) codeword estimator (or decoder) is defined as

\[
\hat{X} = \arg\max_{X \in C} \log p(Y|X)
\]

If the prior distribution of \(X\) is uniform, then the ML decoding rule is equivalent to the MAP decoding rule. We will assume this to be the case throughout this thesis.
2.2. Model for Digital Transmission Over a Noisy Channel

\[ Y_1, \ldots, Y_n \overset{\hat{X} \in C}{\longrightarrow} U_1, \ldots, U_k \]

**Figure 2.3:** Channel decoder
In this chapter we describe tree searching on an abstract level and show how it can be used for codeword estimation. A general introduction to tree search based problem solving can be found in [3].

3.1 Outline

A tree search problem solver consists of:

1) A parameterization of the search space as a tree.
2) A search algorithm.

The goal of a tree search is to search only a subspace of the search space to find a (possibly optimal) solution. As described in Section 2.2.2, a codeword estimator needs to determine the codeword which was most likely transmitted. Therefore, the search space of a tree search decoder is the set of codewords in a code. We call a tree which represents the codewords of a code a code tree.
We introduce the notation we use for code trees in Section 3.2. In Section 3.3 we describe some properties of search algorithms and review the classical search algorithm $A^\star$-search. We then propose a new search algorithm. Finally, in Section 3.4 we propose using multiple trees instead of only one tree for channel decoding.

## 3.2 Code Trees

We restrict our review of tree graphs to a minimum. We introduce rooted trees informally and then explain how a rooted tree can be used to graphically represent the codewords of a code.

(Rooted tree) Figure 3.1 shows a rooted tree. A tree consists of branches and nodes which are connected such that there are no loops. A unique branch connects a pair of nodes.

- **(Root node)** There is one unique node which is called the root node.

- **(Path)** A path is a sequence of nodes such that from each node in the sequence there is a branch to the next node in the sequence.

- **(Depth)** The depth of the root node is zero. The depth of any other node is equal to the number of branches traversed on the path from the root node to the considered node. (As a convention, we number the depth of branches beginning from 1).

- **(Parent node, child node)** A branch connects a parent node with a child node. The parent node has smaller depth than the child node.

- **(Branching factor)** The branching factor out of a node is equal to the node’s number of children.

- **(Leaf node)** A leaf node is a node with no children.

The tree in Figure 3.1 has 10 nodes. The root node has branching factor 2 and the maximum depth occurring in the tree is 3.
(Code tree) A code tree is a rooted tree which depicts the codewords of a linear code such that there is a one-to-one correspondence between a partial codeword and a node in the tree. Note that many different trees can represent the same code. Figure 3.2 shows a code tree for the code

\[ C = \{(0, 0, 0, 0, 0), (1, 0, 1, 0, 0), (0, 1, 1, 1, 1), (1, 1, 0, 1, 1)\}. \] (3.1)

- (Node) Each node represents an unique partial codeword.
- (Partial codeword inheritance) If node \( A \) with partial codeword \( x_A \) is a parent of node \( B \) with partial codeword \( x_B \), then \( A \subseteq B \) and the partial codewords are identical at position \( A \).
- (Branch labels) If node \( A \) with partial codeword \( x_A \) is a parent of node \( B \) with partial codeword \( x_B \), then the branch which connects nodes \( A \) and \( B \) is labelled with the partial codeword \( x_B \setminus x_A \).
- (Goal node) A goal node is a node which represents a codeword. It is always a leaf node.

3.3 Search Algorithms on Code Trees

We assume that the code tree is defined before tree exploration begins. When searching a code tree, we keep track of the already explored subtree. The search always begins at the root node, therefore the explored
subtree is initialized to contain only the root node. Tree exploration is then done as follows. A leaf node from the explored subtree is selected and its children are added to the explored subtree. This operation is called node expansion (the selected leaf node is expanded). It is repeated until some stopping criterion is met. Tree exploration is mainly controlled by:

1) The node selection strategy.
2) The (global) stopping criterion.

An overview of tree search decoding algorithms can be found in [15]. A generic tree search algorithm is described in Algorithm 3.1.

**Algorithm 3.1. (Generic Tree Search)** Given a code tree. Initialize the explored subtree to contain only the root node of the code tree. Repeat the following steps until some stopping criterion is met.

1) Select a leaf node (not a goal node) from the explored subtree.
2) Generate the children of the selected node and add them to the explored subtree.

### 3.3.1 A*-Search

There is a large body of work on the A*-search algorithm, a classical search algorithm (see for instance [3]). The A*-search algorithm uses an
evaluation function to choose the next node for expansion. The evaluation function gives an estimate of how promising a node appears. If we use the $A^*$-search algorithm to decode a sequence $y$ sent over an AWGN channel, the evaluation function should take $y$ as an argument as well as the code constraints of the linear code. Assume that $f$ is real-valued and attributes higher values to more promising nodes.

**Algorithm 3.2. ($A^*$-search)** Given a code tree and a real-valued function $f$ and a sequence $y$ received over an AWGN channel. Initialize the explored subtree to contain only the root node of the code tree. Repeat the following steps.

1) Select the leaf node with the highest $f$-value. If it is a goal node stop the search and return this goal node as the result.

2) Generate the children of the selected node and add them to the explored subtree.

The sequential decoder for convolutional codes is a prominent example of the $A^*$-search used for channel decoding. It uses the Fano metric as an evaluation function (see [8] for a modern description of the sequential decoder).

During the $A^*$-search, any node with higher $f$-value than the optimal goal node is expanded. We will discuss evaluation functions and their properties in Chapter 5. If the evaluation function fulfills some condition (discussed in Chapter 5), the $A^*$-search decoder is provably a maximum-likelihood decoder.

### 3.3.2 Forward Sweep Search

The size of the explored subtree is difficult to control when using the $A^*$-search algorithm; it depends on the $f$-values of the nodes in the tree. Finding a suitable evaluation function is difficult since it must be able to compare nodes at different depths in a code tree (which correspond to partial codewords of different lengths).

To circumvent these difficulties, we propose a different algorithm for decoding. This algorithm also uses an evaluation function $f$ but only compares nodes at the same depth. We traverse the explored subtree
from the root to the maximum depth, and at each depth choose the most promising leaf node for expansion. One such tree traversal is called a forward sweep.

**Algorithm 3.3. (Forward sweep search)** Given a code tree of maximum depth $d$, an evaluation function $f$ and a sequence $y$ received over an AWGN channel. Initialize the explored subtree to contain only the root node and its children.

Repeat until the maximum number of forward sweeps is reached or some stopping criterion is met.

1) (Forward Sweep:) For $i = 1, \ldots, d - 1$
   i) Select the leaf node at depth $i$ in the explored subtree with maximal $f$-value.
   ii) Generate the children of the selected node and add them to the explored subtree.

For a given code tree, the same number of nodes is generated in each sweep. We can therefore tightly control the size of the explored subtree with the maximum number of forward sweeps. In addition, at least one new codeword is found with each forward sweep. To take advantage of this property, we use a stopping criterion which stops the search early if a codeword which is likely to be the correct decision is found. The use of such a stopping criterion significantly reduces the average decoding complexity.

### 3.4 Multitree Search

The forward sweep algorithm fails to find the transmitted codeword in $m$ sweeps, if at least one depth there are more than $m$ nodes with higher $f$-value than the desired node (i.e., the node which belongs to the transmitted codeword). This might be the case in one tree, but may not be the case in a different tree for the same code, where the bits appear in a different order. For that reason, we propose using multiple code trees, each of which is used for an independent decoding attempt. The multitree decoder is stopped if a codeword is accepted or the maximum number of code trees has been searched.
We will show that using multiple trees is superior to using one tree (for a given maximum number of forward sweeps), even though each code tree is searched in fewer sweeps.
Chapter 4

Code Trees for Linear Codes

In Chapter 3 we introduced code trees. In this chapter, we describe how to find code trees for linear codes starting from a parity check matrix. We will present several such methods and compare the resulting code trees.

4.1 From A Parity Check Matrix To A Code Tree

We construct a code tree beginning from a (full rank) parity check matrix which is in row echelon form. A matrix is in row echelon form if:

- All nonzero rows (rows with at least one nonzero element) are above any rows of all zeros.
- The leading coefficient (the first nonzero number from the left, also called the pivot) of a nonzero row is always strictly to the right of the leading coefficient of the row above it.
• The leading coefficient of each nonzero row is 1. (For matrices over $\mathbb{Z}_2$ this is always fulfilled.)

An example matrix in row echelon form is

$$H = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}. \quad (4.1)$$

A corresponding code tree for the parity check matrix (4.1) is shown in Figure 4.1. The following definitions are needed to explain how a tree is constructed from a parity check matrix.

**Definition 4.1.** The support of a vector $x \in \mathbb{F}^n$ is

$$s(x) \triangleq \{i \in \{1, \ldots, n\} : x_i \neq 0\}.$$

**Definition 4.2.** The upper nonintersecting support of a row vector $h$ in matrix $H = (h_1^T, h_2^T, \ldots, h_{n-k}^T)^T$ is

$$S_\Delta(h_i) \triangleq s(h_i) \setminus \bigcup_{j > i} s(h_j).$$

For matrix (4.1), the upper nonintersecting support of each row is

$$(S_\Delta(h_3), S_\Delta(h_2), S_\Delta(h_1)) = (\{4, 5\}, \{3\}, \{1, 2\}).$$
4.1. From A Parity Check Matrix To A Code Tree

As can be seen in Figure 4.1, the bits in $S_{\Delta}(h_3)$ appear in the first section of the code tree, the bits in $S_{\Delta}(h_2)$ in the next section etc. A generic example tree is shown in Figure 4.2. A code tree constructed with this method has the following properties.

- **(Branching factor)** The branching factor out of a node is given by $2^{|S_{\Delta}| - 1}$ (for the corresponding $S_{\Delta}$).

- **(Constraint satisfaction)** In each section, the bits in $S_{\Delta}$ need to satisfy the corresponding constraint $h$ given the partial codeword of the preceding node. This is sufficient to ensure global constraint satisfaction. (Node expansion in the code tree is equivalent to back substitution in the parity check matrix.)
• **(Equal support at equal depth)** The partial codewords of nodes at the same depth have the same support. This is helpful when comparing nodes at the same depth.

• **(Local rate)** The partial codeword \(c_S\) at depth \(d\) in a code tree has local rate

\[
\frac{|s(c_S)| - d}{|s(c_S)|}.
\]

### 4.2 Algorithms for Constructing Code Trees for General Linear Codes

In this section, we present some algorithms for finding multiple matrices in row echelon form from a given a parity check matrix, when the parity check matrix has no known structure. As described in Section 4.1, each row echelon matrix can then be used to construct a code tree. We only describe how to find the row echelon matrices.

#### 4.2.1 General-Code Tree Algorithm

The General-Code Tree Algorithm is suitable for arbitrary linear block codes and can be used to precompute code trees which are independent of any received sequence.

**Algorithm 4.3. (General-Code Tree Algorithm)** Given a parity check matrix \(H\).

1) Find a matrix \(H_0\) by permuting the columns of \(H\) randomly.

2) Bring \(H_0\) into row echelon form with row permutations and row linear combinations.

The random permutation in Step 1) is necessary for producing different matrices when the algorithm is used repeatedly.
4.3. Code Trees for Low-Density Parity Check Codes

4.2.2 General-Code Tree Algorithm With Reliability Ordering

In this section, we generate code trees which are specifically designed to decode a given sequence $y$ received over the AWGN channel. The code trees are constructed such that high-reliability bits appear at small depth. This is achieved with a probabilistic permutation of the columns in the original matrix. The distribution from which the permutation is drawn depends on the received sequence $y$.

Algorithm 4.4. (General-Code Tree Algorithm With Reliability Ordering) Given a received sequence $(y_1, y_2, \ldots, y_n)$ from the AWGN channel and a parity check matrix $H$.

1) Obtain a matrix $H_0$ from $H$ by permuting the columns of $H$ with permutation $(s_n, \ldots, s_1)$. Draw the permutation as follows.

   i) Assign to each index $\ell \in \{1, \ldots, n\}$ a probability
      \[
      p(\ell) = \gamma e^{\lambda \theta(y_\ell)}
      \]
      where $\theta(y_\ell)$ is the reliability (see Section 2.2.1) of $y_\ell$, $\lambda$ is a free parameter and $\gamma \in \mathbb{R}$ is the scale factor required for $\sum_{i=1}^{n} p(\ell) = 1$.

   ii) Draw the permutation sequence $(s_n, \ldots, s_1)$ with these probabilities without replacement.

   iii) Permute the columns of $H$ according to $(s_n, \ldots, s_1)$.

2) Bring $H_0$ into row echelon form with row permutations and row linear combinations.

Note that if $\lambda = 0$, the permutation is drawn uniformly at random and independently of the received sequence $(y_1, \ldots, y_n)$.

4.3 Algorithms for Constructing Code Trees for Low-Density Parity Check Codes

In this section, we assume we are given a matrix $H$ which is low-density (i.e. has few nonzero entries). We present some algorithms to generate
multiple low-density matrices in row echelon form from the given matrix. The corresponding code trees have much smaller branching factors than the code trees in Section 4.2. The presented algorithms are probabilistic and avoid row superpositions if possible.

### 4.3.1 Low-Density Code Tree Algorithm

This algorithm does not use a received channel sequence and can be used to precompute code trees. It uses a free parameter \( m \).

**Algorithm 4.5. (Low-Density Code Tree Algorithm)** Given a (full rank) low-density parity check matrix \( H \) and a parameter \( m \in \{0, \ldots, n-k\} \).

1) Use permutations (described in Algorithm 4.6) to obtain a matrix \( H_0 \) from \( H \), such that the bottom \( m \) rows of \( H_0 \) are in row echelon form.

2) Bring the first \( n-k-m \) rows of \( H_0 \) into row echelon form with row permutations and row linear combinations (of the first \( n-k-m \) rows).

The algorithm is probabilistic and can fail to output a row echelon matrix. In that case, the leading coefficient of the row \( h_{n-k-m} \) in \( H_0 \) is not to the left of the leading coefficient of the row directly below. This is possible, since the upper and the lower part of \( H_0 \) are processed independently. When no row echelon matrix is found, the algorithm has to be repeated with a smaller value for \( m \). The more sparse the original parity check matrix, the larger \( m \) can be chosen.

We propose a greedy algorithm for Step 1) in Algorithm 4.5. It minimizes \( |S_\Delta| \) of every (processed) row in the resulting matrix. This limits the branching factor in the resulting code tree and at the same time maximizes the value of \( m \) for which Algorithm 4.5 is successful.

**Algorithm 4.6.** Given a low-density parity check matrix \( H \). Initialize \( H_0 = H \). The row index \( i \) indicates the position of a row in the permuted matrix such that \( H_0 = (h_1^T, \ldots, h_{n-k}^T)^T \) always holds. Process the last \( m \) rows of \( H_0 \) from the bottom up, i.e., for \( i = n-k, \ldots, n-k-m+1 \):
1.1) Swap row $h_i$ with row $h_j$, $j \in \{1, \ldots, i\}$ such that the resulting $|S_\Delta(h_i)|$ is minimal. (Break ties randomly).

1.2) If $\ell$ is the position of the leading coefficient in row $h_{i+1}$, permute columns $1, \ldots, \ell - 1$ in row $h_i$ such that the columns in $S_\Delta(h_i)$ appear immediately to the left of $\ell$.

We illustrate Algorithm 4.6 with a very small example where $m = n - k$. In the initialized matrix $H_0$,

$$
H_0 = \begin{pmatrix}
1 & 2 & 3 & 4 & 5 \\
1 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 1
\end{pmatrix}
$$

we swap rows $h_3$ and $h_1$ in Step 1.1), since for the bottom row $|S_\Delta(h)| = |s(h)|$ and $|s(h_1)|$ is minimal. Rearranging the rows and columns in Step 1.2) yields

$$
H_1 = \begin{pmatrix}
2 & 4 & 5 & 1 & 3 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 1
\end{pmatrix}
$$

When we process the middle row in Step 1.1), we can swap $h_2$ with either $h_2$ or $h_3$ since $|S_\Delta(h_2)| = |S_\Delta(h_3)| = 2$. We arbitrarily leave $h_2$ at its current position. After Step 1.2), the resulting matrix $H$ has row echelon form.

$$
H = \begin{pmatrix}
5 & 4 & 2 & 1 & 3 \\
1 & 0 & 1 & 0 & 1 \\
0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 1
\end{pmatrix}
$$

4.3.2 Low-Density Code Tree Algorithm With Reliability Ordering

In this section, we again generate code trees which are specifically designed for decoding a given sequence $y$ received from an AWGN channel.
For this purpose, we use a variation of Algorithm 4.5. The proposed algorithm produces low-density parity check matrices in row echelon form such that high-reliability bits appear close to the root in the resulting code trees.

**Algorithm 4.7. (Low-Density Code Tree Algorithm with Reliability Ordering)** Given a low-density parity check matrix $H$, a free parameter $m \in \{0, \ldots, n-k\}$ and a channel sequence $(y_1, \ldots, y_n)$ received over an AWGN channel.

1) Use permutations (described in Algorithm 4.8) to obtain a matrix $H_0$ from $H$, such that the bottom $m$ rows of $H_0$ are in row echelon form.

2) Bring the first $n-k-m$ rows of $H_0$ into row echelon form with row permutations and row linear combinations (of the first $n-k-m$ rows).

This algorithm differs only in the use of Algorithm 4.8 in Step 1) from Algorithm 4.5. Algorithm 4.8 is similar to Algorithm 4.6 in that is minimizes $|S_\Delta|$ of the processed rows with a greedy method. It differs from Algorithm 4.6 in its tie-breaking strategy in Step 1); it compares the maximum-log-likelihood of the parity checks (see Section 2.2.1) when permuting rows.

**Algorithm 4.8.** Given a low-density parity check matrix $H$. Initialize $H_0 = H$. We process the last $m$ rows of $H_0$ from the bottom up, i.e. for $i = n-k, \ldots, n-k-m+1$:

1.1) Swap row $h_i$ with row $h_j, j \in \{1, \ldots, i\}$ such that the resulting $|S_\Delta(h_i)|$ is minimal. If multiple rows qualify, identify the two rows with highest ML($h$). Choose either with probability $1/2$.

1.2) If $\ell$ is the position of the leading coefficient in row $h_{i+1}$, permute columns $1, \ldots, \ell-1$ such that the columns in $S_\Delta(h_i)$ appear immediately to the left of $\ell$.

We use a random decision in 1.1) of Algorithm 4.8 to ensure sufficient diversity among matrices when the algorithm is used repeatedly.
4.3.3 Selection of Precomputed Low-Density Code Trees

Instead of generating code trees for a specific sequence \( y \), we can generate a large number of (randomly ordered) code trees. We then use \( y \) to select those code trees from the precomputed set that have a favorable bit ordering. This saves the cost of generating code trees at the decoding stage. We propose a tree selection method.

**Algorithm 4.9. (Low-Density Code Tree Selection)** Given \( d \) low-density code trees (computed with Algorithm 4.5) and a received sequence \( y \) sent over an AWGN channel. Choose \( t \) trees \( (t < d) \) from the list of trees.

1) For every tree \( T \), compute

\[
 f(T) = \sum_{i=1}^{\ell} \theta(y_{s_i}),
\]

where \( s_1, s_2, \ldots \) is the order in which the bits appear in the corresponding tree, \( \theta \) is the bit reliability and \( \ell \in \{1, \ldots, n\} \) is a free parameter.

2) Select the \( t \) trees with highest \( f(T) \).

4.4 Simulated Average Code Tree Properties

In this section we use simulations to compare code trees generated with the algorithms presented in the previous sections. In particular, we compare the branching factors (and therefore local rates) in general-code trees and low-density code trees. Note that reliability ordering has no influence on the code tree structure for either type of tree. The following codes were used in our simulations.

1) A \((3, 6)\)-regular LDPC code of length \( n = 100 \), where the cycles of length 4 have been removed.
2) A (5, 10)-regular LDPC code of length $n = 100$.

3) A random $(p = 1/2)$-code of length $n = 100$ and rate $r = 1/2$.

### 4.4.1 Simulated Structure of General-Code Trees

We use the General-Code Tree Algorithm 4.3 on all codes presented and obtain the following average results.

(Random $(p = 1/2)$-code) Simulated average code tree parameters for a random $(p = 1/2)$-code of rate $1/2$ and $n = 100$ when using Algorithm 4.3:

- **(Branching factor)** The number of children (beginning from the root node) at each depth are approximately
  \[
  (2^{24}, 2^{13}, 2^6, 2^3, \ldots).
  \]

- **(Depth of final information bit)** Less than $1/1000$ trees contain an information bit at depth $d > 16$ in our simulations.

The average code trees for the low-density (5, 10)-regular code are almost identical to the code trees of the random $(p = 1/2)$-code; their parameters are therefore not listed. The low-density (3, 6)-regular code yields the following parameters.

((3, 6)-regular code) Simulated average code tree parameters for a (3, 6)-regular code of rate $1/2$ and $n = 100$ when using Algorithm 4.3:

- **(Branching factor)** The number of children (beginning from the root node) at each depth are approximately
  \[
  (2^{18}, 2^{11}, 2^8, 2^5, \ldots).
  \]

- **(Depth of final information bit)** Less than $1/1000$ trees contain an information bit at depth $d > 26$ in our simulations.

Figure 4.3 show the simulated average $|S_\Delta|$ for general-code trees for different codes. Note that the (5, 10)-regular code is included in the figure.
4.4. Simulated Average Code Tree Properties

Figure 4.3: Average $|S_\Delta|$ for general-code trees computed with Algorithm 4.3 for different codes of length $n = 100$ and rate 1/2. From top to bottom: A $(p = 1/2)$-random code (solid line), a $(5, 10)$-regular code (solid line) and a $(3, 6)$-regular code (dashed line). (The graphs for the $(p = 1/2)$-code and the $(5, 10)$-regular code cannot be distinguished in the plot.)

4.4.2 Simulated Structure of Low-Density Code Trees

We use the Low-Density Code Tree Algorithm 4.5 on the $(3, 6)$-regular and the $(5, 10)$-regular codes. The value of the free parameter $m$ was set experimentally. It is not possible to use the algorithm on a $(p = 1/2)$-code (except with parameter $m = 0$, in which case the algorithm is equivalent to the General-Code Tree Algorithm 4.3).
(5, 10)-regular code  Simulated average code tree parameters for a
(5, 10)-regular code of rate 1/2 and \( n = 100 \) when using Algorithm 4.5
with parameter \( m = 15 \):

- **(Branching factor)** The number of children (beginning from the
root node) at each depth are approximately

\[(2^9, 2^6, 2^5, 2^5, 2^4, \ldots).\]

- **(Depth of final information bit)** Less than 1/1000 trees contain
an information bit at depth \( d > 26 \) in our simulations.

For the (3, 6)-regular code, the simulated parameters are as follows.

((3, 6)-regular code)  Simulated average code tree parameters for
a (3, 6) code of rate 1/2 and \( n = 100 \) when using Algorithm 4.5 with
parameter \( m = 27 \):

- **(Branching factor)** The number of children (beginning from the
root node) at each depth are approximately

\[(2^5, 2^4, 2^3, 2^3, \ldots).\]

- **(Depth of final information bit)** Less than 1/1000 trees contain
an information bit at depth \( d > 38 \) in our simulations.

Figure 4.4 shows the simulated average \(|S_\Delta|\) for low-density code trees
for different codes. Note that the average \(|S_\Delta|\) are significantly smaller
compared to the general-code trees for the same code.

### 4.4.3 Comparison

As expected, low-density code trees have much smaller branching fac-
tors than general-code trees. Nevertheless, we wish to emphasize a few
observations.

- The general-code trees for the \((p = 1/2)\)-and the (5, 10)-regular
code trees are almost identical.
Figure 4.4: Average $|S_{\Delta}|$ for low-density code trees computed with Algorithm 4.5 for different codes of length $n = 100$ and rate $1/2$. A $(5, 10)$-regular code (simulated with $m = 15$, dashed line) and a $(3, 6)$-regular code (simulated with $m = 27$, solid line).

- The general-code trees of the $(3, 6)$-regular code have smaller branching factors than the general-code trees for the $(p = 1/2)$-code.

- The low-density code trees have significantly smaller branching factors than the general-code trees for the same code family.

This comparison so far only considers codes of length $n = 100$. If we use longer codes, the sequence of branching factors remains the same for low-density code trees. For example, for a $(3, 6)$-regular code, the sequence of branching factors begins with $(2^5, 2^4, 2^3, 2^3, \ldots)$, irrespective of length. General-code trees do not share this property. For a $(3, 6)$-regular code tree for a code of length $n = 200$, the sequence of average branching factors begins with $(2^{23}, 2^{19}, 2^{16}, 2^{13}, \ldots)$. At length $n = 100$, the corresponding sequence is $(2^{18}, 2^{11}, 2^8, 2^5, \ldots)$. 
Chapter 5

Evaluation Functions

In this chapter we briefly review some properties of evaluation functions and propose two evaluation functions which are optimized for forward sweep decoding.

5.1 Introduction

We are given a linear code $C$ and a sequence $y$ transmitted over an AWGN channel. We wish to find an evaluation function to decode the sequence with a tree search over a code tree. Specifically, we wish to find the codeword

$$\hat{x} = \arg\max_{x \in C} \log p(x|y),$$

which is the maximum-a-posteriori (MAP) decision (Definition 2.4). The MAP decoder coincides with the maximum-likelihood (ML) decoder (Definition 2.5), if the prior distribution on the codewords is uniform. We assume this to be the case in this thesis. We also assume that every bit has value zero or one with equal probability. For every bit $x_i$ and received symbol $y_i$, we can compute

$$p(x_i|y_i) = \frac{p(x_i)p(y_i|x_i)}{p(y_i)} \propto p(y_i|x_i).$$
In the tree search context, we wish to expand nodes which are likely to lead to the correct MAP decision. If we could evaluate

\[ g^*(y, c_S, C) = \max_{x \in C : x_S = c_S} \log p(x|y), \]  

(5.1)

for every node (\(c_S\) is the partial codeword of the considered node), we would always expand the correct node and arrive at the MAP decision.

Computing \(g^*\) is computationally expensive, therefore we are looking for efficient approximations of \(g^*\). In the following, we will present two evaluation functions which approximate \(g^*\) and satisfy

\[ g(y, c_S, C) \geq g^*(y, c_S, C), \]

i.e., are upper bounds on \(g^*\). Note that \(A^*\)-search is provably an MAP decoder for evaluation functions with this property. A proof can be found in [3].

### 5.2 The Basic Evaluation Function

The basic evaluation function is

\[ g(y, c_S) = \log p(c_S|y_S), \]  

(5.2)

where \(y\) is the sequence received over the AWGN channel, \(c_S\) is the partial codeword of the node considered. This function is inexpensive to compute and well suited for the Forward Sweep Search Algorithm 3.3 used with code trees constructed as described in Chapter 4.

It is easy to see that (5.2) it is an upper bound on \(g^*\). Let \(x^*\) be the argument of \(g^*\) at this node, i.e., \(x^* = \arg \max_{x \in C : x_S = c_S} \log p(x|y)\). (\(C\) is the given code). Then, clearly,

\[ g(y, c_S) = \log p(c_S|y_S) \geq \log p(x^*|y) = g^*(y, c_S, C), \]  

(5.3)

since

\[ \log p(x^*|y) = \log p(c_S|y_S) + \sum_{i \notin S} \log p(x_i^*|y_i). \]

\[ \leq 0 \]
5.3 The Greedy Evaluation Function

In this section we present a more elaborate evaluation function which is a tighter upper bound on $g^*$ than the basic evaluation function (5.2). For a given code $C$, a node with partial codeword $c_S$ and a sequence $y$ received from an AWGN channel, we propose

$$g(y, c_S, C) = \max_{x \in C': x_S = c_S} \log p(x|y), \quad (5.4)$$

where $C'$ is a linear code such that $C \subset C'$. It is clear that the evaluation function (5.4) is an upper bound on $g^*$, since

$$\max_{x' \in C': x'_S = c_S} \log p(x'|y) \geq \max_{x \in C: x_S = c_S} \log p(x|y) \quad (5.5)$$

follows from $C|_{x_S = c_S} \subset C'|_{x_S = c_S}$. The computational cost and bounding error depend on the choice of $C'$. (A similar evaluation function for sequential decoding of convolutional codes is proposed in [7].)

We define $C'$ via its parity check matrix $H'$. We choose a subset of rows from the original parity check matrix $H$ to build $H'$. Since we wish the evaluation function to approximate $g^*$ as closely as possible, we propose a greedy method for minimizing the approximation error (therefore the name). We choose $H'$ such that the following conditions are fulfilled for all $h'$ in $H'$:

1) $h'x_T H|_{x_S = c_S} \neq 0$.

2) $(s(h'_i) \setminus S) \cap (s(h'_j) \setminus S) = \emptyset$ for $i \neq j$.

3) $s(h') \cap S \neq \emptyset$ for all $h'$ in $H'$.

Condition 1) serves to minimize the approximation error. At least one bit in the support of each row $h'$ has to be erroneous. If condition 2) is met, then the the parity check constraints $h'$ have pairwise disjoint support sets (given the partial codeword $c_S$) and every parity check equation can be maximized separately in (5.4). This makes the maximization step inexpensive to compute. Condition 3) ensures that the parity checks in $H'$ intersect with $S$ and the value of the evaluation function depends on the partial codeword $c_S$. Algorithm 5.1 shows how to construct a matrix $H'$ from a given matrix $H$ which satisfies all three conditions.
and minimizes the approximation error in a greedy way. It has to be noted that this evaluation function works best with sparse parity check matrices.

**Algorithm 5.1. (Greedy Choice of $H'$)** Given a code $C$ with parity check matrix $H$ and partial codeword $c_S$. Initialize an empty index set $I$.

1) Initialize an index set $U$ which contains the row indices of the rows $h$ in $H$ which are unsatisfied given the partial codeword $c_S$ and whose support intersects with $S$.

2) Repeatedly select a row from $U$ according to the following criteria and add its index to $I$. Stop if no row qualifies.

   i) Choose from the rows $h$ in $U$ which satisfy $(s(h) \setminus S) \cap (s(h_i) \setminus S) = \emptyset$ for all $i \in I$ the row whose least reliable bit in $s(h) \setminus S$ has highest reliability. If multiple rows qualify, break ties in the following order.

      1. Choose the row with smallest $|s(h) \setminus S|$.
      2. Choose at random.

3) Use the rows in $I$ to construct the matrix $H'$. 
Chapter 6

Multitree Search
Decoding with the Basic Evaluation Function

In this section we present some simulation results for multitree search decoders. We use the Basic Evaluation Function (5.2) throughout this chapter.

6.1 Simulation Procedure

The multitree search decoder operates as follows.

Algorithm 6.1. (Multitree search) Given a code $C$, a received sequence $y$, a maximum number of code trees and a maximum number of forward sweeps per code tree. An acceptance criterion is also given.

Repeat until the maximum number of trees is reached.

1) Generate a code tree.

2) Search the tree with forward sweep search. After each sweep, verify if a codeword is acceptable. If it is, exit and return the
codeword. Otherwise continue until the maximum number of forward sweeps is reached.

Algorithm 6.1 is used throughout Chapters 6 and 7. The parameters varied are the maximum number of code trees and forward sweeps as well as the code tree generation method and the evaluation function used.

- **(Acceptance criterion)** We accept codeword $x$ if
  \[
  \log p(x|y) \geq a
  \]
  ($a$ is the acceptance threshold). We choose $a$ experimentally. The goal is to stop the search early if a codeword is found which is very likely to be correct. If the same simulations were done without an acceptance threshold, the simulated word error rates should be almost the same.

- **(Complexity)** The worst case computational complexity is determined by the maximum number of trees and the maximum number of forward sweeps per tree. The average case computational complexity is smaller due to the acceptance threshold. How often a search is stopped due to the acceptance threshold strongly depends on the signal-to-noise ratio. Note that most of the computational cost is used in those cases where the decoder fails to find the correct codeword. If two different decoders operate with the same maximum number of code trees and forward sweeps and achieve different word error rates, then the decoder with lower error rate uses fewer computations.

- **(ML lower bound)** We also simulate a lower bound on the word error rate of the ML decoder during tree search. The ML lower bound is obtained as follows. Whenever the tree search decoder finds a codeword $x'$ which satisfies
  \[
  \log p(x'|y) \geq \log p(x^*|y)
  \]
  (and $x^*$ is the transmitted codeword), a word error is counted for the ML decoder.

As a further point of reference, word error rates of a sum-product decoder (see Appendix A) are shown for LDPC codes.
6.2 Simulated Word Error Rates for a $(3, 6)$-regular LDPC Code of Length $n = 100$

We use a $(3, 6)$-regular LDPC code of length $n = 100$. The code was randomly generated and is free of cycles of length 4. Each multitree search decoder performs at most 50 forward sweeps in each of at most 5 trees. The multitree decoders differ only in the method used to construct the code trees.

Figure 6.1 compares simulated word error rates for multitree decoders which use general-code trees (with and without reliability ordering). Figure 6.2 compares simulated word error rates for multitree decoders which use low-density code trees (with and without reliability ordering).

Surprisingly, general-code trees with reliability ordering outperform low-density code trees with reliability ordering. The opposite is true when no reliability ordering is used. For low-density code trees, choosing from precomputed trees offers no better reliability ordering than generating code trees with knowledge of the received sequence.

6.3 Simulated Word Error Rates for a $(p = 1/2)$-Code of Length $n = 100$

In this section, we study the effect of the maximum number of forward sweeps on the word error rate. We also vary the maximum number of code trees in our simulations. We use a random $(p = 1/2)$-code of length $n = 100$ and rate $r = 1/2$. Since only general-code trees can be used for this code and the reliability ordering greatly improves the performance of the multitree search decoder (see Figure 6.1), only general-code trees with reliability ordering (Algorithm 4.4) are used.

Figure 6.3 shows word error rates for multitree search decoders with different simulation limits. It shows clearly that the word error rate decreases with higher maximum number of forward sweeps. It is worthwhile to use multiple trees, since searching 10 trees offers better performance than searching 1 tree when at most 1000 forward sweeps are allowed. Note that the worst case complexity of these two decoders is comparable.
Figure 6.1: Simulated word error rates for a (3, 6)-regular code of length $n = 100$ using multitree search decoding with the basic evaluation function (5.2). In each decoding attempt, at most 5 different trees were searched in at most 50 forward sweeps per tree. From top to bottom: Multitree search with general-code trees (Algorithm 4.3), SP decoding with 50 iterations (dashed line), multitree search with general-code trees with reliability ordering (Algorithm 4.4 with $\lambda = 1$), ML lower bound (dashed line).

(if we neglect the cost of generating code trees), but the average complexity of the multitree search decoder is smaller than the complexity of the single tree decoder. This can be seen from the error rate. If the decoder fails to find the correct codeword, it generally only stops when the computational limits are reached (as opposed to when a codeword is accepted). A higher error rate therefore indicates higher computational cost.
6.4 Comparison of Simulated Word Error Rates for Different Codes of Length $n = 100$

In this section, we compare the word error rates for different codes of length $n = 100$. We use the $(3, 6)$-regular code from Section 6.2, the
Figure 6.3: Simulated word error rates for a \((p = 1/2)\)-code of length \(n = 100\) and rate \(r = 1/2\) using multitre search decoding with the basic evaluation function \((5.2)\). Only general-code trees with reliability ordering were used (Algorithm 4.4 with \(\lambda = 1\)). From top to bottom: 1 tree and at most 200 forward sweeps, at most 5 different trees and at most 50 forward sweeps per tree, 1 tree and at most 1000 forward sweeps, at most 10 different trees and at most 100 forward sweeps per tree, at most 10 different trees and at most 200 forward sweeps per tree, ML lower bound (dashed line).

\((p = 1/2)\)-code from Section 6.3 and a random \((5, 10)\)-regular code. The decoders use general-code trees with reliability ordering (Algorithm 4.4).

Figure 6.4 shows the word error rates for all 3 codes when decoded with at most 10 different code trees and at most 200 forward sweeps per code tree. The gap to the ML lower bound increases slightly with code density, but all three word error rates are near-optimal. Figure 6.4 shows the word error rates for all 3 codes when decoded with at most 5 different code trees and at most 50 forward sweeps per code tree. As expected, the word error rates are higher, but the higher-density codes
still outperform the low-density code.

![Graph showing Simulated Word Error Rates](image)

**Figure 6.4:** Simulated word error rates for different codes of length $n = 100$ and rate $r = 1/2$ using multtree search decoding with the basic evaluation function (5.2). Only general-code trees with reliability ordering were used (Algorithm 4.4 with $\lambda = 1$). At most 10 code trees were searched in at most 200 forward sweeps per tree. From top to bottom: (3, 6)-regular code, (5, 10)-regular code, ($p = 1/2$)-code. An ML lower bound is shown for each code (dashed lines).

### 6.5 Simulated Word Error Rates for a (3, 6)-regular LDPC Code of Length $n = 200$

We simulated the word error rates for a random (3, 6)-regular code of length $n = 200$. The code contains no cycles of length 4. We compare the performance of a multtree search decoder which uses general-code trees with reliability ordering with the performance of a multtree search decoder which uses low-density code trees with reliability ordering. As
Figure 6.5: Simulated word error rates for different codes of length \( n = 100 \) and rate \( r = 1/2 \) using multitree search decoding with the basic evaluation function (5.2). Only general-code trees with reliability ordering were used (Algorithm 4.4 with \( \lambda = 1 \)). At most 5 code trees were searched in at most 50 forward sweeps per tree. From top to bottom: \((3,6)\)-regular code, \((5,10)\)-regular code, \((p = 1/2)\)-code. An ML lower bound is shown for each code (dashed lines).

mentioned in Section 4.4.3, the branching factors of general-code trees scale with length while those of low-density code trees do not.

Figure 6.6 shows the simulated word error rates. Surprisingly, the general-code trees with reliability ordering outperform the low-density code trees with reliability ordering; this is consistent with observations for length \( n = 100 \).
Figure 6.6: Simulated word error rates for a $(3, 6)$-regular code of length $n = 200$ using multitree search decoding with the basic evaluation function (5.2). At most 25 code trees were searched in at most 200 forward sweeps per tree. From top to bottom: SP decoding with 50 iterations (dashed line), multitree search with low-density code trees with reliability ordering (Algorithm 4.7), multitree search with general-code trees with reliability ordering (Algorithm 4.4 with $\lambda = 1$), ML lower bound (dashed line).
Chapter 7

Multitree Search
Decoding of LDPC
Codes with The Greedy Evaluation Function

In this section we present some simulation results for multitree search decoders with the Greedy Evaluation Function (5.4). Since this evaluation function only works well for LDPC codes, only LDPC codes are used. The simulation procedure is described in Section 6.1.

7.1 Simulated Word Error Rates for a (3, 6)-regular LDPC Code of Length $n = 100$

We use the same (3, 6)-regular LDPC code as in Chapter 6. Figure 7.1 compares simulated word error rates for multitree search decoders which use general-code trees. The results from Chapter 6 are included to allow direct comparison. When used with the same type of general-code trees, the Greedy Evaluation Function (5.4) clearly outperforms the Basic
Evaluation Function (5.2). Nevertheless, general-code trees with reliability ordering outperform general-code trees without reliability ordering, regardless of the evaluation function used.

Figure 7.2 compares simulated word error rates for multitree search decoders which use low-density code trees. The results from Chapter 6 are included to allow direct comparison. Interestingly, decoders which use the Greedy Evaluation Function (5.4) achieve comparable word error rates, irrespective of the type of code trees used.

7.2 Simulated Word Error Rates for a (3, 6)-regular LDPC Code of Length $n = 200$

We use the same (3, 6)-regular LDPC code of length $n = 200$ as in Chapter 6. We include the simulated word error rates from Chapter 6 for comparison. The word error rates achieved with either general-code trees or low-density code trees are comparable when the Greedy Evaluation Function (5.4) is used.
Figure 7.1: Simulated word error rates for a $(3, 6)$-regular code of length $n = 100$ using multitype search decoding. In each decoding attempt, at most 5 different code trees were searched in at most 50 forward sweeps. From top to bottom: Multitype search with general-code trees (Algorithm 4.3) and the Basic Evaluation Function (5.2), SP decoding with 50 iterations (dashed line), multitype search with general-code trees (Algorithm 4.3) and the Greedy Evaluation Function (5.4), multitype search with general-code trees with reliability ordering (Algorithm 4.4 with $\lambda = 1$) and the Basic Evaluation Function (5.2), multitype search with general-code trees with reliability ordering (Algorithm 4.4 with $\lambda = 1$) and the Greedy Evaluation Function (5.4), ML lower bound (dashed line).
Figure 7.2: Simulated word error rates for a (3, 6)-regular code of length \( n = 100 \) using multitree search decoding. In each decoding attempt, at most 5 different code trees were searched in at most 50 forward sweeps per tree. From top to bottom: SP decoding with 50 iterations (dashed line), multitree search with low-density code trees (Algorithm 4.5) and the Basic Evaluation Function (5.2), multitree search with low-density code trees with reliability ordering (Algorithm 4.7) and the Basic Evaluation Function (5.2), multitree search with precomputed low-density code trees selected with Algorithm 4.9 (500 code trees were precomputed and the first 40 bits compared at point of selection) and the Basic Evaluation Function (5.2), multitree search with low-density code trees (Algorithm 4.5) and the Greedy Evaluation Function (5.4), multitree search with low-density code trees with reliability ordering (Algorithm 4.7) and the Greedy Evaluation Function (5.4), multitree search with precomputed low-density code trees selected with Algorithm 4.9 (500 code trees were precomputed and the first 40 bits compared at point of selection) and the Greedy Evaluation Function (5.4), ML lower bound (dashed line).
Figure 7.3: Simulated word error rates for a (3, 6)-regular code of length \( n = 200 \) using multtree search decoding. In each decoding attempt, at most 25 different code trees were searched in at most 200 forward sweeps per tree. From top to bottom: SP decoding with 50 iterations (dashed line), multtree search with low-density code trees with reliability ordering (Algorithm 4.7) and the Basic Evaluation Function (5.2), multtree search with general-code trees with reliability ordering (Algorithm 4.4 with \( \lambda = 1 \)) and the Basic Evaluation Function (5.2), multtree search with general-code trees with reliability ordering (Algorithm 4.4 with \( \lambda = 1 \)) and the Greedy Evaluation Function (5.4), multtree search with low-density code trees with reliability ordering (Algorithm 4.7) and the Greedy Evaluation Function (5.4), multtree search with precomputed low-density code trees selected with Algorithm 4.9 (500 code trees were precomputed and the first 80 bits compared at point of selection) and the Greedy Evaluation Function (5.4), ML lower bound (dashed line).
Chapter 8

Multitree Search Aided LDPC Decoding

In this section, we show how multitree search decoding can be used to enhance the standard sum-product (SP) decoder (for a brief review of SP decoding, see Appendix A). When the sum-product decoder fails to find a codeword, a tree search is used to decode a small subset of bits. The channel observations for these bits are then replaced with the tree search output decisions and the sum-product decoder is restarted. In [16] and [17], similar two-stage decoders are proposed. The proposed decoders identify some bits for manipulation and treat them as information bits (i.e., enumerate over all possible configurations).

8.1 Introduction

Tree search decoding can achieve near-optimal decoding performance for small codes (see Chapters 6 and 7). However, the computational complexity scales unfavourably with codebook size (see for instance [3] for an analysis of the A*-search algorithm. We have no analysis of forward sweep search but the two are closely related).

The sum-product decoder is the standard decoder for LDPC codes
and is particularly successful for long codes (see for instance [14]). However, there is no guarantee that the sum-product decoder output is a codeword. Tree search, on the other hand, searches only over valid codewords or partial codewords. We therefore propose to use tree search on a subset of bits to force their values to a partial codeword. In particular, we aim to identify subsets of bits which cause the SP decoder to fail. With such a two-level decoder better performance can be achieved than with either decoder alone.

An outline of the proposed decoder is given in Algorithm 8.1.

Algorithm 8.1. (Generic Tree Search Aided SP Decoder) Given a linear code $C$ of length $n$ and a sequence $y$ received from an AWGN channel for which the SP decoder output is not a codeword.

1) Choose some subset $S \subset \{1, \ldots, n\}$.

2) Use a tree search to decode $x_S$.

3) Replace the channel messages $p(y_i|x_i)$ for $i \in S$ with $\{0, 1\}$-values found with tree search. Restart the SP decoder with these input messages.

8.2 Simulated Tests With an Error-Free Tree Search Decoder

In this section we study how the choice of $S$ in Step 1) of Algorithm 8.1 influences the performance of the additional SP decoding attempts. For this purpose, we use we use knowledge of the transmitted codeword to identify the erroneous received bits and force them to their correct value (therefore the name “error-free” decoder, there is no decoding step). We only manipulate one bit at a time and restart the SP decoder with the modified input. We repeat this experiment for different bits to test the relation between decoder performance and the reliability of the manipulated bit. We choose both the most reliable and the least reliable bits. Algorithm 8.2 explains the procedure for manipulating high-reliability bits. When manipulating low-reliability bits, the ordering $(\pi_1, \pi_2, \ldots)$ is reversed.
Algorithm 8.2. (Single Bit Flipping Test) Given a linear code $C$ and a sequence $y$ received from an AWGN channel (the SP decoder output for $y$ is not a codeword). Given the transmitted codeword $c \in C$ and a maximum number of trials. Order the flipped bits such that in the resulting ordering $(\pi_1, \pi_2, \ldots)$, the most reliable erroneous bit is $x_{\pi_1}$, the second most reliable erroneous bit is $x_{\pi_2}$, etc. Repeat the following steps, until the maximum number of trials is reached or the SP decoder output is a codeword. In this case, return the codeword.

1) Choose bit $x_{\pi_i}$ for the $i$-th trial.

2) Replace the channel message for bit $x_{\pi_i}$ with its value in the transmitted codeword. Restart the SP decoder with the channel messages and the manipulated bit as input messages. (Reset all other messages).

We also test the effect of fixing the value of correct bits, i.e., increasing their reliability. We correct multiple bits at a time.

Algorithm 8.3. (Correct Decision Enforcement Test) Given a linear code $C$ and a sequence $y$ received from an AWGN channel (the SP decoder output for $y$ is not a codeword). Given the transmitted codeword $c \in C$ and a maximum number of bits $|S|$.

1) Select the $|S|$ most reliable correct bits.

2) Replace the channel messages for the selected bits with their value in the transmitted codeword. Restart the SP decoder with the channel messages and the manipulated bits as input messages. (Reset all other messages).

We use two different LDPC codes for our simulations; a code of length $n = 504$ and rate $r = 1/2$ and a code of length $n = 1008$ and rate $r = 1/2$. Both codes were constructed with progressive edge growth and were downloaded from [18].

Figures 8.1 and 8.2 show the simulated word error rates. They clearly show that correcting high-reliability flips is the most successful strategy. Manipulating even a large number of correct bits has only a negligible effect on performance.
Figure 8.1: Simulated word error rates for an LDPC code of length \( n = 1008 \). From top to bottom: SP decoding (50 iterations), correct bit enforcement test (Algorithm 8.3 with \(|S| = 200\)), single bit flipping test (Algorithm 8.2 with 1 least reliable bit), single bit flipping test (Algorithm 8.2 with at most 10 least reliable bits), single bit flipping test (Algorithm 8.2 with 1 most reliable bit), single bit flipping test (Algorithm 8.2 with at most 10 most reliable bits). (Note that in this plot the WER for the correct bit enforcement test and the single bit flipping test with 1 least reliable bit cannot be distinguished).

Another question is whether the SP decoder output marginals can be used to localize suitable positions for bit flipping. The SP decoder computes an approximate marginal \( \tilde{p}(x_\ell|y) \) (called the “belief”) for every bit in the codeword (see Appendix A). Each marginal is normalized to satisfy \( \sum_{x_\ell} p(x_\ell|y) = 1 \). We again use knowledge of the transmitted codeword. Both most “confident” (i.e., high \( \max_{x_i} \log \tilde{p}(x_i|y) \)) and least confident positions are selected.

Algorithm 8.4. (Single Belief-Based Bit Flipping Test) Given a
Figure 8.2: Simulated word error rates for an LDPC code of length $n = 504$. From top to bottom: SP decoding (50 iterations), correct bit enforcement test (Algorithm 8.3 with $|S| = 100$), single bit flipping test (Algorithm 8.2 with 1 least reliable bit), single bit flipping test (Algorithm 8.2 with at most 10 least reliable bits), single bit flipping test (Algorithm 8.2 with 1 most reliable bit), single bit flipping test (Algorithm 8.2 with at most 10 most reliable bits).

linear code $C$ and a sequence $y$ received from an AWGN channel. The SP decoder output for $y$ is not a codeword and the SP decoder output marginals $\tilde{p}(x_\ell | y)$ are given for all bits. Given the transmitted codeword $c \in C$ and a maximum number of trials. Order the bits with erroneous SP output decision such that in the resulting ordering $(\pi_1, \pi_2, \ldots)$ satisfies $\max_{x_{\pi_1}} \log \tilde{p}(x_{\pi_1} | y) > \max_{x_{\pi_2}} \log \tilde{p}(x_{\pi_2} | y) > \max_{x_{\pi_3}} \log \tilde{p}(x_{\pi_3} | y) > \ldots$. Repeat the following steps, until the maximum number of trials is reached or the SP decoder output is a codeword. In this case, return the codeword.

1) Choose bit $x_{\pi_i}$ for the $i$-th trial.

2) Replace the channel message for the selected bit with its value in
the transmitted codeword. Restart the SP decoder with the channel messages and the manipulated bit as input messages. (Reset all other messages).

Analogously to Correct Decision Enforcement Test Algorithm 8.3, we propose an algorithm which localizes candidate bits using the SP decoder output marginals.

Algorithm 8.5. (Correct Belief-Based Decision Enforcement Test)

Given a linear code $C$ and a sequence $y$ received from an AWGN channel. The SP decoder output for $y$ is not a codeword and the SP decoder output marginals $\tilde{p}(x_\ell|y)$ are given for all bits. Given the transmitted codeword $c \in C$ and a maximum number of bits $|S|$.

1) Select the $|S|$ bits with correct SP decoder output decision with highest $\max_{x_\ell} \log \tilde{p}(x_\ell|y)$.

2) Replace the channel messages for the selected bits with their value in the transmitted codeword. Restart the SP decoder with the channel messages and the manipulated bits as input messages. (Reset all other messages).

Figures 8.4 and 8.3 show some simulation results for the belief-based tests. The belief-based selection algorithms’ error rates are comparable to the channel message based selection algorithms’ error rates, with the belief-based algorithms doing slightly better. For either bit selection method, the most successful strategy is to find flipped bits with high reliability (or high belief confidence).

8.3 Single Bit Flipping Aided LDPC Decoding Algorithms

In this section present a special case of the Generic Tree Search Aided SP Decoder (Algorithm 8.1), namely the case where only one bit is forced to a value in each SP decoding attempt (therefore the name single bit flipping, no tree search is necessary). Using the conclusions from the previous section, we aim to find high reliability erroneous bits and and force them to the correct decision.
8.3. Single Bit Flipping Aided LDPC Decoding Algorithms

Figure 8.3: Simulated word error rates for an LDPC code of length $n = 1008$. $b_\ell = \max_{x_\ell} \log \tilde{p}(x_\ell|y)$ was used for selecting bits, where $\tilde{p}(x_\ell|y)$ is the bit’s SP output marginal and $y$ is the received channel sequence. From top to bottom: SP decoding (50 iterations), correct belief-based bit enforcement test (Algorithm 8.5 with $|S| = 200$), single belief-based bit flipping test (Algorithm 8.4 with 1 bit with smallest $b_\ell$), single belief-based bit flipping test (Algorithm 8.4 with at most 10 bits with smallest $b_\ell$), single belief-based bit flipping test (Algorithm 8.4 with 1 bit with highest $b_\ell$), single belief-based bit flipping test (Algorithm 8.4 with at most 10 bits with highest $b_\ell$).

Algorithm 8.6. (Single Bit Flipping) Given a linear code $C$ (and its parity check matrix $H$) and a sequence $y$ received from an AWGN channel. The SP decoder output for $y$ is not a codeword. Rank the unsatisfied parity checks $h \in H$ according to their maximum log-likelihood $ML(h)$ (see Section 2.2.1). Go through the list of unsatisfied parity check equations, beginning with the parity check with highest $ML(h)$. For each check $h$, select the two least reliable bits in the support of $h$. Use each
Figure 8.4: Simulated word error rates for an LDPC code of length $n = 504$. $b_\ell = \max_{x_\ell} \log \tilde{p}(x_\ell|y)$ was used for selecting bits, where $\tilde{p}(x_\ell|y)$ is the bit’s SP output marginal and $y$ is the received channel sequence. From top to bottom: SP decoding (50 iterations), correct belief-based bit enforcement test (Algorithm 8.5 with $|S| = 100$), single belief-based bit flipping test (Algorithm 8.4 with 1 bit with smallest $b_\ell$), single belief-based bit flipping test (Algorithm 8.4 with at most 10 bits with smallest $b_\ell$), single belief-based bit flipping test (Algorithm 8.4 with 1 bit with highest $b_\ell$), single belief-based bit flipping test (Algorithm 8.4 with at most 10 bits with highest $b_\ell$).

selected bit for an SP decoding attempt as follows. (Stop if the maximum number of trials is reached).

1) **Fix the decision for bit $i$ as** $\hat{x}_i = \arg\min_{x_i} \log p(y_i|x_i)$.

2) **Replace the channel message for the selected bit with the decision** $\hat{x}_i$. **Restart the SP decoder with the channel messages and the manipulated bit as input messages. (Reset all other messages).** If the
SP decoder finds a codeword, stop and return the codeword.

In analogy to the previous section, we also present an algorithm which uses the (normalized) SP output marginals to select bits for manipulation. We use the following belief-based terms.

- **(Belief-based hard decision)** The belief-based hard decision of a bit $x_\ell$ is
  \[ \hat{x}_{\ell,SP} = \arg\max_{x_\ell} \tilde{p}(x_\ell|y), \]
  where $\tilde{p}(x_\ell|y)$ is the SP output marginal (or belief).

- **(Unsatisfied parity check equation)** A parity check equation $h$ (a row of the parity check matrix $H$) is unsatisfied with respect to the belief vector if
  \[ h\hat{x}_SP^T \neq 0. \]
  ($\hat{x}_{SP}$ is the vector of belief-based hard decisions).

**Algorithm 8.7. (Single Belief-Based Bit Flipping)** Given a linear code $C$ (and its parity check matrix $H$) and a sequence $y$ received from an AWGN channel. The SP decoder output for $y$ is not a codeword and the SP decoder output marginals $\tilde{p}(x_\ell|y)$ are given for all bits. Rank the unsatisfied parity checks $h \in H$ (unsatisfied w.r.t. the beliefs) according to
  \[ b(h) = \max_{h\hat{x}^T = 0} \sum_{i \in s(h)} \log \tilde{p}(x_i|y). \]
  Go through the list of unsatisfied parity check equations, beginning with the parity check with highest $b(h)$. For each check $h$, select the two bits from the support of $h$ with smallest $b_\ell = \max_{x_\ell} \tilde{p}(x_\ell|y)$. Use each selected bit for an SP decoding attempt as follows. (Stop if the maximum number of trials is reached).

1) Fix the decision for bit $i$ as $\hat{x}_i = \arg\min_{x_i} \log \tilde{p}(y_i|x_i)$.

2) Replace the channel message for the selected bit with the decision $\hat{x}_i$. Restart the SP decoder with the channel messages and the manipulated bit as input messages. (Reset all other messages). If the SP decoder finds a codeword, stop and return the codeword.
8.4 Tree Search Aided LDPC Decoding Algorithms

In this section we propose the use of tree searching to decode multiple bits from a codeword. The resulting partial codeword is then used for a new sum-product decoding attempt. For a given set of bits, only one tree is used for decoding, but multiple sets are used for the same received sequence. We choose some subset of parity check constraints from the parity check matrix $H$ and construct a (partial) code tree from those; the bits contained in the code tree are in the support set of the chosen parity check constraints.

Algorithm 8.8. (Tree Search Aided SP Decoding) Given a (full rank) low-density parity check matrix $H$ and a parameter $m \in \{1, \ldots, n-k\}$ and a sequence $y$ received from an AWGN channel. The SP decoder output for $y$ is not a codeword. Repeat the following steps until the maximum number of trials is reached.

1) Initialize $H_0 = H$. Process the last $m/2$ rows of $H_0$ from the bottom up, i.e. for $i = n-k \ldots, n-k+m/2-1$:

   i) Swap row $h_i$ with row $h_j$, $j \in \{1, \ldots, i\}$ such that the resulting $|S_\Delta(h_i)|$ is minimal. If multiple rows qualify choose the unsatisfied row with highest $ML(h)$. If all rows are satisfied, choose the row with highest $ML(h)$.

   ii) If $\ell$ is the position of the leading coefficient in row $h_{i+1}$, permute columns $1, \ldots, \ell - 1$ such that the columns in $S_\Delta(h_i)$ appear immediately to the left of $\ell$.

2) Process the rows $i = n-k+m/2, \ldots, n-k+m-1$:

   i) Swap row $h_i$ with row $h_j$, $j \in \{1, \ldots, i\}$ such that the resulting $|S_\Delta(h_i)|$ is minimal. If multiple rows qualify choose the satisfied row with highest $ML(h)$. If all rows are unsatisfied, choose the row with highest $ML(h)$.

   ii) If $\ell$ is the position of the leading coefficient in row $h_{i+1}$, permute columns $1, \ldots, \ell - 1$ such that the columns in $S_\Delta(h_i)$ appear immediately to the left of $\ell$. 
3) Form a code tree up to depth $m$ from the parity check matrix $H_0$, whose last $m$ rows are in row echelon form. Use the Forward Sweep Algorithm 3.3 with the Greedy Evaluation Function 5.4 to decode the depth-limited tree. (Each sweep stops at depth $m$).

4) Use the list of most likely partial codewords found by tree search. For each partial codeword, replace the channel messages for the decoded bits with the tree search output decisions. Restart the SP decoder with the channel messages and the manipulated bits as input messages. (Reset all other messages). If the SP decoder finds a codeword, stop and return the codeword. Otherwise repeat with the next partial codeword, until the maximum number of trials is reached.

Steps 1) and 2) are a modified version of Step 1) of Algorithm 4.5. Note that only the last $m$ rows in the resulting matrix $H_0$ are in row echelon form. If $m$ is chosen too large, it may not be possible to bring the entire matrix $H_0$ into row echelon form and the corresponding code tree of depth $m$ is incorrect. Since we only choose very small values for $m$ in this chapter, we assume that the code tree is correct without verifying.

In Step 1), the last $m/2$ rows are chosen to include flipped bits with high reliability in the code tree. This is based on the observation that a parity check whose support only includes high-reliability bits is likely to contain only one erroneous bit if the parity check is unsatisfied. In Step 2), another $m/2$ rows are added to the code tree. These are likely to contain no flipped bits. These rows are added to help the tree search decoder find the correct decisions for the partial codeword. If only unsatisfied parity checks are grouped together for decoding, it is likely that the partial codeword with maximum likelihood is different from the decisions of the globally optimal codeword at those positions. For this reason we include some parity checks which are likely to be error-free in the code tree. We also use a list of most likely partial codewords in Step 4).

As in the previous sections, we also test if knowledge of the SP output marginals improved the tree search aided decoder with a slightly modified algorithm.

Algorithm 8.9. (Belief-Based Tree Search Aided SP Decoding)
Given a (full rank) low-density parity check matrix $H$ and a parameter $m \in \{1, \ldots, n-k\}$. Given a sequence $y$ received from an AWGN channel..
The SP decoder output for \( y \) is not a codeword and the SP decoder output marginals \( \tilde{p}(x_i|y) \) are given for all bits. Repeat the following steps until the maximum number of trials is reached.

1) Initialize \( H_0 = H \). Process the last \( m/2 \) rows of \( H_0 \) from the bottom up, i.e. for \( i = n - k \ldots, n - k + m/2 - 1 \):

   i) Swap row \( h_i \) with row \( h_j, j \in \{1, \ldots, i\} \) such that the resulting \( |S_\Delta(h_i)| \) is minimal. If multiple rows qualify choose the unsatisfied (w.r.t. the belief vector) row with highest

   \[
   b(h) = \max_{h : x^T = 0} \sum_{i \in s(h)} \log \tilde{p}(x_i|y).
   \]

   If all rows are satisfied, choose the row with highest \( b(h) \).

   ii) If \( \ell \) is the position of the leading coefficient in row \( h_{i+1} \), permute columns \( 1, \ldots, \ell - 1 \) such that the columns in \( S_\Delta(h_i) \) appear immediately to the left of \( \ell \).

2) Process the rows \( i = n - k + m/2, \ldots, n - k + m - 1 \):

   i) Swap row \( h_i \) with row \( h_j, j \in \{1, \ldots, i\} \) such that the resulting \( |S_\Delta(h_i)| \) is minimal. If multiple rows qualify choose the satisfied (w.r.t. the belief vector) row with highest \( b(h) \). If all rows are unsatisfied, choose the row with highest \( b(h) \).

   ii) If \( \ell \) is the position of the leading coefficient in row \( h_{i+1} \), permute columns \( 1, \ldots, \ell - 1 \) such that the columns in \( S_\Delta(h_i) \) appear immediately to the left of \( \ell \).

3) Form a code tree up to depth \( m \) from the parity check matrix \( H_0 \), whose last \( m \) rows are in row echelon form. Use the Forward Sweep Algorithm 3.3 with the Greedy Evaluation Function 5.4 to decode the depth-limited tree. (Each sweep stops at depth \( m \)).

4) Use the list of most likely partial codewords found by tree search. For each partial codeword, replace the channel messages for the decoded bits with the tree search output decisions. Restart the SP decoder with the channel messages and the manipulated bits as input messages. (Reset all other messages). If the SP decoder finds a codeword, stop and return the codeword. Otherwise repeat with the next partial codeword, until the maximum number of trials is reached.
8.5 Simulation Results

Figures 8.5 and 8.6 show some simulated word error rates for an LDPC code of length \( n = 1008 \) and rate \( r = 1/2 \) (previously used in Section 8.2). The results in both figures are comparable; the belief-based algorithms perform slightly worse than the algorithms which use only the channel messages. The performance of the SP decoder is improved significantly.

Some simulated word error rates for a code of length \( n = 504 \) and rate \( r = 1/2 \) (previously used in Section 8.2) are shown in Figures 8.7 and 8.8. The results are consistent with those for the code of length \( n = 1008 \), with the belief-based algorithms showing slightly degraded but comparable performance as the channel message based algorithms.
Figure 8.5: Simulated word error rates for an LDPC code of length $n = 1008$ and rate $r = 1/2$. From top to bottom: SP decoding (50 iterations), SP decoding with at most 200 additional decoding attempts with the Single Bit Flipping Algorithm 8.6, SP decoding with at most 200 additional decoding attempts with the Single Bit Flipping Algorithm 8.6 and at most $10 \times 200$ additional SP decoding attempts with the Tree Search Aided SP Decoding Algorithm 8.8 ($m = 10$). 10 forward sweeps were used in each tree search and the 10 most likely partial codewords from each tree search were passed to the SP decoder.
Figure 8.6: Simulated word error rates for an LDPC code of length \( n = 1008 \) and rate \( r = 1/2 \). From top to bottom: SP decoding (50 iterations), SP decoding with at most 200 additional decoding attempts with the Single Belief-Based Bit Flipping Algorithm 8.7, SP decoding with at most 200 additional decoding attempts with the Single Belief-Based Bit Flipping Algorithm 8.7 and at most \( 10 \times 200 \) additional SP decoding attempts with the Belief-Based Tree Search Aided SP Decoding Algorithm 8.9 \((m = 10)\). 10 forward sweeps were used in each tree search and the 10 most likely partial codewords from each tree search were passed to the SP decoder.
Figure 8.7: Simulated word error rates for an LDPC code of length $n = 504$ and rate $r = 1/2$. From top to bottom: SP decoding (50 iterations), SP decoding with at most 100 additional decoding attempts with the Single Bit Flipping Algorithm 8.6, SP decoding with at most 200 additional decoding attempts with the Single Bit Flipping Algorithm 8.6 and at most $10 \times 100$ additional SP decoding attempts with the Tree Search Aided SP Decoding Algorithm 8.8 ($m = 10$). 10 forward sweeps were used in each tree search and the 10 most likely partial codewords from each tree search were passed to the SP decoder.
Figure 8.8: Simulated word error rates for an LDPC code of length $n = 504$ and rate $r = 1/2$. From top to bottom: SP decoding (50 iterations), SP decoding with at most 100 additional decoding attempts with the Single Belief-Based Bit Flipping Algorithm 8.7, SP decoding with at most 100 additional decoding attempts with the Single Belief-Based Bit Flipping Algorithm 8.7 and at most $10 \times 200$ additional SP decoding attempts with the Belief-Based Tree Search Aided SP Decoding Algorithm 8.9 ($m = 10$). 10 forward sweeps were used in each tree search and the 10 most likely partial codewords from each tree search were passed to the SP decoder.
We have presented multitree search decoders for linear block codes. The presented algorithms have controllable complexity and achieve near-optimal error rates for short block lengths. Several interesting results have been found.

- When no reliability ordering is used, specialized trees for low-density parity check codes outperform code trees for general codes. They have lower local rate and therefore a better distance profile. This is consistent with the behavior of the sequential decoder for convolutional codes.

- When reliability ordering is used, specialized trees for low-density parity check codes show inferior performance to general-code trees. This is due to the fact that a better reliability ordering can be achieved in general-code trees. The benefit of having lower local rates in a code tree is much smaller than the benefit of having a strict reliability ordering.

- The Greedy Evaluation Function 5.4 greatly outperforms the Basic Evaluation Function 5.2.
• Most of the computational complexity is used in unsuccessful decoding attempts. It pays to stop the search early when a codeword which is likely to be correct is found. The decoding complexity strongly depends on the signal-to-noise ratio. This is consistent with the behavior of $A^*$-search.

• Multitree decoding outperforms single tree decoding.

We have also proposed using tree search algorithms in combination with the sum-product decoder to improve the performance of the sum-product decoder at longer block lengths. Algorithms for fixing the values of some bits and decoding the remaining bits with a sum-product decoder have been previously proposed in [16] and [17]. Our approach differs in significant ways.

• We specifically correct high-reliability flips. (Other approaches choose the least reliable bits for correction).

• We handle the local constraints correctly to search over partial codewords. (In both [16] and [17] the manipulated bits are treated as information bits).

The combined tree search/sum-product decoder achieves a significant improvement over the sum-product decoder. In particular, we note:

• Single bit-flipping algorithms are surprisingly successful, particularly at medium to high signal-to-noise ratios.

• The performance of the proposed decoding scheme works well for longer codes ($n = 1008$). This is in contrast to the results presented in [17].

We believe that the Tree Search Aided SP Decoding Algorithm 8.8 can be greatly improved with a better choice of bit positions for manipulation. The bit positions most likely to cause the sum-product decoder to fail are difficult to decode locally, since the locally optimal partial codeword is unlikely to be part of the globally optimal codeword. How to choose good subsets of bit positions and the tradeoff between using multiple subsets versus using a longer list of partial codewords for the same subset is an interesting topic for future research.
Appendix A

The Sum-Product Decoder

We briefly review the sum-product (SP) decoder which is the standard decoder for LDPC codes. There is a vast literature about the sum-product decoder, the original publication where it was proposed in context with error correcting codes was in 1963 by R. Gallager [13]. A general introduction can be found in [19]. The sum product decoder is an instance of a graphical model and consists of two parts:

1) A graph (a factor graph).
2) An algorithm on the graph (the summary-product algorithm).

The sum-product decoder is also called belief propagation decoder, message passing decoder or iterative decoder.

A.1 Factor Graphs

We will use the notation of normal factor graphs (or Forney-style factor graphs) as they have been presented in [20].
Appendix A. The Sum-Product Decoder

Figure A.1: Factor graph for $f(x, y) = f_1(x)f_2(x, y)f_3(y)$

(Factor Graph) A factor graph is a graphical representation of a global factorizable function $f$. It consists of nodes and edges and obeys the following conventions:

- **(Global function)** $f$ is called the global function.
- **(Local function)** A local function is a factor of $f$, (e.g. $f = f_1 \cdot f_2$ consists of the local functions $f_1$ and $f_2$).
- **(Factor nodes)** A (unique) factor node belongs to each factor (local function) of $f$ (e.g. for $f = f_1 \cdot f_2$ there is a node for $f_1$ and a node for $f_2$).
- **(Edges)** There is a unique edge for every variable. Every edge is connected to at most two factor nodes. An edge which is connected to only one factor node is called a half edge.
- **(Connectedness)** An edge is connected with a factor node if the factor takes the edge variable as an argument.
- **(Configuration)** A configuration is a particular assignment of values to all variables.
- **(Configuration space)** The configuration space $\Omega$ is the domain of the global function $f$.

A generic example is shown in Figure A.1. Consider the code membership indicator function

$$I_C : \mathbb{Z}_2^n \rightarrow \{0, 1\} : x \mapsto \begin{cases} 1, & \text{if } x \in C \\ 0, & \text{else} \end{cases}$$

of a linear binary code. To build a factor graph of $I_C$ (and that of any binary linear block code), we need two types of local functions.
\( f = \)

\[
\begin{array}{c}
\hline
\hline

\end{array}
\]

\[
\begin{array}{c}
\hline
\hline

\end{array}
\]

\( f_\oplus \)

\[
\begin{array}{c}
\hline
\hline

\end{array}
\]

\[
\begin{array}{c}
\hline
\hline

\end{array}
\]

\textbf{Figure A.2:} An equality and a parity check node.

- **(Equality check function)** The discrete domain equality check function is
  \[
  f_\equiv(x, y, z) \triangleq \begin{cases} 
  1, & \text{if } x = y = z \\
  0, & \text{else}
  \end{cases}
  \]

- **(Parity check function)** The zero-sum constraint function (or the parity check function) is
  \[
  f_\oplus(x, y, z) \triangleq \begin{cases} 
  1, & \text{if } x \oplus y \oplus z = 0 \\
  0, & \text{else}
  \end{cases}
  \]
  where \( \oplus \) is addition in \( \mathbb{Z}_2 \).

Example: the parity check matrix of a binary repetition code of length \( n = 3 \) can be written as
\[
H = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}.
\]
The code consists of the codewords \( C = \{(0, 0, 0), (1, 1, 1)\} \). The code membership function can be written as
\[
I_C(x_1, x_2, x_3) = f_\oplus(x_1, x_3) \cdot f_\oplus(x_2, x_3) \quad (A.1)
\]
and a factor graph for the global function (A.1) can be seen in Figure A.3.

To build a factor graph which can be used for decoding, we need to connect the factor graph for the code constraints with a factor graph for the channel model to obtain a factor graph for the joint likelihood function \( p(y|x)I_C(x) \). Recall that we are only considering the AWGN channel. Since this channel is memoryless, we can factorize
\[
p(y|x) = \prod_{k=1}^{n} p(y_k|x_k).
\]
As an example, assume we receive a block $Y = (y_1, y_2, y_2)$ sent over an AWGN channel. The code used is the binary repetition code of length $n = 3$. We extend the factor graph for the code indicator function (shown in Figure A.3) with the factor graph for the channel model $p(y|x)$ to obtain the factor graph in Figure A.4.

**A.2 The Sum-Product Algorithm**

The summary-product algorithm can be defined for any commutative semi-ring with unity element. We again refer the reader to [19] for a general introduction. In this thesis we introduce a special instance of the summary-product algorithm (called the sum-product algorithm). The sum-product algorithm is defined over the real field, the operators are the real summation and the real product.

The basic operation in the sum-product algorithm is the computation of messages. We define messages implicitly by giving the message computation rule (A.2).

If a factor node is connected to only one edge, the outgoing message (the message pointing away from the node) is the node factor itself. Otherwise, the outgoing message is computed by the basic sum-product rule. Let $f(x_1, \ldots, x_n)$ be some function node. We compute the message

![Figure A.3: Factor graph for the membership indicator function of a repetition code of length $n = 3$.](image)
A.2. The Sum-Product Algorithm

Figure A.4: Factor graph for the joint likelihood function $p(y|x)I_C(x)$ for a memoryless channel and a repetition code of length $n = 3$.

$\mu$ pointing away from the node $f$ along the edge $x_k$, where $\mu_{in}$ are the messages pointing towards the factor node:

$$\mu_{out}(x_k) = \sum_{x_i : i \neq k} f(x_1, \ldots, x_n) \prod_{i \neq k} \mu_{in}(x_i). \quad (A.2)$$

As an example, consider Figure A.5. We compute the outgoing message $\mu(y)$ according to

$$\mu(y) = \sum_{x} \sum_{z} f(x, y, z) \mu(x) \mu(z)$$

for the example to the left and

$$\mu(x) = \sum_{y} \sum_{z} f(x, y, z) \mu(y) \mu(z)$$

for the example to the right (the arrows in the graph show the direction of the messages).

We apply the sum-product update rule to a simple factor tree, shown
Figure A.5: Computation of messages to the left and to the right.

in Figure A.6 to illustrate the relationship between messages and the global function $f$ in a factor graph.

- Factor $f_1$ and $f_2$ have no incoming messages, the outgoing messages from nodes $f_1$ and $f_2$ are therefore

$$
\overrightarrow{\mu}(u) = f_1(u) \\
\overrightarrow{\mu}(v) = f_2(u)
$$

(where the arrow denotes the horizontal direction of the message).

- The outgoing message from node $f_3$ along the edge $W$ is

$$
\overrightarrow{\mu}(w) = \sum_u \sum_v f_3(u, v, w) \cdot \overrightarrow{\mu}(u) \overrightarrow{\mu}(v) \\
= \sum_u \sum_v f_3(u, v, w) f_1(u) f_2(u),
$$

in other words it is the marginal function of $w$ over the local factors $f_1, f_2$ and $f_3$.

- Analogously, the outgoing message from node $f_4$ along the edge $W$ is

$$
\overrightarrow{\mu}(w) = \sum_x \sum_y f_4(x, y, w) \cdot \overrightarrow{\mu}(x) \overrightarrow{\mu}(y) \\
= \sum_x \sum_y f_4(x, y, w) f_5(x) f_6(y).
$$

- The product of both messages along the edge $W$ is therefore
\[ \overline{\mu}(w) \cdot \overline{\mu}(w) = \left( \sum_u \sum_v f_3(u, v, w) f_1(u) f_2(u) \right) \cdot \left( \sum_x \sum_y f_4(x, y, w) f_5(x) f_6(y) \right) \]

\[ = \sum_u \sum_v \sum_x \sum_y f(u, v, w, x, y) \]

where \( f(u, v, w, x, y) \) is the global function of the factor graph.

We can see that the sum-product computes marginals. A message over an edge is the marginal of the variable which belongs to the edge; the marginalized local function is depicted in the subgraph the message is pointing away from. Note that in the case where the factor graph is a tree, the sum-product algorithm computes the exact marginals. When the factor graph contains cycles, the computations are only approximations to the true marginals. Nevertheless the sum-product algorithm is very successful on loopy factor graphs if they are sparse, to the extent that it is the standard decoder for LDPC codes. There are two main differences to consider when implementing a sum-product decoder on a loopy factor graph:

1) The messages are initialized to be neutral, i.e. \( \mu = 1 \) for all messages.

2) The algorithm becomes iterative, i.e. the same message gets recomputed multiple times.
3) Some schedule needs to be found to decide in which order the messages are updated (and when to stop).

### A.3 Iterative Channel Decoding

Assume we are given some received sequence $Y$ sent over an AWGN channel. If $p(x)$ is uniformly distributed and $y$ is known, the relations

\[
p(y|x) I_C(x) \propto p(x, y) I_C(x) \quad \text{(A.3)}
\]

\[
p(x|y) I_C(x) \propto p(x|y) I_C(x) \quad \text{(A.4)}
\]

hold. Likewise,

\[
p(x_\ell | y) = \sum_{x_i : i \neq \ell} p(x_i | y) I_C(x) \quad \text{(A.5)}
\]

\[
\propto \sum_{x_i : i \neq \ell} p(x_i | y) I_C(x) \quad \text{(A.6)}
\]

\[
\propto \sum_{x_i : i \neq \ell} p(y|x) I_C(x). \quad \text{(A.7)}
\]

The sum-product algorithm computes the marginal (A.7) for every variable $x_\ell$ in the factor graph (or an approximation thereof if the factor graph contains loops). These marginals are then used for symbolwise decoding as in

\[
\hat{x}_\ell = \operatorname{argmax}_{x_\ell} p(x_\ell | y) \quad \text{(A.8)}
\]

\[
= \operatorname{argmax}_{x_\ell} \sum_{x_i : i \neq \ell} p(y|x) I_C(x). \quad \text{(A.9)}
\]

Note that this method may give block decisions $\hat{x}$ which are not valid codewords, since the decisions are made symbolwise.
List of Figures

2.1 Generic communication system ................................. 8
2.2 AWGN Channel ............................................. 10
2.3 Channel decoder ............................................. 11

3.1 Example tree .................................................. 15
3.2 Code tree for to the code 3.1. ............................... 16

4.1 Code tree for parity check matrix (4.1). .......................... 22
4.2 Code tree from a parity check matrix $H$. $H$ is in row echelon form and $H = (h^T_1, h^T_2, \ldots, h^T_{n-k})^T$. ............... 23
4.3 Average $|S_{\Delta}|$ for general-code trees computed with Algorithm 4.3 for different codes of length $n = 100$ and rate 1/2. From top to bottom: A $(p = 1/2)$-random code (solid line), a (5,10)-regular code (solid line) and a (3,6)-regular code (dashed line). (The graphs for the $(p = 1/2)$-code and the (5,10)-regular code cannot be distinguished in the plot.) ............................................. 31

4.4 Average $|S_{\Delta}|$ for low-density code trees computed with Algorithm 4.5 for different codes of length $n = 100$ and rate 1/2. A (5,10)-regular code (simulated with $m = 15$, dashed line) and a (3,6)-regular code (simulated with $m = 27$, solid line). .......................... 33
6.1 Simulated word error rates for a (3,6)-regular code of length $n = 100$ using multitree search decoding with the basic evaluation function (5.2). In each decoding attempt, at most 5 different trees were searched in at most 50 forward sweeps per tree. From top to bottom: Multitree search with general-code trees (Algorithm 4.3), SP decoding with 50 iterations (dashed line), multitree search with general-code trees with reliability ordering (Algorithm 4.4 with $\lambda = 1$), ML lower bound (dashed line).

6.2 Simulated word error rates for a (3,6)-regular code of length $n = 100$ using multitree search decoding with the basic evaluation function (5.2). In each decoding attempt, at most 5 different trees were searched in at most 50 forward sweeps per tree. From top to bottom: SP decoding with 50 iterations (dashed line), multitree search with low-density code trees (Algorithm 4.5), multitree search with low-density code trees with reliability ordering (Algorithm 4.7), multitree search with precomputed low-density code trees selected with Algorithm 4.9 (500 code trees were precomputed and the first 40 bits compared at point of selection), ML lower bound (dashed line).

6.3 Simulated word error rates for a ($p = 1/2$)-code of length $n = 100$ and rate $r = 1/2$ using multitree search decoding with the basic evaluation function (5.2). Only general-code trees with reliability ordering were used (Algorithm 4.4 with $\lambda = 1$). From top to bottom: 1 tree and at most 200 forward sweeps, at most 5 different trees and at most 50 forward sweeps per tree, 1 tree and at most 1000 forward sweeps, at most 10 different trees and at most 100 forward sweeps per tree, at most 10 different trees and at most 200 forward sweeps per tree, ML lower bound (dashed line).
6.4 Simulated word error rates for different codes of length $n = 100$ and rate $r = 1/2$ using multitree search decoding with the basic evaluation function (5.2). Only general-code trees with reliability ordering were used (Algorithm 4.4 with $\lambda = 1$). At most 10 code trees were searched in at most 200 forward sweeps per tree. From top to bottom: (3,6)-regular code, (5,10)-regular code, $(p = 1/2)$-code. An ML lower bound is shown for each code (dashed lines).

6.5 Simulated word error rates for different codes of length $n = 100$ and rate $r = 1/2$ using multitree search decoding with the basic evaluation function (5.2). Only general-code trees with reliability ordering were used (Algorithm 4.4 with $\lambda = 1$). At most 5 code trees were searched in at most 50 forward sweeps per tree. From top to bottom: (3,6)-regular code, (5,10)-regular code, $(p = 1/2)$-code. An ML lower bound is shown for each code (dashed lines).

6.6 Simulated word error rates for a (3,6)-regular code of length $n = 200$ using multitree search decoding with the basic evaluation function (5.2). At most 25 code trees were searched in at most 200 forward sweeps per tree. From top to bottom: SP decoding with 50 iterations (dashed line), multitree search with low-density code trees with reliability ordering (Algorithm 4.7), multitree search with general-code trees with reliability ordering (Algorithm 4.4 with $\lambda = 1$), ML lower bound (dashed line).
7.1 Simulated word error rates for a (3,6)-regular code of length $n = 100$ using multitree search decoding. In each decoding attempt, at most 5 different code trees were searched in at most 50 forward sweeps. From top to bottom: Multitree search with general-code trees (Algorithm 4.3) and the Basic Evaluation Function (5.2), SP decoding with 50 iterations (dashed line), multitree search with general-code trees (Algorithm 4.3) and the Greedy Evaluation Function (5.4), multitree search with general-code trees with reliability ordering (Algorithm 4.4 with $\lambda = 1$) and the Basic Evaluation Function (5.2), multitree search with general-code trees with reliability ordering (Algorithm 4.4 with $\lambda = 1$) and the Greedy Evaluation Function (5.4), ML lower bound (dashed line).

7.2 Simulated word error rates for a (3,6)-regular code of length $n = 100$ using multitree search decoding. In each decoding attempt, at most 5 different code trees were searched in at most 50 forward sweeps per tree. From top to bottom: SP decoding with 50 iterations (dashed line), multitree search with low-density code trees (Algorithm 4.5) and the Basic Evaluation Function (5.2), multitree search with low-density code trees with reliability ordering (Algorithm 4.7) and the Basic Evaluation Function (5.2), multitree search with precomputed low-density code trees selected with Algorithm 4.9 (500 code trees were precomputed and the first 40 bits compared at point of selection) and the Basic Evaluation Function (5.2), multitree search with low-density code trees (Algorithm 4.5) and the Greedy Evaluation Function (5.4), multitree search with low-density code trees with reliability ordering (Algorithm 4.7) and the Greedy Evaluation Function (5.4), multitree search with precomputed low-density code trees selected with Algorithm 4.9 (500 code trees were precomputed and the first 40 bits compared at point of selection) and the Greedy Evaluation Function (5.4), ML lower bound (dashed line).
7.3 Simulated word error rates for a $(3,6)$-regular code of length $n = 200$ using multitree search decoding. In each decoding attempt, at most 25 different code trees were searched in at most 200 forward sweeps per tree. From top to bottom: SP decoding with 50 iterations (dashed line), multitree search with low-density code trees with reliability ordering (Algorithm 4.7) and the Basic Evaluation Function (5.2), multitree search with general-code trees with reliability ordering (Algorithm 4.4 with $\lambda = 1$) and the Basic Evaluation Function (5.2), multitree search with general-code trees with reliability ordering (Algorithm 4.4 with $\lambda = 1$) and the Greedy Evaluation Function (5.4), multitree search with low-density code trees with reliability ordering (Algorithm 4.7) and the Greedy Evaluation Function (5.4), multitree search with precomputed low-density code trees selected with Algorithm 4.9 (500 code trees were precomputed and the first 80 bits compared at point of selection) and the Greedy Evaluation Function (5.4), ML lower bound (dashed line).

8.1 Simulated word error rates for an LDPC code of length $n = 1008$. From top to bottom: SP decoding (50 iterations), correct bit enforcement test (Algorithm 8.3 with $|S| = 200$), single bit flipping test (Algorithm 8.2 with 1 least reliable bit), single bit flipping test (Algorithm 8.2 with at most 10 least reliable bits), single bit flipping test (Algorithm 8.2 with 1 most reliable bit), single bit flipping test (Algorithm 8.2 with at most 10 most reliable bits). (Note that in this plot the WER for the correct bit enforcement test and the single bit flipping test with 1 least reliable bit cannot be distinguished).

8.2 Simulated word error rates for an LDPC code of length $n = 504$. From top to bottom: SP decoding (50 iterations), correct bit enforcement test (Algorithm 8.3 with $|S| = 100$), single bit flipping test (Algorithm 8.2 with 1 least reliable bit), single bit flipping test (Algorithm 8.2 with at most 10 least reliable bits), single bit flipping test (Algorithm 8.2 with 1 most reliable bit), single bit flipping test (Algorithm 8.2 with at most 10 most reliable bits).
8.3 Simulated word error rates for an LDPC code of length $n = 1008$. $b_\ell = \max_{x_\ell} \log \tilde{p}(x_\ell | y)$ was used for selecting bits, where $\tilde{p}(x_\ell | y)$ is the bit’s SP output marginal and $y$ is the received channel sequence. From top to bottom: SP decoding (50 iterations), correct belief-based bit enforcement test (Algorithm 8.5 with $|S| = 200$), single belief-based bit flipping test (Algorithm 8.4 with 1 bit with smallest $b_\ell$), single belief-based bit flipping test (Algorithm 8.4 with at most 10 bits with smallest $b_\ell$), single belief-based bit flipping test (Algorithm 8.4 with 1 bit with highest $b_\ell$), single belief-based bit flipping test (Algorithm 8.4 with at most 10 bits with highest $b_\ell$). . . . . 61

8.4 Simulated word error rates for an LDPC code of length $n = 504$. $b_\ell = \max_{x_\ell} \log \tilde{p}(x_\ell | y)$ was used for selecting bits, where $\tilde{p}(x_\ell | y)$ is the bit’s SP output marginal and $y$ is the received channel sequence. From top to bottom: SP decoding (50 iterations), correct belief-based bit enforcement test (Algorithm 8.5 with $|S| = 100$), single belief-based bit flipping test (Algorithm 8.4 with 1 bit with smallest $b_\ell$), single belief-based bit flipping test (Algorithm 8.4 with at most 10 bits with smallest $b_\ell$), single belief-based bit flipping test (Algorithm 8.4 with 1 bit with highest $b_\ell$), single belief-based bit flipping test (Algorithm 8.4 with at most 10 bits with highest $b_\ell$). . . . . 62

8.5 Simulated word error rates for an LDPC code of length $n = 1008$ and rate $r = 1/2$. From top to bottom: SP decoding (50 iterations), SP decoding with at most 200 additional decoding attempts with the Single Bit Flipping Algorithm 8.6, SP decoding with at most 200 additional decoding attempts with the Single Bit Flipping Algorithm 8.6 and at most $10 \times 200$ additional SP decoding attempts with the Tree Search Aided SP Decoding Algorithm 8.8 ($m = 10$). 10 forward sweeps were used in each tree search and the 10 most likely partial codewords from each tree search were passed to the SP decoder. . . 68
8.6 Simulated word error rates for an LDPC code of length $n = 1008$ and rate $r = 1/2$. From top to bottom: SP decoding (50 iterations), SP decoding with at most 200 additional decoding attempts with the Single Belief-Based Bit Flipping Algorithm 8.7, SP decoding with at most 200 additional decoding attempts with the Single Belief-Based Bit Flipping Algorithm 8.7 and at most $10 \times 200$ additional SP decoding attempts with the Belief-Based Tree Search Aided SP Decoding Algorithm 8.9 ($m = 10$). 10 forward sweeps were used in each tree search and the 10 most likely partial codewords from each tree search were passed to the SP decoder. 

8.7 Simulated word error rates for an LDPC code of length $n = 504$ and rate $r = 1/2$. From top to bottom: SP decoding (50 iterations), SP decoding with at most 100 additional decoding attempts with the Single Bit Flipping Algorithm 8.6, SP decoding with at most 200 additional decoding attempts with the Single Bit Flipping Algorithm 8.6 and at most $10 \times 100$ additional SP decoding attempts with the Tree Search Aided SP Decoding Algorithm 8.8 ($m = 10$). 10 forward sweeps were used in each tree search and the 10 most likely partial codewords from each tree search were passed to the SP decoder.

8.8 Simulated word error rates for an LDPC code of length $n = 504$ and rate $r = 1/2$. From top to bottom: SP decoding (50 iterations), SP decoding with at most 100 additional decoding attempts with the Single Belief-Based Bit Flipping Algorithm 8.7, SP decoding with at most 100 additional decoding attempts with the Single Belief-Based Bit Flipping Algorithm 8.7 and at most $10 \times 200$ additional SP decoding attempts with the Belief-Based Tree Search Aided SP Decoding Algorithm 8.9 ($m = 10$). 10 forward sweeps were used in each tree search and the 10 most likely partial codewords from each tree search were passed to the SP decoder.

A.1 Factor graph for $f(x, y) = f_1(x)f_2(x, y)f_3(y)$.
A.2 An equality and a parity check node. .......................... 77
A.3 Factor graph for the membership indicator function of a repetition code of length $n = 3$. ___________________________ 78
A.4 Factor graph for the joint likelihood function $p(y|x)I_C(x)$ for a memoryless channel and a repetition code of length $n = 3$. ___________________________ 79
A.5 Computation of messages to the left and to the right. ... 80
A.6 Factor tree. ___________________________________________ 81
Bibliography


