Universiteit Utrecht

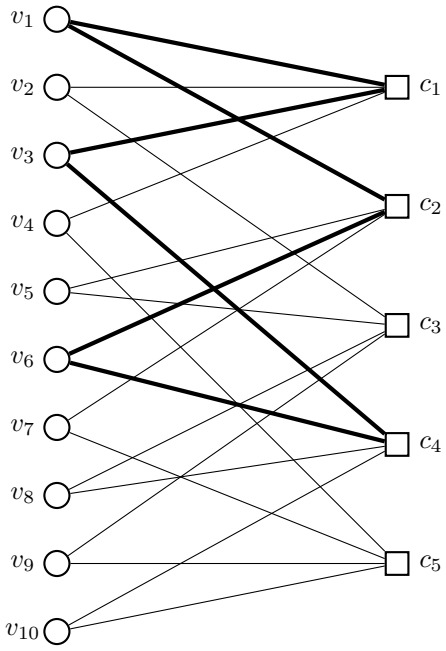
Faculteit Bètawetenschappen

Irregular LDPC Code Design using Large Girth Tanner Graphs

Master Thesis

Hannah Onverwagt

Mathematical Sciences



Supervisors:

Dr. Ivan Kryven
Mathematical Institute, Utrecht University

Prof. Dr. Willemien Kets
Mathematical Institute, Utrecht University

April 15, 2023

Abstract

Low-density parity-check (LDPC) codes are error-correcting codes that have been shown to have good performance approaching Shannon's limit. Although initially developed in the early 1960's, LDPC codes have experienced a remarkable comeback in recent years. As a result, numerous modern communication standards have embraced LDPC codes. To achieve high processing speed and energy efficiency, both the encoding and decoding processes must have a low level of complexity.

LDPC codes can be represented graphically by Tanner graphs. Tanner graphs with a large girth are often used because of their excellent performance in terms of error correction and transmission efficiency. The primary objective of this thesis is to explore the encoding and decoding processes to answer the question why a large girth in the Tanner graph is beneficial. Moreover, this thesis presents methods to construct these large girth Tanner graphs for irregular LDPC codes.

Acknowledgement

I would like to express my sincere gratitude to all those who have supported me throughout this process. First and foremost, I would like to thank my supervisor Ivan Kryven for his guidance, insightful feedback and support throughout the research process. I would also like to thank Mark Snelders for providing helpful feedback and motivation when it was needed. I want to thank Joris van der Hijden and Leon Goertz for their support and the many insightful conversations we had about the topic, during which I was able to share the challenging aspects of writing a thesis.

Lastly, I want to thank my family and friends for their unwavering love, support, and encouragement, especially during times of difficulty. Their belief in me has been a constant source of strength and motivation.

Contents

1 Introduction

In a communication system, information is transmitted from a sender to a receiver over a channel. In real-world scenarios, this channel is often affected by various sources of noise and interference that can corrupt the transmitted signal. The effect of noise on the transmitted signal can result in errors, or even complete loss of data, at the receiver end, which can severely degrade the performance of the system. To overcome these effects, various techniques are used, including error-correcting codes. These codes are used to add redundancy to the transmitted signal, allowing the receiver to detect and correct errors caused by noise.

We all face this problem during ordinary voice communication, when a word is misheard in a conversation. In practice, to recover from this corruption, we simply repeat ourselves. This is called the repetition code and it is one of the simplest error correction codes. However, it fails whenever the same word is misheard in the repetition. This unfortunately happens quite often and therefore this strategy leaves us very vulnerable to failure. What we could do, is repeat ourselves even more often. But this means we have to send a lot more data than only the message. The rate of a code represents the ratio between the number of information bits and the number of sent data bits. Repeating many times lowers the probability of an error, but also lowers the efficiency. We need to find a balance between the error probability and the code rate.

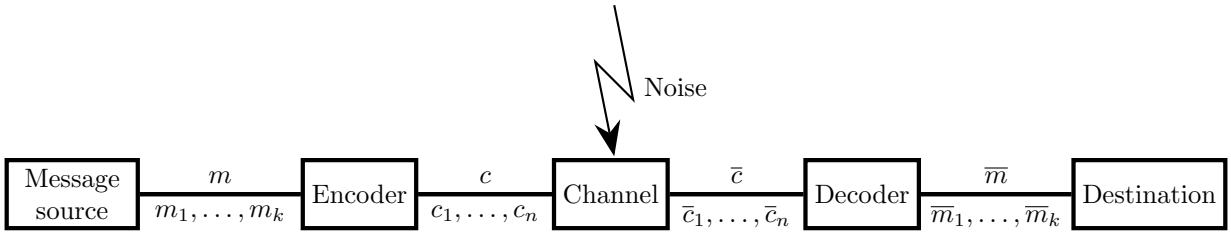


Figure 1: A message source sends a message m containing of $k \in \mathbb{N}$ bits $m_1, \ldots, m_k$, whereafter the encoder encodes the message with a so called 'codeword' c which consist of $n \in \mathbb{N}$ bits, where $n \geq k$. The noise in the channel changes the codeword c into $\overline{c}$ by e.g. flipping or corrupting bits. The decoder then tries to restore the original codeword m from the received value $\overline{c}$, we call this restored message $\overline{m}$. The goal is to find an encoder and decoder function such that $\overline{m} = m$ with high probability for every message.

Shannon's Noisy Channel Theorem [1] is a fundamental result in information theory that states that for any given communication channel, there is a limit to the amount of information that can be reliably transmitted over the channel with a given level of reliability. In other words, it is possible to transmit information over a given channel reliably if and only if the code rate is smaller than the *channel capacity*. Shannon proves that for every rate and error probability ϵ there exists a code with this rate which has an error probability of at most ϵ. Unfortunately, the proof is non-constructive, so Shannon does not tell us how we can find these codes. After Shannon presented this theorem in 1948 a lot of research has been done to construct codes that perform close to this capacity.

Low-density parity-check (LDPC) codes are a type of linear error-correcting codes that have become a popular choice in communication systems. These codes are advantageous because of their low decoding complexity, flexibility and close to channel capacity performance. LDPC codes were presented by Robert Gallager [2] in 1962, but did not receive much attention at that moment, mostly because of the lack of implementation. Computers at that time were not powerful enough to verify that LDPC could approach the channel capacity. Only after Turbo codes, another type of error-correcting codes with low decoding complexity and reliable performance, were introduced in 1993, LDPC codes were reintroduced by MacKay and Neal [3]. The name 'low-density parity-check' codes refers to the structure of the code, which is represented by a sparse parity-check matrix. This property makes LDPC codes efficient in terms of both storage and computational power.

A separate goal in code construction is having an easy hardware implementation, meaning that to make efficient use of LDPC codes we need effective encoder and decoder functions. Encoding of linear codes can be done efficiently via their generator matrix. However, LDPC codes are defined by their parity-check matrix

and forming the generator matrix from this parity-check matrix can be complex. It is preferred to accomplish encoding directly via the parity-check matrix. Richardson and Urbanke designed an algorithm that uses the approximate upper triangular form of a matrix [4] to improve the encoding complexity. Finding an optimal decoder for LDPC codes is an NP-complete problem [5]. However, using message passing algorithms we can find a polynomial algorithm that performs close to the channel capacity [6]. This Belief Propagation (BP) algorithm uses the Tanner graph as representation of a graph. Messages containing a belief about the state of the variable nodes is send over the edges. These believes are revised and updated via so-called update rules. Under the assumption that the Tanner graph is a tree we can show that this BP decoder is an optimal decoder. Nonetheless, this assumption is not so relevant, since codes whose Tanner graph does not have cycles are inconvenient as their minimum distance is low. It has been shown that small cycles and small stopping sets influence the decoder and reduce the error capability of a code [7][8]. We are thus looking for codes that avoid these. We will focus our attention on avoiding small cycles by looking at Tanner graphs with a large girth, where the girth of a Tanner graph is defined as the size of the smallest cycle.

The next challenge lies in constructing codes whose Tanner graph has a large girth. Luby suggested in 1998 to use irregular LDPC codes instead of regular LDPC codes [9]. These codes have a performance even closer to the channel capacity than the regular codes Gallager proposed. The interest in research therefore moved to irregular codes and since then we examine the constructing of irregular codes with a large girth. The goal of this thesis is first to understand why Tanner graphs with large girth are needed and, second, to construct Tanner graphs with large girth to generate codes with a good performance.

Organization

We start with covering the background materials in Chapter 2. We examine the basics of coding theory; definition of codes, examples, LDPC codes and their different representations, including the Tanner graph. We define noisy channels in Section 2.2 and lastly, in Section 2.4 we give some essential information for encoding and decoding functions.

Chapter 3 discusses the encoding problem of LDPC codes, presenting a new efficient algorithm as presented by Richardson and Urbanke [4]. In Chapter 4 we cover the Belief Propagation (BP) algorithm, which is a decoding algorithm for decoding LDPC codes. This message passing algorithm uses Tanner graphs as graphical representation of LDPC codes. We work on optimizing the algorithm and discuss the influence of stopping sets and cycles. We introduce the girth and show why codes with a large girth are favored.

Lastly, in Chapter 5 we engage in the challenge of constructing codes with a Tanner representation without small cycles. Irregular codes with their corresponding degree sequences are introduced in Sections 5.2 and 5.3. Existing constructions of Tanner graphs with a large girth are presented and using some of these ideas we propose a method for constructing Tanner graphs with a large girth realizing a given degree sequence in Section 5.6.

2 Coding theory

This chapter introduces the fundamental concepts and terminology related to error-correcting codes. In Section 2.1 we cover the basics of error-correcting codes in general, and Section 2.2 describes two types of noisy channels. In Section 2.3, we delve into some specific codes and we introduce LDPC codes. Lastly, in Section 2.4 we discuss some basic encoding and decoding methods.

2.1 Preliminaries

There are two main types of codes: block codes and convolutional codes. Block codes operate on fixed-length blocks of data, while convolutional codes process data streams without dividing them into blocks. Block codes encode each block of data into a longer block, and each block is independent of each other. The output of convolutional codes depends not only on the present input bits, but also on previous input bits that are stored in memory. We will focus on the second type, since low-density parity-check (LDPC) codes are block codes, and they work with finite blocks of data. In this thesis we will refer to block codes simply as codes. The notation and terminology used to describe codes in this thesis are largely based on Richardson and Urbanke's book [5].

Definition 2.1 (Block code)**.** A *block code* $\mathcal{C}$ of length n and cardinality M over a field $\mathbb{F}_q$ is a non-empty collection of M elements from $\mathbb{F}_q^n$, *i.e.*,

$$\mathcal{C} = \{x_1, \ldots, x_M\}, \qquad x_i \in \mathbb{F}_q^n, 1 \leq i \leq M.$$

Its elements are called codewords.

Definition 2.2 (Linear code)**.** We call a (block) code *linear* if its codewords are closed under addition and scalar multiplication, this means that for every codewords c_1, c_2 the sum $c_1 + c_2$ and $\alpha c_1, \forall \alpha \in \mathbb{F}$ are also codewords. A linear code of length n and dimension k is also called an $[n, k]$ code.

A block code is defined by the set of its codewords. For linear block codes only a basis of the code is needed to define all its codewords and thus the code. It is easy to see that each linear block code contains the all-zero codeword.

In this thesis we will focus on binary codes and so we will work over the field $\mathbb{F}_2$, therefore we often omit the notation of $\mathbb{F}_q$ and simply write $\mathbb{F}$.

A code consists of message bits, which contain the information you want to send, and auxiliary bits, to protect the data. The rate of a code is defined as the ratio of message bits in a codeword to the total number of bits in a codeword.

Definition 2.3 (Code rate)**.** The *rate* of a code $\mathcal{C}$ of length n and cardinality M is $r = \frac{1}{n} \log_2 M$. The *rate* of a $[n, k]$-code is $r = \frac{k}{n}$.

Repeating the message more often in the aforementioned mentioned repetition code increases the probability that the receiver obtains the right message, but it reduces the efficiency. This efficiency is expressed as the code rate. The rate of the 3-repetition code is $r = \frac{1}{3} \log_2 2 = \frac{1}{3}$, where $M = 2$, since $\mathbb{F} = \mathbb{F}_2$. A high code rate means that the number of message bits is close to the number of transmitted bits, this however gives us a high probability of failure. A lower code rate gives us more protection, since we send more auxiliary bits, but it is also more expensive, we can send less messages in the same time frame. Therefore, we need to find a balance between the code rate and the probability of failure.

The following concepts of the Hamming weight and the Hamming distance will help to analyze error-correcting codes in relation to the code's minimum distance.

Definition 2.4 (Hamming distance)**.** The *Hamming weight* $w(x)$ of a string x is equal to the number of non-zero elements in x. The *Hamming distance* $d(x, y)$ is the number of positions in which x differs from y, *i.e.*,

$$d(x, y) := \sum_{i=1}^{n} |x_i - y_i|.$$

Note that there is a relation between the Hamming distance and Hamming weight, namely $d(x, y) = d(x - y, 0) = w(x - y)$. Moreover, note that the Hamming distance is non-negative $d(x, y) \geq 0$, symmetric $d(x, y) = d(y, x)$ and $d(\cdot, \cdot)$ satisfies the triangle inequality, *i.e.*, $d(x, z) \geq d(x, y) + d(y, z)$.

Definition 2.5 (Minimum distance)**.** The *minimum distance* d_{min} of a code $\mathcal{C}$ is defined as

$$d_{min} := \min\{d(x, y) \quad | \quad x, y \in \mathcal{C}, x \neq y\}.$$

The following propositions tells us that instead of computing the minimum distance of a code we can also compute the minimum weight of all the non-zero codewords, which is preferred because of the lower complexity.

Proposition 2.6. *For a linear code $\mathcal{C}$, the minimum distance is equal to the minimum weight of the non-zero codewords.*

Proof. By utilizing the properties of the minimum distance, we obtain

$$d_{min} = \min_{x,y \in \mathcal{C}, x \neq y} d(x, y) = \min_{x,y \in \mathcal{C}, x \neq y} d(x - y, 0) = \min_{z \in \mathcal{C}, z \neq 0} d(z, 0) = \min_{z \in \mathcal{C}, z \neq 0} w(z).$$

□

What can we tell about the recovery of a codeword from the minimum distance? A code with minimal distance d can detect up to $d - 1$ bit flips errors and perfectly correct up to $\lfloor (d - 1)/2 \rfloor$ bit flip errors. This is because flipping up to $d - 1$ bits can never result in another codeword, otherwise the distance between these codewords is smaller than d, hence we can detect such an error if the corrupted codeword is not in the code anymore. Detecting errors is not as powerful as correcting errors but it can nevertheless be valuable. Furthermore, if all codewords are guaranteed to be different for at least d positions, then the set of strings that differ at most $(d - 1)/2$ bits are unique for every codeword. Thus from a received string with up to $(d - 1)/2$ errors we can fully restore the original message. Under certain circumstances the error correcting capability of a code can exceed this bound [10].

From the definition, we have seen that a code can be described by the set of its codewords. For linear codes however is it more convenient to give a basis of the codewords, such that all the other codewords can be constructed from this set. We implement this by giving a generator matrix for a code, where the rows contain the codewords of a basis for this code.

Definition 2.7 (Generator matrix)**.** A matrix G of dimension $k \times n$, with entries in $\mathbb{F}$, is a *generator matrix* for a linear code $\mathcal{C}$ of length n and dimension k if it has full rank and its rows generate $\mathcal{C}$ over $\mathbb{F}$, *i.e.*,

$$\mathcal{C} = \{xG \quad | \quad x \in \mathbb{F}^k\}.$$

We say that the rows of a generator matrix 'generate' the code, all rows and all linear combinations of the rows of G are codewords. Note that G uniquely determines the code $\mathcal{C}$. However, a code can have multiple generator matrices. We say a generator matrix is in *systematic form* if

$$G = \begin{bmatrix} I_k & | & A \end{bmatrix}, \tag{2.1}$$

where I_k is the $k \times k$ identity matrix and A is a $k \times (n - k)$ matrix. With elementary row operations and column permutations every generator matrix can be transformed in its equivalent systematic form.
We define the *dual* code $\mathcal{C}^\perp$ for every linear code $\mathcal{C}$ as

$$\mathcal{C}^\perp = \{x \in \mathbb{F} \quad | \quad Gx^T = \mathbf{0}^T\}.$$

Any $(n-k) \times k$ generator matrix of the code $\mathcal{C}^T$ is called a *parity-check matrix* of the original code $\mathcal{C}$ and is denoted by H. From the rank-nullity theorem it follows that $GH^T = \mathbf{0}$. This means that for all codewords c in the code, and only for these, $Hc^T = \mathbf{0}$ must hold.

If the generator matrix G of a code $\mathcal{C}$ is in systematic form, then the parity-check matrix H of this code can be computed easily from it

$$H = \begin{bmatrix} A^T & | & I_{n-k} \end{bmatrix}. \tag{2.2}$$

To find a parity-check matrix for any linear code we first put the generator matrix in the systematic form by performing elementary row operations and swapping columns if necessary. Then we can use (2.2) to find a parity-check matrix. On the other hand, if we have the parity-check matrix and we want to find a generator matrix we can bring the parity-check matrix in its systematic form as in (2.2) and use (2.1) to find a generator matrix.

2.2 Noisy channels

Noisy communication channels are the reason that error-correcting codes are used. In the real world, each communication channel is noisy. The following definition of [11] gives a formal notation of a noisy channel.

Definition 2.8. A (discrete) channel is a tuple $(X, P_{Y|X}, Y)$ such that X and Y are finite sets, and for any $x \in X$, the function $P_{Y|X}$ is a probability distribution.

The set X represents the possible inputs of the channel. In our case, this is the set of all possible codewords $\mathcal{C}$. Y represents that possible outputs of the channel, let us call the output $\bar{c}$, that is received after the corruption of the channel.
For example, one could think about a channel where bits are flipped. When transmitting messages over this channel, the channel will flip bits with a certain probability. The simplest form of such a channel is the binary symmetric channel.

Definition 2.9. We define the *binary symmetric channel* (BSC) with parameter p by $X = Y = \{0, 1\}$ and

$$\mathbb{P}_{Y|X}(0 \mid 0) = \mathbb{P}_{Y|X}(1 \mid 1) = 1 - p$$
$$\mathbb{P}_{Y|X}(0 \mid 1) = \mathbb{P}_{Y|X}(1 \mid 0) = p.$$

So with a probability of p the input will be flipped and with probability $1 - p$ it remains unaffected. This channel be visualized by Figure 2.

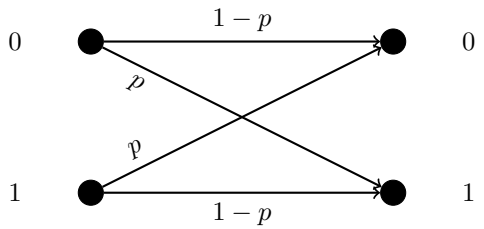


Figure 2: Binary Symmetric Channel

This channel is an example of a *memory less* channel. The probability distribution of the output solely relies on the current input. When the channel is used repeatedly, the channel distribution remains unaffected by prior inputs and outputs.

Another example of a memory less channel is the binary erasure channel (BEC). This is an erasure channel, it models situations where information may be lost, but is never corrupted. The BEC captures these erasure in the simplest form: when transmitted, individual bits are either received correctly, or known to be lost.

Definition 2.10. We define the *binary erasure channel* (BEC) with parameter ϵ by $\mathcal{X} = \{0, 1\}$, $\mathcal{Y} = \{0, 1, ?\}$, where '?' is the *erasure symbol* and ϵ is the erasure probability.

$$P_{Y|X}(0 \mid 0) = P_{Y|X}(1 \mid 1) = 1 - \epsilon$$
$$P_{Y|X}(? \mid 1) = P_{Y|X}(? \mid 0) = \epsilon.$$

The input is erased with probability ϵ and with probability $1 - \epsilon$ it remains unaffected, the channel is illustrated in Figure 3.

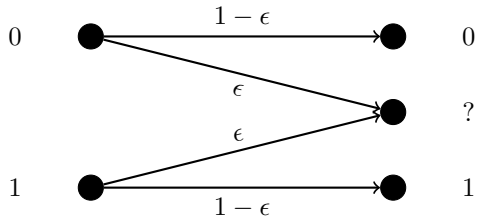


Figure 3: Binary Erasure Channel

Even though the BEC might appear to be an excessively basic and impractical model of a communication channel, there are real-world instances of it. We will study how error-correcting codes behave on these channels, since these channel simplify a lot of the ideas. For example, analysis of LDPC codes is possible on the BEC, whereas for other channels this is more difficult to achieve. Despite its simplicity, the BEC can provide valuable insights into what occurs in more general scenarios. This is because a lot of the concepts that we face during the study of BEC hold also in more general cases [5].

2.3 Overview of different codes

Before giving the definition of LDPC codes, we will take a look at some examples of other codes. The repetition code is one we have already seen.

Example 2.11 (Repetition code)**.** The code $\mathcal{C} = \{(\alpha, \ldots, \alpha) \mid \alpha \in \mathbb{F}^n\}$ is called the *n-times repetition code*. The generator matrix of this code is the $1 \times n$ matrix $G = \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix}$. △

The efficiency of this code is not very high, as it has a rate of $1/n$. The next code, the parity-check code, uses a smart technique and only needs one auxiliary bit, so the rate is $(n-1)/n$. This last bit is equal to the sum of the first $n - 1$. Therefore, if one bit flips, the decoder can see that there was a bit flipped and even though it does not know where, it can detect a failure and ask the sender to send the message again.

Example 2.12 (Parity-check code)**.** The code $C = \{(x_1, \ldots, x_n) \in \mathbb{F}^n \mid x_n = \sum_{i=1}^{n-1} x_i)\}$ is called the *parity-check code* of length n. The generator matrix of this code is $G = \begin{bmatrix} I_{n-1} | \mathbf{1} \end{bmatrix}$, *i.e.*,

$$G = \begin{bmatrix} 1 & 0 & \cdots & 0 & 1 \\ 0 & 1 & \cdots & 0 & 1 \\ \vdots & \vdots & \ddots & 0 & 1 \\ 0 & 0 & \ldots & 1 & 1 \end{bmatrix}.$$

△

Example 2.13 (Hamming code)**.** A more sophisticated code is the $[n, k]$ *Hamming code*. It encodes a message of k bits into a message of n bits, the rate is thus k/n. For the $[7, 4]$-Hamming code the encoding function is defined as

$$\text{enc}(m_1 m_2 m_3 m_4) = m_1 m_2 m_3 m_4 t_1 t_2 t_3,$$

with the so called parity bits

$$t_1 = m_1 \oplus m_2 \oplus m_3$$
$$t_2 = m_2 \oplus m_3 \oplus m_4$$
$$t_3 = m_1 \oplus m_3 \oplus m_4.$$

We can check that the minimal distance of this code is 3 and thus it can detect up to two errors and can solve one error. So if a codeword is corrupted in one bit, the $[7, 4]$-Hamming code can correctly decode it, we will see a decoding function for this code later. The parity-check matrix of this code is

$$H = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}. \tag{2.3}$$

△

Low-density parity-check (LDPC) codes, founded by Gallager in 1962 [2], are the main topic of this thesis. By referring to LDPC codes we actually refer to codes with a low-density representation.

Definition 2.14 (LDPC code)**.** A low-density parity-check (LDPC) code is a linear binary block code for which the parity-check matrix of interest has a low density of 1's.

The term 'low density' may be imprecise, but it means that the $n - k \times n$ matrix H only has $O(n)$ non-zero elements. Since a code can have multiple parity-check codes, we call it an LDPC code if there is at least one sparse representation. Gallager introduced *regular* LDPC codes in his work [2]. A (l, r)-regular LDPC code is defined by a parity-check matrix H where each column of H has weight l and each row has weight r. The class of irregular LDPC codes is more general, the rows and columns do not have to be of a constant weight and it includes regular codes. In Chapter 5 we will delve further into regular and irregular codes and their construction.

We have seen different representations of codes. We can represent a code by the set of its codewords, a generator matrix or a parity-check matrix. Recall that these last two are not unique. Another convenient way is to use a graphical representation. A *Tanner* graph is the graph representation of a parity-check matrix [12].

Definition 2.15 (Bipartite graph)**.** A graph $G(V, E)$ is *bipartite* if the set of nodes V can be partitioned into two classes, such that no edge connects two nodes from the same class.

Definition 2.16 (Tanner graph)**.** A *Tanner graph* for a code with a $n - k \times n$ parity-check matrix H is a bipartite graph $G(V_1 \cup V_2, E)$, such that:

- There are n so called *variable nodes* $v_1, \ldots, v_n$ in V_1
- There are $n - k$ so called *check nodes* $c_1, \ldots, c_{n-k}$ in V_2
- An edge connects a variable node v_i to a check node c_j if and only if $H_{ji} = 1$.

We often represent the variable nodes of a Tanner graph by circles and the check nodes by squares.

We have seen that the $[7, 4]$-Hamming code in Example 2.13 can be represented by the parity-check matrix given in (2.3). The Tanner graph for this $[7, 4]$-Hamming code based on this parity-check code can be found in Figure 4.

Recall that the encoding function of this code is

$$\text{enc}(m_1 m_2 m_3 m_4) = m_1 m_2 m_3 m_4 t_1 t_2 t_3.$$

We can look at the variable nodes $v_1, \ldots, v_7$ of the Tanner graph as the message bits $m_1, m_2, m_3, m_4, t_1, t_2, t_3$ and the check nodes, as the nodes that check if the requirements are met. In the Tanner graph, we see that first check node c_1 is connected to variable nodes v_1, v_2, v_3, v_5, which means that c_1 checks if

$$v_1 \oplus v_2 \oplus v_3 \oplus v_5 = 0,$$

which is equivalent to

$$t_1 = m_1 \oplus m_2 \oplus m_3.$$

The variable nodes of the Tanner graph consist of the message bits and any auxiliary bits that are used to protect the data. The check nodes tell us which equations must hold for these variable nodes. When a check node is connected to three variable nodes, it checks if the sum of these bits is even (and thus 0, because $\mathbb{F} = \mathbb{F}_2$).

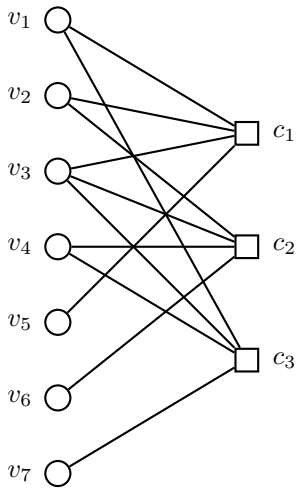


Figure 4: Tanner graph for the $[7, 4]$-Hamming code

Definition 2.17 (Cycle)**.** A *cycle* in a graph $G(V, E)$ is a sequence of distinct nodes that starts and ends in the same node.

Definition 2.18 (Girth)**.** The *girth* of a graph $G(V, E)$ is the length of the shortest cycle in the graph.

In the next chapters we will study these Tanner graphs more closely. In Chapter 3 we will see that this graphical representation plays a vital role in the decoding algorithm. Moreover, we will see what the part of cycles is in the decoding algorithm and that it is important that the girth is as big as possible. Obviously, the girth can only be an even number, since the Tanner graph is bipartite. It is also evident that the shortest possible cycle in any Tanner graph is four. The girth of the Tanner graph given in Figure 4 is four, since $(v_2, c_1, v_3, c_2, v_2)$ gives a cycle of length four. Observe that a code can have multiple representations and thus multiple Tanner graphs. Different representations can lead to different girths.

2.4 Encoding and Decoding of codes

Encoding

The encoding function is dependent of the code that is used. For linear codes there is a simple procedure for encoding. The encoding of non-linear however, can more be complex, although non-linear codes are usually defined by their encoding function. Encoding of a linear code can be done with the generator matrix G. We simply multiply our message m from the left side with G to obtain the corresponding codeword $c = mG$. In the case that the generator matrix is unknown, for instance for LDPC codes, we need to find this matrix to encode. If however, a parity-check matrix of the code is given, we can derive the corresponding generator matrix of this code. The problem is that the complexity of this procedure is rather high, the time to generate the generator matrix in a brute-force approach is $O(n^3)$, where n is the block size. There are alternative methods to encode, using properties of the code. As an example, the complexity of encoding of LDPC codes can be reduced due to their sparseness. In Chapter 3 we will discuss an algorithm, the belief propagation algorithm, to do this.

Decoding

Decoding is a relatively more complex issue. For decoding, we want to minimize the error probability. This is the probability that our decoding algorithm, with as input the received codeword $\overline{c}$ returns a message that is different than the original message m:

$$\mathbb{P}[\text{dec}(\overline{c}) \neq m].$$

Again, the decoding procedure of non-linear codes is more complicated than that of linear codes. For non-linear codes we need the definition of the code and since there is no structure we cannot really say anything about a decoding algorithm. For linear codes there are multiple methods known for decoding. One of the simplest is syndrome decoding, which uses the concept of the syndrome of a corrupted codeword.

Definition 2.19. Let $\mathcal{C}$ be a code with parity-check matrix H. The syndrome of a received codeword $\overline{c}$ is $H\overline{c}^T$.

The syndrome gives valuable information about if and where an error occurred. If the syndrome is $\mathbf{0}$, the output $\overline{c}$ is a codeword and thus with a high probability we can say that no error has occurred.
Assume now that the codeword c is send over a channel and only the ith bit is flipped. Then the output of the channel is $\overline{c} = c + e_i$ (where e_i is the ith unit vector). The syndrome of this output is

$$H\overline{c}^T = H(c + e_i)^T = Hc^T + He_i^T = He_i^T,$$

using that $Hc^T = \mathbf{0}$ for every codeword $c \in \mathcal{C}$.
He_i^T is equal to the ith column of H, meaning that if we receive the output $\overline{c}$, we can compute its syndrome and compare it to the parity-check matrix to find out which bit is most likely being flipped.

Example 2.20. Let $\mathcal{C}$ be a linear binary code with $n = 6, k = 3$ and let a generator matrix G and a parity-check matrix H given by

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}, H = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

Say we receive the vector $\overline{c} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 1 \end{bmatrix}$ from the channel, we first conclude that $H\overline{c}^T = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}^T \neq \mathbf{0}$ and thus we are sure that an error has been made. We see $He_2 = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}^T = H\overline{c}$, so this means that most likely the second bit is flipped and the sent messages is $c = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$. However, it could also have been the case that two bits have been flipped, e.g. bit 4 and bit 6. These columns add up to the same syndrome. If that would be the case, then the original codeword is not restored and our decoder algorithm failed. But, since it is generally less likely that two bits are flipped, the decoder selects the codeword assuming there was only one bit flip.

Now suppose we received $\overline{c} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$, the syndrome would be $H\overline{c} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}^T$ and we have multiple options for bits that have been flipped. Since there is no column in H with these values, at least two bits are flipped, but this can be either the first one and the fourth one, the second one and the fifth one or the third one and the sixth one. All with the same probability. So in this example, syndrome decoding does not help us with recovering the original message. △

We see that the syndrome decoding algorithm does not perform well in most cases, for this $[6, 3]$ code only in the case of a single bit flip. The question is if we can find another way to successfully decode a code? To try to answer this question, we define the maximum a posteriori (MAP) decoder.

Assume we use a channel with input space $X = \mathcal{C}$, output space Y and transition probability $\mathbb{P}_{Y|X}(y|x)$. The transmitter chooses a codeword $x \in \mathcal{C}$ with probability $\mathbb{P}_X(x)$ and transmits this codeword over the channel. Let y be the output of the channel. Our goal is to decide to which codeword y should be decoded. If we decode y to $\hat{x}(y)$, then the probability that this is not the originally codeword is $1 - \mathbb{P}_{X|Y}(\hat{x}(y)|y)$. We want to minimize this error probability and thus find the $\hat{x}(y)$ such that $\mathbb{P}_{X|Y}(\hat{x}(y)|y)$ is maximized.

This decoder $\hat{x}(y)$ is called the maximum a posteriori (MAP) decoder:

$$\hat{x}^{MAP}(y) = \arg\max_{x \in \mathcal{C}} \mathbb{P}_{X|Y}(x|y).$$

This MAP decoder minimizes the probability of (block) error, which we define as $P_e = \mathbb{P}\{\hat{x}^{MAP}(y) \neq x\}$.

An alternative way of decoding is with the maximum likelihood (ML) decoder. This decoder finds the $x \in \mathcal{C}$ for a given output $y \in Y$ that maximizes $\mathbb{P}_{Y|X}(y|x)$, *i.e.*,

$$\hat{x}^{ML}(y) = \arg\max_{x \in \mathcal{C}} \mathbb{P}_{Y|X}(y|x).$$

This decoder can be more convenient, since a channel is often defined by the $\mathbb{P}_{Y|X}$, but it does not always minimize the probability of error. The next proposition from Richardson and Urbanke [5] shows the relation between the MAP decoder and the ML decoder.

Proposition 2.21. *When all codewords are equally likely to be send, i.e., if $\mathbb{P}_X$ is uniform, then the MAP decoder and the ML decoder are equal.*

Proof. We write out the definition of the MAP decoder and use Bayes' theorem to see

$$\begin{aligned}
\hat{x}^{MAP}(y) &= \arg\max_{x \in \mathcal{C}} \mathbb{P}_{X|Y}(x|y) \\
&= \arg\max_{x \in \mathcal{C}} \mathbb{P}_{Y|X}(y|x) \frac{\mathbb{P}_X(x)}{\mathbb{P}_Y(y)} && \text{(Bayes' theorem)} \\
&= \arg\max_{x \in \mathcal{C}} \mathbb{P}_{Y|X}(y|x)\mathbb{P}_X(x) && (\mathbb{P}_Y(y) \text{ can be seen as a constant}) \\
&= \arg\max_{x \in \mathcal{C}} \mathbb{P}_{Y|X}(y|x) && (\mathbb{P}_X \text{ is uniform}) \\
&= \hat{x}^{ML}(y).
\end{aligned}$$

□

We can find the optimal decoder by searching through all codewords to look for the one that maximizes $\mathbb{P}_{X|Y}(x|y)$, but this takes exponential time, since there are $|\mathcal{C}| = 2^k$ codewords. Berlekamp, McElice and van Tilburg show that optimal decoding of linear block codes is generally an NP-complete problem [13]. However, this doesn't mean that there are no suitable decoding algorithms. There exist efficient algorithms for certain subclasses of codes. MAP decoding on the BEC can be for example solved in polynomial time in the length of the code [14]. For the repetition code on the BSC we can use the majority logic decoding algorithm below. In Chapter 4 we deal with MAP decoding for LDPC codes.

Majority logic decoding is a method to decode repetition codes. It decodes a received codeword by taking the majority vote of multiple candidate codewords to determine the most likely correct bit values. Suppose we are using the k-repetition code to send information over the BSC with parameter $p < 1/2$. Assume for simplicity that k is odd. If we want to transmit the message x then the input of the channel is the k-tuple $x, \ldots, x$. Let $y_1, \ldots, y_k$ be the output. It is easy to see that the MAP decoder then is defined by the majority rule

$$\hat{x}^{MAP}(y_1, \ldots, y_k) = \text{majority of } \{y_1, \ldots, y_k\}.$$

This gives the following probability of error

$$P_b = \mathbb{P}\{\hat{x}^{MAP}(y) \neq x\} = \mathbb{P}\{\text{at least } \lceil k/2 \rceil \text{ errors occur}\} = \sum_{i > k/2} \binom{k}{i} p^i (1-p)^{k-i}.$$

The MAP decoder depends strongly on the code and thus we should design an algorithm based on the specific code we want to use. As for general codes, the MAP decoding of an LDPC code on the BSC is an NP-complete problem [5]. In Chapter 4 of this thesis, we will discuss a decoding algorithm that has a performance close to the MAP decoder, but can be executed in polynomial time.

3 Encoding LDPC codes

In this chapter, we delve into the encoding of LDPC codes, a crucial process in the transmission of digital data over communication channels. Encoding is the process of converting a message into codewords, making it suitable for transmission over a noisy channel. While research on LDPC codes has primarily focused on decoding, encoding is equally, if not more, important when time complexity is considered. Unfortunately, for most LDPC codes, linear time encoding remains an unsolved problem to this day. The most significant contribution is that of Richardson and Urbanke [4] from 2001. They show that LDPC codes can be encoded in almost linear time, meaning that the constant of the quadratic term can be made quite small, so that for large block lengths encoding is still quite practical. For codes which allow transmission at rates close to capacity, linear time encoding is even possible [4].

There is a simple approach for encoding LDPC codes. A generator matrix can be obtained via the systemic form of the parity-check matrix H, as indicated in Section 2.1. This approach requires an encoding complexity of $O(n^2)$, since the generator matrix G obtained through Gaussian elimination will be in general not a sparse matrix, and thus multiplying with this matrix has complexity $O(n^2)$. We will present another method, where the codeword is generated directly from the parity-check matrix. Additionally, we discuss an algorithm suggested by Richardson and Urbanke [4], where the approximate upper triangular form of the parity-check matrix H is used, and we show that this reduces the complexity significantly.

3.1 Encoding via backward substitution

Considering that LDPC codes are usually defined by their parity-check matrix, a method that is using solely the parity-check matrix could be beneficial. By definition, all codewords satisfy

$$Hc^T = \mathbf{0}. \tag{3.1}$$

A straightforward way would be to bring H into an equivalent upper triangular form. We can split our codeword c of length n into a message part m of length k and a parity part p of length $n-k$, such that $c = (p, m)$. Matrix H can be similarly split into a square $n-k \times n-k$ upper triangular matrix H_p and a $n-k \times k$ matrix H_m. To encode a message m we fill the message part of the codeword with the desired information symbols m and determine the other $n-k$ symbols of the parity part p by using backward substitution. Since we work with the binary field, Equation 3.1 becomes

$$\begin{aligned}
\begin{bmatrix} H_p & H_m \end{bmatrix} \begin{bmatrix} p^T \\ m^T \end{bmatrix} &= \mathbf{0} \\
H_p p^T + H_m m^T &= \mathbf{0} \\
H_p p^T &= H_m m^T \\
p^T &= H_p^{-1} H_m m^T
\end{aligned}$$

Calculating the inverse of a square matrix is generally $O(n^3)$, but since H is in upper triangular form, we can use backward to solve this last equation. More exact, we can compute each p_i for all i, $1 \leq i \leq n-k$, with the following equation

$$p_i = \sum_{j=i+1}^{n-k} H_{i,j} p_j + \sum_{j=1}^{k} H_{i,j+n-k} m_j.$$

To analyse the complexity of this algorithm we can distinguish two parts of the encoding algorithm. First, we have the preprocessing step, where we can do all calculations that are not dependent of the message m we want to encode. In this construction the preprocessing step consists of bringing H into its equivalent upper triangular form. The complexity of this part is $O(n^3)$ since Gaussian elimination is used. The second part is the actual encoding of the message. This can be done with backward substitution and thus has a complexity of $O(n^2)$.

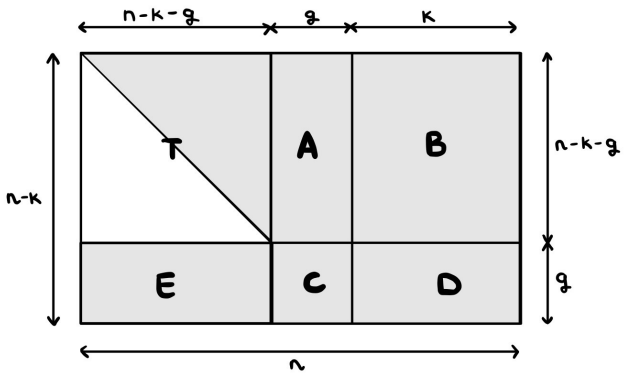
The preprocessing step is used only once, whereas the encoding step is used repeatedly for every message. Since H is no longer sparse after putting it is its equivalent upper triangular form, efficiency is lost and we

wonder if encoding can be accomplished in linear time. The algorithm of Richardson and Urbanke [4] does not use Gaussian elimination, but solely relies on row and column permutations to put H into an approximate triangular form. This preserves the sparseness of H and thus makes the encoding step more efficient.

3.2 Encoding via approximate lower triangular parity-check matrices

The method of Richardson and Urbanke [4] to encode LDPC codes consists of two steps. A preprocessing step and an encoding step. In the first step the parity-check matrix H is brought into an approximate upper triangular form using only row and column permutations. The second step is the actual encoding step, where we use the matrix of the previous step to encode a message.

In the first step row and column permutations are used to get H in the following form



where T is a square upper triangular matrix. The dimensions of the submatrices are as indicated. We call this form the *approximate upper triangular form*. Observe that H is still sparse, since we used only row and column permutations. The aim is to get T to get as large as possible, because if T has dimension $n-k \times n-k$ we have succeeded in finding an encoding algorithm with complexity $O(n)$, since we can use back substitution with a sparse matrix. However, in general this is not the case, and we need to proceed with the following steps. The height of the matrices E, C, D is called the gap g. This parameter measures the 'distance' of H from an upper triangular decomposition. We want to minimize the gap to get the dimensions of T as close as possible to $n-k$. An efficient algorithm to bring H in this form with a minimized gap can be found in [4]. If the gap is of size $O(\sqrt{n})$ then encoding can still be done in linear time [4].

Assuming that we have brought our H into this approximate upper triangular form, we can continue with the second part of this preprocessing step by multiplying H from the left side with the matrix

$$\begin{bmatrix} I & 0 \\ -ET^{-1} & I \end{bmatrix}$$

which yields

$$\begin{bmatrix} T & A & B \\ 0 & C - ET^{-1}A & D - ET^{-1}B \end{bmatrix}.$$

As in the first step of Gaussian elimination, we have successfully eliminated the matrix E by this premultiplication step. The codeword $c = (p, m)$ is further broken down into $c = (p_1, p_2, m)$, where p_1 is of length $n-k-g$ and p_2 of length g. Now the equation $Hx^T = 0^T$ can be split in two equations, namely

$$Tp_1^T + Ap_2^T + Bm^T = \mathbf{0}, \tag{3.2}$$

$$(C - ET^{-1}A)p_2^T + (D - ET^{-1}B)m^T = \mathbf{0}. \tag{3.3}$$

We define $\phi = C - ET^{-1}A$ and assume that ϕ is invertible. If not, we have to permutate columns of H such that ϕ becomes invertible, as explained by Richardson and Urbanke [4]. Equation 3.3 can then be turned into

$$p_2^T = \phi^{-1}(D - ET^{-1}B)m^T.$$

Hence, if we precompute the $g \times g$ matrix ϕ^{-1}, we can determine p_2 by solving this equation directly, following the steps as in Table 1. Most steps have complexity $O(n)$, because the matrices B, D, E, T are sparse and T is upper triangular. The last step has a complexity of $O(g^2)$, since ϕ is a $g \times g$ matrix and in general not sparse. The overall complexity for computing p_2 is thus $O(n + g^2)$.

We can find p_1 using the operations in Table 2. Each step has complexity $O(n)$, the matrices A and B are sparse (having $O(n)$ non-zero elements) so we can multiply with these matrices in linear time $O(n)$. Moreover, because T is sparse and upper triangular, we can use back substitution, also having a complexity of $O(n)$. Thus, with these steps we can compute p_1 with a total complexity of $O(n)$.

We conclude that the overall complexity of this algorithm is $O(n + g^2)$. In general, the coefficient of the quadratic term is typically small, which implies that encoding can still be manageable for large block lengths, even if it is not linear, as long as the value of g is sufficiently small. A summary of the two steps of the algorithm, both the preprocessing part and the encoding part can be found respectively in Algorithm 1 and Algorithm 2.

Operation	Comment	Complexity
Bm^T	Multiplication by sparse matrix	O(n)
$T^{-1}[Bm^T]$	$T^{-1}[Bm^T] = y^T \iff [Bm^T] = Ty^T$	O(n)
$-E[T^{-1}Bm^T]$	Multiplication by sparse matrix	O(n)
Dm^T	Multiplication by sparse matrix	O(n)
$[Dm^T] - [ET^{-1}Bm^T]$	Subtraction	O(n)
$-\phi^{-1}[Ds^T - ET^{-1}Bm^T]$	Multiplication by dense $g \times g$ matrix	O(g^2)

Table 1: Efficient computation of $p_2^T = -\phi^{-1}(Ds^T - ET^{-1}Bm^T)$ [4].

Operation	Comment	Complexity
Ap_2^T	Multiplication by sparse matrix	O(n)
Bm^T	Multiplication by sparse matrix	O(n)
$[Ap_2^T] + [Bm^T]$	Addition	O(n)
$-T^{-1}[Ap_2^T + Bm^T]$	$-T^{-1}[Ap_2^T + Bm^T] = y^T \iff -[Ap_2^T + Bm^T] = Ty^T$	O(n)

Table 2: Efficient computation of $p_1^T = -T^{-1}(Ap_2^T + Bm^T)$ [4].

Algorithm 1 Preprocessing step for encoding LDPC codes

Input: A non-singular parity-check matrix H

Output: An equivalent parity-check matrix of the form $\begin{bmatrix} T & A & B \\ E & C & D \end{bmatrix}$ such that $\phi = C - ET^{-1}A$ is non-singular and the matrix ϕ^{-1}.

1: Perform row and column permutations to bring the parity-check matrix H into approximate upper triangular form

$$H = \begin{bmatrix} T & A & B \\ E & C & D \end{bmatrix},$$

with as small a gap g as possible.

2: Use Gaussian elimination to perform the premultiplication

$$\begin{bmatrix} I & 0 \\ -ET^{-1} & I \end{bmatrix} \begin{bmatrix} T & A & B \\ E & C & D \end{bmatrix} = \begin{bmatrix} T & A & B \\ 0 & C - ET^{-1}A & D - ET^{-1}B \end{bmatrix}.$$

Check that $\phi = C - ET^{-1}A$ is non-singular - perform further column permutations if necessary to ensure this property.

Algorithm 2 Encoding step for encoding LDPC codes

Input: Parity-check matrix of the form $\begin{bmatrix} T & A & B \\ E & C & D \end{bmatrix}$ such that $\phi = C - ET^{-1}A$ is non-singular, ϕ^{-1} and a vector $m \in \mathbb{F}^k$.

Output: The vector $c = (p_1, p_2, m)$, $p_1 \in \mathbb{F}^{n-k-g}$ and $p_2 \in \mathbb{F}^g$, such that $Hx^T = \mathbf{0}^T$.

1: Determine p_1 as shown in Table 2.

2: Determine p_2 as shown in Table 1.

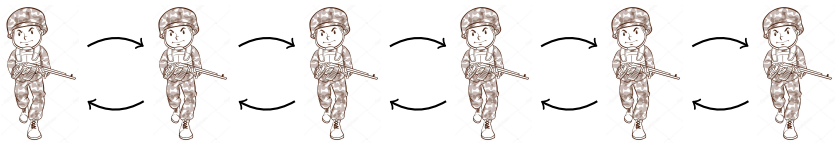
4 Decoding LDPC codes

When Gallager introduced LDPC codes in his work [2], he also suggested a decoding algorithm. Decoding LDPC codes is a critical process in the reliable and efficient transmission of digital data over communication channels. A decoder attempts to recover the original message from a corrupted codeword. Gallager's decoding algorithm is known under different names, depending on the context: the belief propagation algorithm, the sum-product algorithm, the message passing algorithm and many more. This last name is not really precise, since the algorithm is actually an instance of a message passing algorithm. We will refer to the algorithm as the belief propagation (BP) algorithm. This is an iterative algorithm that is used to compute approximate marginal probabilities for variables in a graphical model. When decoding LDPC codes, this graphical model is the Tanner graph of the code. The algorithm operates by passing messages between nodes of the graph, which represent the variables in the model. These messages contain a belief about what the values of the variable bits are, in the form of a probability.

Despite the fact that there are many decoding algorithms, the BP algorithm is often preferred [15]. It is relatively simple and in linear time it comes closest to an optimal decoder in terms of error correcting performance [16]. For that reason we stick solely to the BP algorithm in this chapter. We will start with presenting the main idea of messages passing algorithms on the basis of a simple counting problem. We then introduce the BP algorithm and try to optimize it.

4.1 Message Passing Algorithms

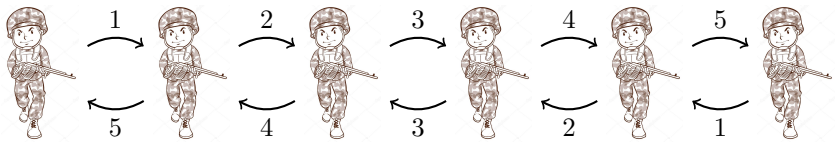
We will illustrate the main idea of message passing algorithms with an example. One of the simplest examples where a message passing algorithm can be used, is the soldier counting problem, where the goal is to count the number of soldiers in a group [17]. This is done by sending messages between themselves. Suppose the soldiers are standing on a line in dense fog, and they can only communicate with the persons directly next to them.



The following rules cause each soldier to know the total number of soldier in the row.

1. Firstly, if you are a soldier standing at one of the ends of the line, you say the number 1 to the soldier next to you.
2. Then, if a soldier next to you tells you a number, you add 1 to it and tell the soldier on the other side the new number.

With these rules, all soldiers end up knowing how many soldiers there are, as they can compute this number by adding 1 to the sum of the messages they received from their direct neighbours.



Now, suppose that the soldiers are not standing in a line, but in a structure as in the graph in Figure 5. The algorithm works similar, but we have to slightly adjust the rules, since now a soldier can have more than two neighbours.

1. The process starts at the *leaves* of the graph, these are the nodes with exactly one connected edge, where the soldiers with only one neighbour stand. These soldiers tell the number 1 to their only neighbour.

2. All soldiers count the number of neighbours N.

3. The soldiers keep track of the number of messages they have received from their neighbours. They denote the message they receive from neighbour i by v_i.

4. When they have received $N-1$ messages, they identify the neighbour who has not sent them a message yet and tell this neighbour the number which they get by adding 1 to the sum of the messages they got from the other neighbours.

5. If a soldier has received N messages, he knows that the total number of soldiers is $1+V$, where $V := \sum_{i=1}^{N} v_i$. He sends neighbour n the number $V+1-v_n$, if this is not done yet. After all, neighbour n already knows v_n, so you subtract it from the total number of soldiers before you send it.

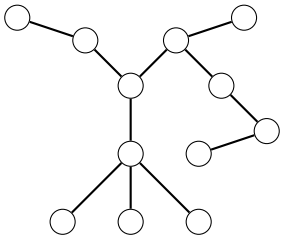


Figure 5: A graph that represents the structure of a group of soldiers

Figure 6 shows the different steps of this algorithm on the graph from Figure 5. The leaves are the only nodes that can send a message without receiving one, so the flow starts at the leaves and moves inward. The center node is the first to receive messages from all its neighbours. In the next iteration it sends messages to all its neighbours, reversing the flow of messages from the center towards the leaves. At the end of the process, every node has received messages from all its neighbours.

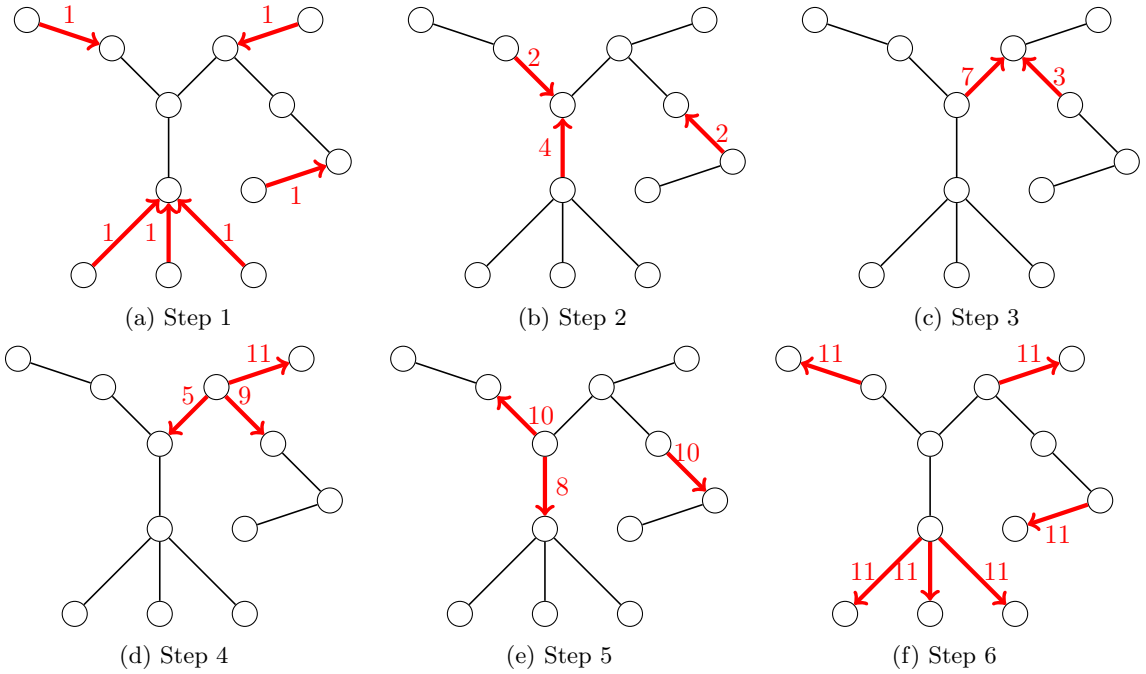


Figure 6: An example of a message passing algorithm for the soldier counting problem on a tree. After the last iterations all soldiers know there are, including themselves, 12 soldiers.

This algorithm sends two messages along each edge, so the total number of messages is twice the number of edges. Note that the graph in Figure 6 is a tree, since it does not contain any cycles. Recall that a tree with n nodes has $n-1$ edges, so in this case we need to send $2(n-1)$ messages when we want to count n soldiers. Furthermore, it is easy to see that the number of steps is equal to the *diameter* of the graph, which is defined as the length of the shortest path between the most distanced nodes.

Now for the last case, suppose that the soldiers are standing in the structure as indicated in Figure 7, where the graph contains a cycle. If we want to apply our algorithm on this graph we do not get the same required result. According to the rules as stated above, soldiers $3, 4, 6, 7, 8$ and 9 will never receive enough messages to send any messages. This is because they are in a cycle together. As soon as a message reaches the cycle, it cannot continue. Hence this algorithm fails on this graph and we conclude that is can only be successful on a tree. In the remainder of this chapter we will consider a specific message passing algorithm and explore the significance of cycles in graphs.

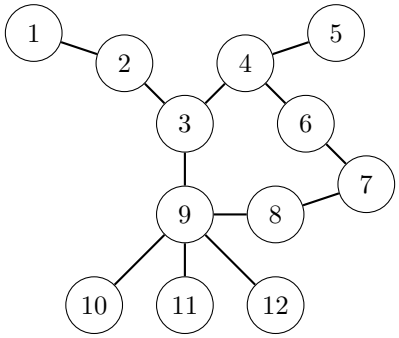


Figure 7: Graph with a cycle

4.2 Belief propagation

Belief propagation (BP) is an iterative decoding algorithm that uses the structure of the Tanner graph. In each iteration, messages are passed along the edges of the Tanner graph. First, variable nodes send a message to all the check nodes they are connected to, followed by the check nodes that send a message to their adjacent variable nodes. After receiving all messages, variable and check nodes undergo an updating process where they compute their new messages. Generally, these messages are different for every neighbour.

Each variable node in the Tanner graph represents one bit of the channel input x. We say that a variable node v_i 'is in state' 0 if x_i, the ith bit of x, equals 0. The decoding algorithm has as input the received corrupted codeword y and tries to recover all bits of x, *i.e.*, tries to identify the states of the variable nodes. The messages in the BP algorithm contain a 'belief' about what states of the variable nodes are. It often takes the form of a probability. Suppose that there are n possible states, then a message is a n-dimensional vector, where each entry contains the probability of being in that specific state. We need the entries to sum up to 1. Since we primarily look at binary fields in this thesis, we will for now assume that a message is a 2-dimensional vector. We denote the message that is send from variable node v_i to check node c_j by $u_{i\to j}$ and the message from check node c_j to variable node v_i by $w_{j\to i}$.

Instead of viewing the input in its binary representation 0 and 1, it can be more convenient to consider the two field elements as either 1 or -1. The field $\mathbb{F}_2$ is a commutative group under addition, and the set $\{-1, +1\}$ is a commutative group under multiplication. We use the standard mapping of 0 to 1 and 1 to -1. This notation is often more convenient, but we may switch between the two notations. The messages send over the edges are as following in this new notation

$$u_{i\to j} = [u_{i\to j}(+1), u_{i\to j}(-1)]$$
$$w_{j\to i} = [w_{j\to i}(+1), w_{j\to i}(-1)],$$

where $u_{i\to j}(x)$ represents the probability that $x_i = x$, given the channel output y and the information passed to v_i from all adjacent check nodes except c_j. We call this extrinsic information because the message from check node c_j to variable node v_i is excluded and is not used to calculate $u_{i\to j}$, as in the soldier counting problem. Similarly, $w_{j\to i}(x)$ denotes the probability of check node c_j being satisfied, given $x_i = x$ and the probability distributions of the other variable nodes connected to c_j as indicated by the messages they have sent to c_j.

The algorithm consist of four steps. First we initialize the algorithm, where the channel output is used. Next, we proceed in iterations, which contains two steps: a variable node update step and a check node update step. In each iteration all the check nodes process their incoming messages and compute their outgoing messages and, subsequently, the variable nodes process their incoming messages and compute their outgoing messages. Lastly, there is the termination step where we check if we have successfully completed the algorithm, and if not, we return to the second step, the variable node update step.

1. **Initialization**
   To initialize the algorithm, the channel output y is used to compute the messages from the variable nodes to the check nodes, since we have no other information to base it on. The input bits, and thus the output bits, are all independent and to compute $u_{i\to j}$ we only need y_i

   $$u_{i\to j} = [\mathbb{P}_{X|Y}(+1|y_i), \mathbb{P}_{X|Y}(-1|y_i)].$$

   Observe that these messages are independent of j. We set l_i equal to the channel information, $[\mathbb{P}_{X|Y}(+1|y_i), \mathbb{P}_{X|Y}(-1|y_i)]$ for every i. To conclude the initialization, the variable nodes broadcast their messages.

To determine the rules for the update steps, we will use the bit-wise maximum a posteriori (MAP) decoder

$$\hat{x_i}^{MAP}(y) = \underset{x_i\in\{\pm 1\}}{\arg\max}\, \mathbb{P}(x_i|y).$$

This decoder similar to the one we have seen in Section 2.4, but now we check which bit maximizes the probability. Under the assumption that the codewords are equiprobable, the rule reads

$$\begin{aligned}
\hat{x_i}^{MAP}(y) &= \underset{x_i\in\{\pm 1\}}{\arg\max}\, \mathbb{P}(x_i|y) \\
&= \underset{x_i\in\{\pm 1\}}{\arg\max} \sum_{\substack{z\in\mathbb{F}^n \\ z_i = x_i}} \mathbb{P}(z|y) \\
&= \underset{x_i\in\{\pm 1\}}{\arg\max} \sum_{\substack{z\in\mathbb{F}^n \\ z_i = x_i}} \mathbb{P}(y|z)\mathbb{P}(z) \\
&= \underset{x_i\in\{\pm 1\}}{\arg\max} \sum_{\substack{z\in\mathbb{F}^n \\ z_i = x_i}} \prod_j \mathbb{P}(y_j|z_j)\mathbb{1}_{x\in\mathcal{C}}. \qquad (4.1)
\end{aligned}$$

We often write $\sum_{\sim x_i}$ when taking the summation over all variables except x_i. A brute force approach can again be used for the optimal decoder, but this requires $O(2^n)$ operations, making calculations for big n impossible. So another, polynomial, method is used, where we assume the independence of the received messages.

2. **Check node update rule**
   A check node receives messages from the adjacent variable nodes, each containing probabilities about the state the variable nodes think they are in. The message sent from a check node c_j to a variable node v_i indicates the probability that this check node is satisfied. The message $w_{j\to i}(x)$ contains this probability that check node c_j is fulfilled given that v_j is in state x. Hence using the independence assumption and (4.1) we see that this message is the product of all feasible options of the incoming messages, except the message from the variable node it is sending a message to.

Recall that $N(v)$ represents the neighbourhood of a node v in a graph, *i.e.*, it contains all nodes which are connected to v by an edge. Let d be the degree the check node. The rule states

$$w_{j\to i}(x) = \sum_{\sim x} f_i(x, x_1, \ldots, x_d) \prod_{\substack{v_h \in N(c_j) \\ h \neq i}} u_{h\to j}(x_h),$$

where

$$f_i(x, x_1, \ldots, x_d) = \begin{cases} 1 & \text{if } x_1 \cdots x_{i-1} x_{i+1} \cdots x_d = x \\ 0 & \text{otherwise,} \end{cases}$$

tells us which options are feasible. These options ensure that the parity check constraint is not violated.

We want the probabilities in each message to add up to 1 to keep the probabilities normalized. Since $\sum_{\sim x} f_i(-1, x_1, \ldots, x_d) + \sum_{\sim x} f_i(+1, x_1, \ldots, x_d) = 1$ this property will be retained. An example of this rule is illustrated in Figure 8.

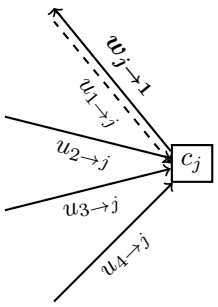


Figure 8: Check node c_j receives messages from its adjacent variable nodes. Message $w_{j\to 1}$ is sent from c_j to v_1. The message $u_{1\to j}$ from variable node v_1 is not used in the calculation of message $w_{j\to 1}$.

In Figure 8 we see that check node c_j is connected to variable nodes v_1, v_2, v_3 and v_4. The message $w_{j\to 1}$ from c_j to v_1 can be computed as

$$\begin{aligned} w_{j\to 1}(+1) &= u_{2\to j}(+1)u_{3\to j}(+1)u_{4\to j}(+1) + u_{2\to j}(+1)u_{3\to j}(-1)u_{4\to j}(-1) \\ &\quad + u_{2\to j}(-1)u_{3\to j}(+1)u_{4\to j}(-1) + u_{2\to j}(-1)u_{3\to j}(-1)u_{4\to j}(+1) \\ w_{j\to 1}(-1) &= u_{2\to j}(-1)u_{3\to j}(-1)u_{4\to j}(-1) + u_{2\to j}(-1)u_{3\to j}(+1)u_{4\to j}(+1) \\ &\quad + u_{2\to j}(+1)u_{3\to j}(-1)u_{4\to j}(+1) + u_{2\to j}(+1)u_{3\to j}(+1)u_{4\to j}(-1). \end{aligned}$$

3. **Variable node update rule**
   The messages sent by variable nodes contain a belief of the state the variable nodes are in. Assuming that the messages the variable node receives from its adjacent check nodes are independent, we can simply multiply the probabilities of the incoming messages

$$\mathbb{P}_{X|Y}(x|y_1, \ldots, y_n) = \mathbb{P}_{X|Y}(x|y_1) \cdots \mathbb{P}_{X|Y}(x|y_n).$$

   The message sent from a variable node v_i on an edge to a check node c_j is the product of probabilities of the incoming messages, including the channel output and without the message received from the check node c_j, *i.e.*,

$$u_{i\to j}(x) \propto l_i \prod_{\substack{c_k \in N(v_i), \\ k \neq j}} w_{k\to i}(x).$$

   We use the proportion symbol $\propto$ to show that we still need to scale this probability to make sure that $u_{i\to j}(+1) + u_{i\to j}(-1) = 1$. Figure 9 displays an example of this rule.

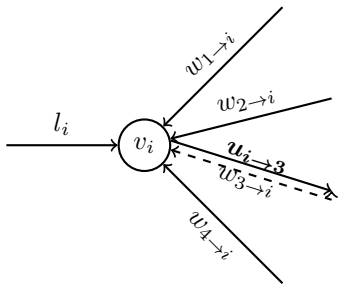


Figure 9: Suppose a variable node v_i is connected to check nodes c_1, c_2, c_3 and c_4. The message $u_{i\to 3}$ from variable node v_i to check node c_3 is computed by $u_{i\to 3}(+1) = l_i(+1)w_{1\to i}(+1)w_{2\to i}(+1)w_{4\to i}(+1)$ and $u_{i\to 3}(-1) = l_i(-1)w_{1\to i}(-1)w_{2\to i}(-1)w_{4\to i}(-1)$.

4. **Termination**
   After each iteration, containing two update steps, we check if the stopping criteria has been met, *i.e.*, if we restored the codeword. To achieve this, we look at the current 'belief' of the codeword x. The belief b_i of a variable node v_i is a two dimensional vector which contains the probabilities $b_i(+1)$ and $b_i(-1)$, where
   $$b_i(x) \propto l_i \prod_{c_j \in N(v_i)} w_{j\to i}(x),$$
   such that the entries of b_i add up to 1. We terminate the algorithm if we have found a codeword, that is when the codeword x satisfies $Hx^T = \mathbf{0}$, or if we have reached a maximum number of iterations. As long as this maximum is not reached, we check after each iteration if our current guess $\hat{x}$ is a codeword. Let $\hat{x}_i$ be our guess for x_i, defined by the state with the highest probability, *i.e.*,
   $$\hat{x_i} = \begin{cases} +1 & \text{if } b_i(+1) \geq 0.5, \\ -1 & \text{otherwise.} \end{cases}$$
   We can check if $\hat{x} = [\hat{x}_1, \ldots, \hat{x}_n]$ is a codeword by testing if the equation $H\hat{x}^T = \mathbf{0}$ holds. If $\hat{x}$ satisfies this condition we have found our codeword and our decoding algorithm is terminated. If not, the algorithm continues with another iteration.

4.3 Log-likelihood ratio

In the BP algorithm above, messages are vectors of length two. However, since we require that the entries sum up to 1, it would be enough to send only one entry. So instead of the whole message $u_{i\to j} = [u_{i\to j}(+1), u_{i\to j}(-1)]$ we could just send $u_{i\to j}(+1)$, $u_{i\to j}(-1)$, the difference between the two or simply the ratio. Gallager used the last option, or to be more specific, the logarithm of the ratio, to further simplify the BP algorithm [2].

Definition 4.1. We define the log-likelihood ratio (LLR) of a binary random variable X as
$$L(X) := \ln\left(\frac{\mathbb{P}_X(+1)}{\mathbb{P}_X(-1)}\right).$$

Similar, the channel-transition LLR, where X is the binary channel input and Y the channel output is defined as
$$L(Y|X) := \ln\left(\frac{\mathbb{P}_{Y|X}(y|+1)}{\mathbb{P}_{Y|X}(y|-1)}\right).$$

We have observed that in the BP algorithm, the messages need to be normalized after each updating step to keep it a normalized probability distribution. The LLR can optimize the algorithm because it makes this step

unnecessary. Furthermore, because of the properties of the logarithm, multiplications turn into additions and thus the complexity is reduced. Hence, using LLRs is a convenient way of sending messages.

Observe that when the probability distribution of X is uniform, *i.e.*, when $\mathbb{P}_X(+1) = \mathbb{P}_X(-1) = \frac{1}{2}$, the LLR of X will be 0. Moreover, we see that if $\mathbb{P}_X(+1) > \mathbb{P}_X(-1)$, then $L(X) > 0$ and if $\mathbb{P}_X(+1) < \mathbb{P}_X(-1)$, then $L(X) < 0$. Hence, we see that the sign of a LLR indicates the most probable binary value

$$\operatorname{sign}(L(X)) = \underset{b\in\{-1,+1\}}{\arg\max}\ \mathbb{P}_X(b).$$

The magnitude of the LLR of X, $|L(X)|$, indicates how reliable the decision on the binary value based on the sign is.

To recover probabilities from a LLR $L(X)$, an inversion function is needed. It is simple to verify that the following functions accomplish this.

$$\begin{aligned}\mathbb{P}_X(+1) &= \frac{e^{L(X)}}{1+e^{L(X)}}\\ \mathbb{P}_X(-1) &= \frac{1}{1+e^{L(X)}}.\end{aligned} \tag{4.2}$$

With the concept of the LLR we can update the algorithm to make it more efficient. For the variable nodes we use the following computation to see that the update rule simply consist of taking the sum of the received messages, excluding the check node we are sending the message to.

$$\begin{aligned}L(x_i|y) &= \ln\left(\frac{\mathbb{P}_X(+1|y_1,\dots,y_n)}{\mathbb{P}_X(-1|y_1\cdots y_n)}\right) = \ln\left(\frac{\mathbb{P}_X(+1|y_1)\cdots\mathbb{P}_X(+1|y_n)}{\mathbb{P}_X(-1|y_1)\cdots\mathbb{P}_X(-1|y_n)}\right) = \ln\left(\frac{\mathbb{P}_X(+1|y_1)}{\mathbb{P}_X(-1|y_1)}\cdots\frac{\mathbb{P}_X(+1|y_n)}{\mathbb{P}_X(-1|y_n)}\right)\\ &= \ln\left(\frac{\mathbb{P}_X(+1|y_1)}{\mathbb{P}_X(-1|y_1)}\right) + \cdots + \ln\left(\frac{\mathbb{P}_X(+1|y_n)}{\mathbb{P}_X(-1|y_n)}\right)\end{aligned}$$

The new rules for the initialization step and the terminal step are very straightforward. However, the changes in the update rule of the check nodes are less self-evident, but later on we will give a justification how this is originated. The optimized algorithm consists of the following steps:

1. **Initialization step**

$$\begin{aligned}L(u_{i\to j}) &= \tilde{L}_i,\\ \tilde{L}_i &:= \ln\left(\frac{\mathbb{P}_X(+1|y_i)}{\mathbb{P}_X(-1|y_i)}\right).\end{aligned}$$

2. **Variable node update rule**

$$L(u_{i\to j}) = \tilde{L}_i + \sum_{\substack{c_k\in N(v_i),\\ k\neq j}} L(w_{k\to i}).$$

3. **Check node update rule**

$$L(w_{j\to i}) = 2\tanh^{-1}\left(\prod_{\substack{c_k\in N(c_j)\\ k\neq j}} \tanh\left(\frac{L(u_{h\to j})}{2}\right)\right)$$

4. **Termination** After each iteration we compute the LLR of the total belief by

$$L(b_i) = \tilde{L}_i + \sum_{c_j\in N(v_i)} L(w_{j\to i}).$$

Then, we let the guess x_i for the ith bit of x be

$$\hat{x_i} = \begin{cases} +1 & \text{if } L(b_i) \geq 0, \\ -1 & \text{otherwise.} \end{cases}$$

Now we need to check if $H\hat{x}^T = \mathbf{0}$. If so, we have found a codeword and we can terminate the algorithm. Otherwise we continue with iterations, consisting of steps 2 and 3, until we have found a codeword or the maximum number of iterations is reached.

The following lemma from Gallager [2] will help use clarify the check node update rule.

Lemma 4.2. *Consider a sequence of m independent binary digits in which the lth digit is a 1 with probability P_l. The the probability that an even number of digits are 1 is*

$$\frac{1 + \prod_{l=1}^{m}(1 - 2P_l)}{2}.$$

Proof. Consider the function

$$f(t) = \prod_{l=1}^{m}(1 - P_l + P_l t).$$

Expanding this function into a polynomial gives

$$\begin{aligned} f(t) = &(1 - P_1) \cdots (1 - P_m) \\ &+ [P_1(1 - P_2) \cdots (1 - P_m) + (1 - P_1)P_2(1 - P_3) \cdots (1 - P_m) + \cdots + (1 - P_1)(1 - P_2) \cdots (1 - P_{m-1})P_m]\, t \\ &+ [P_1 P_2(1 - P_3) \cdots (1 - P_m) + \ldots]t^2 + \ldots. \end{aligned}$$

Observe that the coefficient of t^i is the probability of having i 1's. The function

$$g(t) = \prod_{l=1}^{m}(1 - P_l - P_l t)$$

is identical to $f(t)$ except for the odd powers of t as those are negative. Therefore, adding $f(t)$ and $g(t)$ will cancel the odd powers out and double the even terms. Hence $\frac{1}{2}(f(1) + g(1))$ equals the probability of having an even number of 1's. And because

$$\frac{1}{2}\left(f(1) + g(1)\right) = \frac{\prod_{l=1}^{m}(1 - P_l + P_l) + \prod_{l=1}^{m}(1 - P_l - P_l)}{2} = \frac{1 + \prod_{l=1}^{m}(1 - 2P_l)}{2}$$

we got the required result. □

This lemma uses binary digits, *i.e.*, bits, so the results are in $\mathbb{F}_2$. When we translate this to the field where we use $\{-1, +1\}$ the lemma gives us the probability of an even number of -1's in a sequence. Furthermore, it tells us the probability that the product of the elements in the sequence equals 1.

We can translate the requirement for the check nodes to the field $\{-1, +1\}$ to see that the following equations are equivalent

$$\begin{aligned} x_1 \oplus \cdots \oplus x_n &= 0 \\ x_1 \cdots x_n &= 1 \\ x_1 &= x_2 \cdots x_n. \end{aligned}$$

Thus, if we want to find the probability that x_i equals -1, we can look at the probability that the product of the others equals -1. Therefore

$$\begin{aligned}
\mathbb{P}_X(x_i = -1) &= \mathbb{P}_X\left(\prod_{j\neq i} x_j = -1\right) \\
&= 1 - \mathbb{P}_X\left(\prod_{j\neq i} x_j = 1\right) \\
&= 1 - \mathbb{P}(\text{There are an even number of } x_j\text{'s equal to -1 }) \\
&= 1 - \frac{1 + \prod_{j\neq i}(1 - 2\mathbb{P}_X(x_j = 1))}{2} && \text{(Gallager's lemma)} \\
&= \frac{1 - \prod_{j\neq i}(1 - 2\mathbb{P}_X(x_j = 1))}{2}.
\end{aligned}$$

Rewriting this statement gives us

$$1 - 2\mathbb{P}_X(x_i = -1) = \prod_{j\neq i} 1 - 2\mathbb{P}_X(x_j = 1). \tag{4.3}$$

Now, using the inversion function of (4.2) we can express the this probability using the following notion

$$1 - 2\mathbb{P}_X(-1) = 1 - \frac{1}{1 + e^{L(X)}} = \frac{e^{L(X)} - 1}{e^{L(X)} + 1} = \tanh\left(\frac{L(X)}{2}\right). \tag{4.4}$$

To conclude, combining (4.3) and (4.4) gives

$$\begin{aligned}
\tanh\left(\frac{L(x_i)}{2}\right) &= \prod_{j\neq i} \tanh\left(\frac{x_j}{2}\right) \\
L(x_i) &= 2\tanh^{-1}\left(\prod_{j\neq i} \tanh\left(\frac{x_j}{2}\right)\right),
\end{aligned}$$

explaining the rule.

4.4 Belief Propagation on the BEC

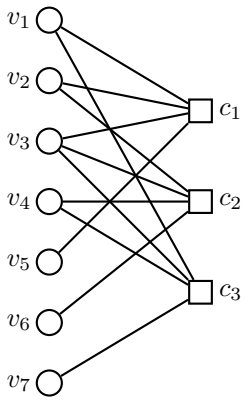
In this section we will explore the Belief Propagation (BP) algorithm on the Binary Erasure Channel (BEC). Recall that the BEC is an erasure channel, it models situations where information may be lost, but is never corrupted. We can use the BP algorithm of Section 4.3 to decode codes that are send over the BEC. Since this channel is excessively simplified, the algorithm can be described with some simple rules. We can use the LLRs as messages, but since the LLR can only be equal to $-\infty, \infty$ or 0, it can be done more efficient. Instead messages simply contain a 0, a 1 or the erasure symbol '?'. We furthermore investigate in this section when this decoding algorithm is exact and which problems can occur.

We simplify the rules from Section 4.2 for the BEC and obtain the following rules

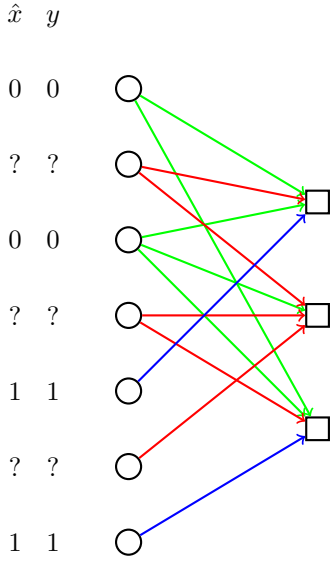
1. **Initialise**
   Initialize all n variable nodes with the received value y_i.
   Each variable node broadcast their value over all connected edges to the check nodes.
2. **Check nodes updates**
   Check nodes compute the messages for each of their neighbours. If all incoming edges from the other nodes are not erased, the check node computes the outgoing message on this edge as the sum of all other incoming messages. Otherwise the check node declares the message as an erasure and sends the erasure symbol.
3. **Variable nodes updates**
   The variable nodes compute their messages for each neighbour. If all incoming edges from the other neighbours are erased, output erasure. Otherwise broadcast the received non-erased value to this edge.
4. **Termination**
   If all variable nodes are recovered, stop the algorithm. Otherwise go to step 3.

It is clear that if this algorithm returns a codeword, it is the correct one. We will illustrate the algorithm using the $[7, 4]$-Hamming code from Example 2.13.

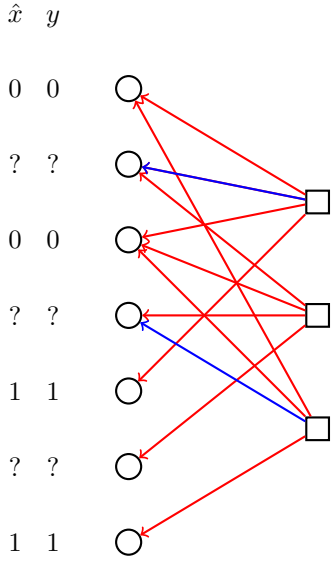
Example 4.3. Let C be the $[7, 4]$-Hamming code and assume we use the BEC. The Tanner graph associated with this code can be found below. Suppose we receive the message $y = \begin{bmatrix} 0 & ? & 0 & ? & 1 & ? & 1 \end{bmatrix}$ from the channel. We want to decode this message and find the codeword $\hat{x}$ such that $\mathbb{P}_{Y|X}(y|\hat{x})$ is maximized. Following the steps from the algorithm above, Figure 10 shows how the algorithm operates on this decoding problem. To simplify matters, we denote the messages over the edges containing a 0 by a green line, containing a 1 by a blue line and the erasure messages are red. We see that the algorithm terminates with $\hat{x} = \begin{bmatrix} 0 & \mathbf{1} & 0 & \mathbf{1} & 1 & \mathbf{0} & 1 \end{bmatrix}$. It turns out that $\mathbb{P}_{Y|X}(y|\hat{x}) = 1$ for this $\hat{x}$ so we know we restored the original message.



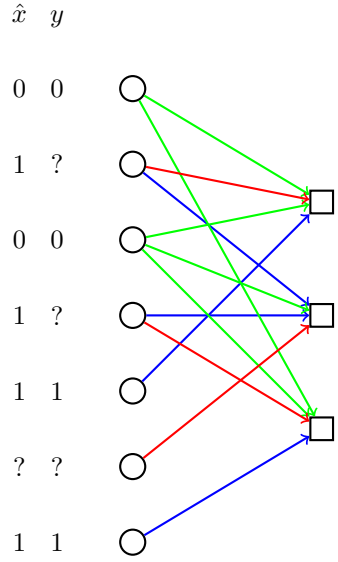
△



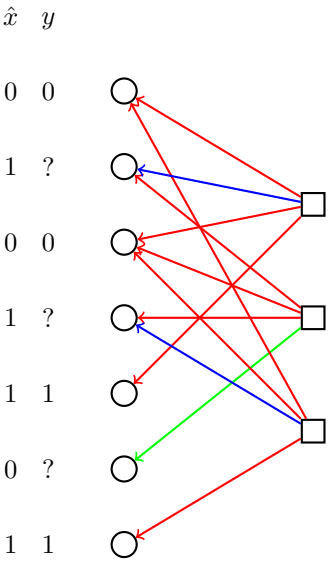
(a) Step 1: Initialize all variable nodes with the received value. Each variable node broadcast their value



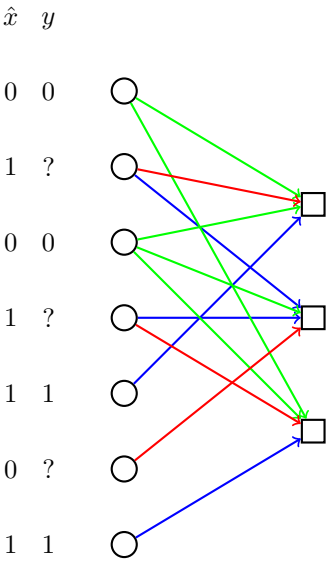
(b) Step 2: Process check nodes. We see that the only non-erasure messages can be send from c_1 to v_2 and from c_3 to v_4. These equal the sum of the other incoming messages of the check node.



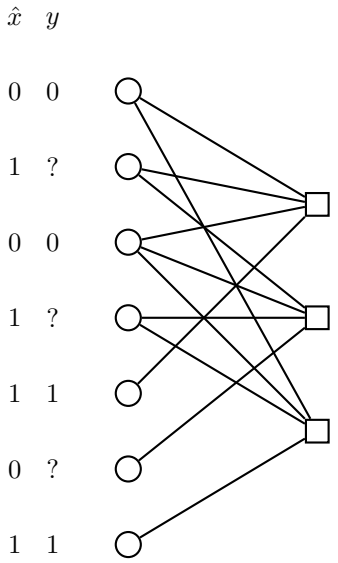
(c) Step 3: Process variable nodes. Variable nodes v_2 and v_4 now know there value and can send it to their other neighbours. Not all variable nodes are recovered, so we go back to step 2.



(d) Step 2: Check nodes compute their messages and send them back.



(e) Step 3: Now all variable nodes are recovered.



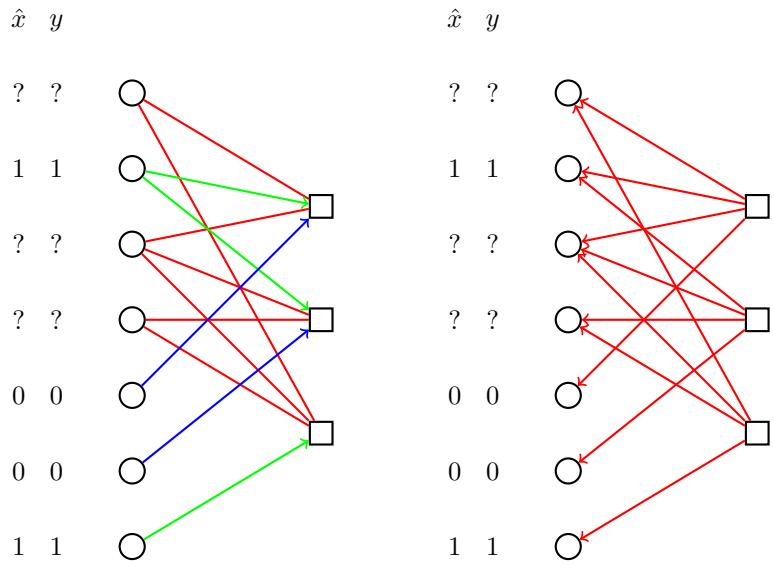
(f) Step 4: All variable nodes are recovered and we check that indeed $H\hat{x}^T = \mathbf{0}$.

Figure 10: Step by step Belief Propagation decoding algorithm of the $[7, 4]$-Hamming code on the BEC.

Example 4.4. Let C be the $[7, 4]$-Hamming code used over the BEC again. Now, suppose that some other values are erased and we receive $y = \begin{bmatrix} ? & 1 & ? & ? & 0 & 0 & 1 \end{bmatrix}$. We follow the same steps as described in the algorithm, as shown in Figure 11. However, in step 2, when the check nodes compute their messages, we see that the only messages that are sent from the check nodes are erasure messages. This is because after the first step each check node has at least two incoming error messages. The algorithm does not fail to find a solution because there is none. There is an unique solution for this decoding problem and it can be found by solving the following equations

$$\begin{aligned} v_1 \oplus v_3 &= 1 \\ v_1 \oplus v_3 \oplus v_4 &= 1 \\ v_3 \oplus v_4 &= 1. \end{aligned}$$

From this set of equations we can see that $v_1 = 0, v_3 = 1, v_4 = 0$ is an unique solution. This indicates that the massage passing decoding algorithm cannot always find the optimal solution. △



(a) Step 1: Initialization. Variable nodes broadcast their value.

(b) Step 2: Check nodes compute their messages, here all messages are erasure messages.

Figure 11: An example of an erasure decoding problem where the algorithm is ineffective.

The reason why the algorithm cannot continue is because each check node is connected to at least two variable nodes whose value is erased. To have a better understanding when the algorithm succeeds, we introduce stopping sets.

Definition 4.5 (Stopping set)**.** Let $G(H)$ be a Tanner graph and V be the set of variable nodes in $G(H)$. A stopping set S is a subset of V, such that all neighbours of S are connected to S at least twice.

In Figure 11 we see that $S = \{v_1, v_2, v_4\}$ is a stopping set. Observe that from the definition we can conclude that the empty set and the support set (the indices of the non-zero elements) of any codeword are stopping sets. However, a stopping set need not be the support of a codeword. For example, $\{v_2, v_3, v_4\}$ is a stopping set, but $[0, 1, 1, 1, 0, 0, 0]$ which has as support $\{v_2, v_3, v_4\}$ is not a codeword. It is not hard to see that if S_1 and S_2 are both stopping sets, then so is $S_1 \cup S_2$. This means that each subset of V contains an unique maximum stopping set, which might be the empty set. The significance of stopping sets in the iterative decoding of LDPC codes over the BEC is highlighted in the following lemma [7], with an adjusted proof.

Lemma 4.6. *Let G be a generator matrix for an LDPC code over the BEC and let $\mathcal{E}$ denote the initial subset of V which is erased by the channel after the transmission of a message. Then the set of erasures which remain when the decoder stops is equal to the maximum stopping set contained in $\mathcal{E}$.*

Proof. Let S be the set of erasures which remain when the decoder stops. First, we need to show that S is actually a stopping set, which can be achieved by assuming the exact opposite. In that case, there is a check node, a neighbour of the set S, which is connected to S exactly once. But this means that we can use this check node to recover the variable node it is connected to and this contradicts the fact that the decoder stops with this set. Therefore S is a stopping set.
To show that it is the maximum stopping set, we first note that there is no difference between a maximal stopping set and a maximum stopping set, since we have seen that the union of two stopping sets is again a stopping set. Let us assume that there is a superset $T \supseteq S$ in the set of erasures that is the maximum stopping set. Since there is no way for the decoder to determine the variable nodes contained in T, each neighbor of T has at least two connections to T according to the definition, and hence the decoder stops at T. Therefore, this means that $S = T$ and the set of erasures which remain when the decoder stops is equal to the maximum stopping set contained in $\mathcal{E}$. □

Stopping sets are called like this because the decoder must stop the algorithm when having to restore these sets. The presence of stopping sets determines the performance of a code over the BEC. Especially small stopping sets influence the performance of LDPC codes [18], since it is more likely that all bits in the stopping set are erased. Stopping sets illustrate the suboptimality of the BP decoder. A MAP decoder will fail if and only if the set of erasures include the support set of a codeword, whereas the BP decoder is obstructed by a bigger class, namely stopping sets. Every decoder will be unsuccessfully if the support of a codeword is erased. However, since the minimal size of the support set of codewords is the minimum weight of all non-zero codewords, it equals the minimum distance of the code. We can thus assume that for good codes these stopping sets are quite large and thus do not have a large probability of being completely erased. However, stopping sets can be in general very small and have high probability of full erasure. Thus, in order to generate a good code, small stopping sets should be avoided to reduce the change of a decoding failure.

There exist several procedures for constructing codes without small stopping sets. This can either be done during the construction of the code by avoiding small stopping sets, or by optimizing the code after it is generated by removing small stopping sets. One attempt of removing stopping sets is adding redundant rows to the parity-check matrix [19]. This is equivalent with adding check nodes to the Tanner graph and connecting them with variable nodes. These redundant rows can for example be low weight redundant rows that eliminate a large number of small-sized stopping sets. The addition of these redundant rows are computationally inexpensive, but the code rate of the resulting LDPC codes will be reduced, since the number of rows in the parity-check matrix increases. However, it cannot guarantee that stopping sets will be disrupted.

Other approaches include using constructing techniques that prevent the formation of stopping sets, such as progressive edge growth (PEG) [20]. Moreover, graph covers can be used to remove stopping sets from existing constructions [21]. In 2007, Rosnes and Ytrehus [22] introduced an exhaustive tree search algorithm to find low weight stopping sets of LDPC codes. This algorithm can be used to remove stopping sets in existing codes.

While it is important to avoid stopping sets, it is not reasonable, and more importantly, necessary, to avoid or remove every stopping set in a code [19]. Optimizing codes by removing all stopping sets can lead to impractical and complex code designs. The previously discussed methods to avoid stopping sets come at cost of the other properties of the code, such as the code length, the density of the code and even the capability of correcting errors [23]. Milenkovic, Soljanin and Whiting found that many stopping sets include small cycles. Hence, the probability of stopping sets is small in graphs without small cycles [24].

4.5 Codes with cycles

The Belief Propagation (BP) decoder is a MAP decoder for codes whose Tanner graph does not contain cycles. This means that this decoder is exact on trees, *i.e.*, graphs without cycles. This follows from the independence assumption that was made during the construction of the algorithm, where we assumed the independence of the neighbours of a node in the Tanner graph. Therefore, the probabilities could be separated

$$\mathbb{P}(x|y_1,\ldots,y_n) = \mathbb{P}(x|y_1)\cdots\mathbb{P}(x|y_n).$$

Since the neighbours of a node v in a tree are only connected via a path through v, this node v can assume that the information it gets from its different neighbours is independent. This exactness of the BP decoder can also be proved using factor graphs, as is done in [5].

From this point of view, it would be advantageous to construct codes without cycles. For these codes we have a simple decoding algorithm that always finds the unique solution, if it exists. However, Etzion, Trachtenberg and Vardy [25] show that binary codes with cycle-free Tanner graphs necessarily have a small minimum distance. The next proposition, adjusted from [5], shows that for codes with a rate above a half, there are a significant number of codewords of weight 2. This means that the minimum distance of this code is 2 and hence these codes will have a higher error probability.

Proposition 4.7. *Let C be a binary linear code of length n, dimension k and rate $r > \frac{1}{2}$ that admits a binary Tanner graph that is a forest. Then C contains at least $(2r-1)n$ codewords of weight 2.*

Proof. The Tanner graph has n variable nodes and $n-k = n-rn = (1-r)n$ check nodes. The total number of nodes in V is thus $n+(1-r)n = (2-r)n$. We know that a tree has exactly $|V|-1$ nodes and thus this Tanner graph has strictly less than $|V| = (2-r)n$ edges. Since each edge is connected to exactly one variable node, we have that the average degree d_{av} of the variable node is less than $2-r$. We call a node an internal node when it is not a leaf node, so each internal variable node has degree ≥ 2. Suppose we have x variable leaf nodes, then there are $n-x$ internal variable nodes and we can use the average degree to see that

$$(2-r)n > nd_{av} \geq 2\cdot(n-x) + 1\cdot x = 2n-x,$$

and thus $x > nr$, which means that there are at least nr variable nodes that are leaf nodes.

We now ask ourselves how many pairs of nr leaf variable nodes can be made such that they are connected to the same check node. These pairs can generate a codeword of weight 2 by setting these variable nodes to 1 and the others to 0. We claim there are $(2r-1)n$ such pairs of variable leaf nodes. We consider two cases. First we assume that every check node is connected to at least one leaf variable node. Since $r > 1/2$ we have more leaf variable nodes than check nodes. The remaining $nr-(1-r)n = (2r-1)n$ leaf nodes must be connected to a check node that is already connected to a leaf node, so for every of these remaining leaf variable nodes we can make a pair with another leaf variable node. Therefore there are $(2r-1)n$ such pairs.

In the second case we assume there are $a \geq 1$ check nodes that are not connected to a leaf variable node. We use the same argument as in the first case, now saying that there are $(1-r)n-a$ check nodes connected to leaf variable nodes. Thus, after counting one variable node per check node, we still have $nr-((1-r)n-a) = (2r-1)n+a$ leaf variable nodes. These are all connected to a check node that is already connected to a leaf variable node so we can make at least $(2r-1)n+a$ pairs.

In both cases C contains at least $(2r-1)$ codewords of weight 2.

□

This proposition only covered the case of codes with $r > 1/2$. For $r > 2/3$ Kozlik [26] shows that the bound for the number of codewords of weight 2 can be extended to $\frac{rn(2r-1)}{2(1-r)}$. For the more complicated case where $r \leq 1/2$, Etzion, Trachtenberg, and Vardy show that the minimum distance is upper bounded by $2/r$ [25].

Despite the fact that the BP decoding algorithm works optimal on codes with cycle-free Tanner graphs, the class of binary codes that satisfy this property is not powerful enough to perform well. This implies that we need at least some cycles in the Tanner graph of our code. For smaller cycles, the dependence of the neighbours is bigger than for larger cycles, since the nodes are connected with a shorter path. Codes whose Tanner graph has no small cycles generally have a better performance [8]. The bigger the dependence, the more problems can arise. When the girth is large, the estimates for the values of the variable node are less dependent on the node's contribution. Since creating Tanner graphs with a large girth also reduces the number of stopping sets, we will analyse codes with large girth. We can adjust our BP decoder so that we can also us it on Tanner graphs with cycles.

Applying the BP algorithm, as it is, on general graphs with loops, leads to several problems. However, we can slightly adjust the algorithm and still use the, now called loopy, BP algorithm (LBP). The first problem that we encounter is that the initial messages as defined before are not enough to start the whole process. As seen in Section 4.1 the nodes in the cycle need extra information to send messages. LBP solves this by starting the process with random initial messages and iterate according to the regular BP updating rules. The aim is to reach a fixed point where the beliefs of the variable node lead to a codeword. The second problem is that we assumed earlier in the algorithm that neighbours of a node are independent. However, for nodes in a cycle we can no longer assume this. This problem is simply ignored in the LBP problem and we pretend as if the messages of the neighbours are independent. This could lead to invalid results, but in practice this is often not the case for graphs with a large girth. These graphs locally look like a tree and thus the problem of dependency is limited [27]. In the next chapter we consider the construction of Tanner graphs with a large girth.

5 Construction of Tanner graphs for LDPC codes

The focus of this chapter will be on constructing LDPC codes. There is not 'one' method of constructing these codes as they can be created using various methods. The primary goals in code constructions are to achieve high decoding performance and to have an easy hardware implementation, meaning that it is essential that the encoding and decoding must be simple to implement. To construct a code we need to know its parameters, such as row and column weights, rate, girth and code length. The designing process of the code determines these parameters.

One way of constructing LDPC codes is via a graph-based construction. In this constructing is first the Tanner graph specified, and the the parity-check matrix is constructed based on this graph. Another method uses a matrix-based construction, where the code is directly constructed via the parity-check matrix. The choice of the construction method depends on the specific requirements and design criteria of the application. In this chapter we will also focus on the problem of deciding the existence of a graph for some given parameters as it is unknown whether there exist graphs with a specific set of parameters. This problem is exploded using graphicality tests.

5.1 Regular LDPC codes with a large girth

Chapter 4 discusses why graphs with a high girth are favorable. In this chapter we look into constructing these graphs without small cycles. We begin with addressing the construction of regular LDPC codes. Recall that regular LDPC codes have a fixed column and row weight in their parity-check matrix. Constructing regular LDPC codes can be done randomly, with unstructured connections between nodes in the Tanner graph or rows and columns in the parity-check matrix. LDPC codes can also be constructed using a more structured method, where there is a predefined way of connecting. Random constructions offer flexibility in design and construction and often have a good performance, but are harder to optimize and decoding can be difficult. On the other hand, structured constructions have a simple decoding algorithm and improved performance, but are less flexible because of their complex design. For both constructions we will delve into some of the methods for constructing graphs with a large girth.

Many algorithms for random construction are known. Note that the randomness in these algorithms is often simulated by a computer. Therefore, the connections are actually pseudo-random. With the introduction of LDPC codes, Gallager [2] also provided a construction of regular LDPC codes which is now called the 'Gallager construction'. He constructed a (l, r)-regular LDPC code of length n and dimension k by creating the $n - k \times n$ parity-check matrix H that consists of l smaller $\frac{n-k}{l} \times n$ submatrices $H_1, \ldots, H_l$. Each row of the submatrices contains l 1's and each column a single 1. The matrix H_1 contains in the ith row 1's in columns $(i-1)l+1$ to il, for $1 \leq i \leq (n-k)/l$. The rest of the matrix consist of 0's. $H_2, \ldots, H_l$ are generated by taking pseudo-random column permutations of H_1.

$$H = \begin{bmatrix} H_1 \\ H_2 \\ \vdots \\ H_l \end{bmatrix}$$

There are no known methods yet to ensure that the code does not contain cycles of length 4, but computer simulations can find good permutations to construct codes with a reliable performance.

Another popular type of randomized construction is the so-called 'Mackay construction' [28]. MacKay developed random construction methods to produce sparse regular (l, r)-LDPC codes. Some of these algorithms are listed below.

- Matrix H is generated by starting from an all-zero matrix and randomly flipping bits, not necessarily distinct.
- Matrix H is generated by randomly creating weight l columns.

- Matrix H is generated with weight l per column and uniform weight per row, and no two columns having overlap greater than 1.
- Matrix H is further constrained so that its bipartite graph has large girth.

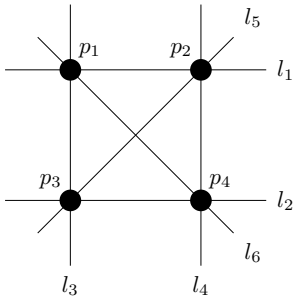
These algorithms were used to find codes of different length and rates with a desirable performance. They are also used as a basis for other constructing algorithms.

An example where the code is constructed via the Tanner graph is the bit-filling algorithm, first mentioned in [29]. This algorithm consist of different iterations, where in each iteration a variable node is added to a Tanner graph and connected to check nodes by adding new edges to the Tanner graph. These edges must not create any cycles of length $g-2$ or smaller. A test to enforce this constraint is given in [29]. While the algorithm is capable of generating codes with a high rate and a high girth, the resulting code may be not be easily implementable in hardware due to the inconsistent structure of row-column connections. More advanced algorithms, like the progressive edge-growth algorithm (PEG) [20] and quasi-cyclic construction [30] are also regularly used for constructing graphs with a large girth and are often a combination of randomized and structured constructions.

Sequential constructions are more structured, and often preferred over randomized constructions, since the performance of randomized constructions may vary depending on the chosen parameters and constraints, and the resulting codes may be harder to optimize for specific applications. Many methods for constructing structured LDPC codes can be found in the literature.

Bayati gives a method for randomly generating simple graphs without small cycles [31]. This was the first polynomial algorithm for this problem. Given any constants $k, \alpha \leq 1/2k(k+3)$ and $m = O(n^{1+\alpha})$, the algorithm generates an asymptotically uniform random graph with n nodes, m edges and girth larger than k. It does this by starting with an empty graph of n nodes and adding edges between the nodes one by one. It samples from the set of all edges according to a probability distribution described in the article and it checks if adding this edges leads to a small cycle. If this happens, the algorithm samples another edge and tries this one. If there is no edge that satisfies this constraint, the algorithm stops and returns a failure. This challenge lies in designing the probability distribution such that a graph is chosen uniformly at random.

Also other topics within mathematics can be used for constructing LDPC code. A construction via finite geometry is presented in [32]. A finite geometry is defined by n points and J lines with the following properties: (1) Every line consists of ρ points; (2) Every point is intersected by γ lines; (3) two lines are either parallel (*i.e.*, they have no point in common) or they intersect at one and only one point. A $J \times n$ matrix can be formed where the rows represent the J lines and the columns the n points of the finite geometry. A 1 in the matrix represents the intersection of a line and a point. If ρ and γ are small compared to n and J, the matrix can be regarded as a LDPC matrix. The parameters γ, ρ are then the column and row weights of the regular LDPC code. Property (3) ensures that there are no cycles of length 4, since this would mean that two different lines intersect two points. An example:



(a) Finite geometry with $J = 6$, $n = 4$, $\rho = 2$ and $\gamma = 3$.

$$\begin{array}{c} \\ l_1 \\ l_2 \\ l_3 \\ l_4 \\ l_5 \\ l_6 \end{array} \begin{array}{c} \begin{array}{cccc} p_1 & p_2 & p_3 & p_4 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

(b) Corresponding parity-check matrix

Unfortunately, it turns out that these codes always have girth 6 which is a limitation, since cycles of length six are still considered as small cycles.

These methods are only a small fraction of the attempts that are known for the construction of graphs with a large girth. For the remainder of the thesis we will focus on constructing LDPC codes via their Tanner graphs.
The constructions above are all constructions for regular LDPC codes. The Tanner graph of a regular LDPC code is a *biregular* graph. This is a bipartite graph $G = (V_1 \cup V_2, E)$ for which every two nodes on the same side of the given partition have the same degree as each other. The parity-check matrix of a regular (l, r)-LDPC code has l 1's in each column and r 1's in each row. This means that the Tanner graph is biregular and all the variable nodes have degree l and the check nodes degree r.

5.2 Irregular LDPC codes

After Gallager shared his work about LDPC codes, many efforts have been made to improve the performance of these LDPC codes. One remarkable contribution is the introduction of the irregular LDPC codes by Luby *et al.* in 1998 [9]. These codes allow irregular variable and check node degrees in their Tanner graph. Additionally, they show in their article that the performance of irregular codes are typically better than of regular LDPC codes. Irregular LDPC codes perform close to the Shannon capacity [33].

The following terminology and notation of irregular LDPC codes and their irregular degree distribution is adopted from [9]. For convenience, we represent the degrees of the nodes in the Tanner graph of an irregular code in the following compact way. We display the *degree distributions* of both the variable and the check nodes. Let Λ_i be the number of variable nodes of degree i and P_i the number of check nodes of degree i. The degree distributions of a code then is

$$\Lambda(x) = \sum_{i=1}^{l_{max}} \Lambda_i x^i, \qquad P(x) = \sum_{i=1}^{r_{max}} P_i x^i.$$

We call this code a $(\Lambda(x), P(X))$-irregular LDPC code.

Example 5.1. The degree distributions of the $[7, 4, 3]$-Hamming code from Example 2.13 are

$$\Lambda(x) = x^3 + 3x^2 + 3x, \qquad P(x) = 3x^4$$

△

Note that $\Lambda(1)$ is the number of variable nodes in the Tanner graph and thus the length of the code. $P(1)$ denotes the number of check nodes. The code rate can thus be expressed as

$$r = 1 - \frac{\Lambda(1)}{P(1)}.$$

The number of edges can easily be expressed, since there are $i\Lambda_i$ edges attached to all variable nodes of degree i. The number of edges equals

$$\sum_i i\Lambda_i = \Lambda'(1), \qquad \text{or equivalent} \qquad \sum_i iP_i = P'(1).$$

For some applications it can be more convenient to use the normalized degree distribution. This gives the fraction of nodes that have a certain degree, instead of the exact number of nodes with this degree.

$$L(x) = \frac{\Lambda(x)}{\Lambda(1)}, \qquad R(x) = \frac{P(x)}{P(1)}.$$

Example 5.2. The normalized degree distributions of the $[7, 4, 3]$ Hamming code from Example 2.13 are

$$L(x) = \frac{1}{7}x^3 + \frac{3}{7}x^2 + \frac{3}{7}x, \qquad R(x) = x^4$$

△

The ensemble of LDPC codes with degree distributions $\Lambda(x)$ and $P(x)$ is often depicted as LDPC$(\Lambda(x), P(X))$.

Good degree distributions are non-trivial to find. These distributions will be determined during the design of the code and depend on the application. In 2001, a powerful code design, which is based on the degree distribution in a Tanner graph, was represented by Richardson, Shokrollahi and Urbanke [34]. Various techniques have been proposed to design good degree distribution. Richardson *et al.* [34] used density evolution to optimize the degree distribution. These concepts are beyond the scope of this thesis. Instead, we will turn to the problem of constructing Tanner graphs for codes with given degree distributions.

5.3 Degree sequences

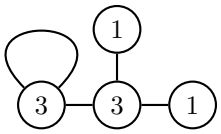
In the last section we have shown that the performance of LDPC codes can be improved by using irregular codes. Irregular codes have degree distributions which regulate the degrees of the variable and check nodes in the Tanner graph of the code. In the following sections we discuss how we construct Tanner graphs for irregular codes, given their degree distributions. We try to construct Tanner graphs which have a large girth. We start with generalizing this problem. We look at degree sequences and try to find conditions on the existence of a graph with this degree sequence.

Definition 5.3 (Degree sequence)**.** A sequence $\boldsymbol{d} = (d_1, \ldots, d_n)$ of non-negative integers is called a *degree sequences* of a graph G if the nodes can be labeled $v_1, \ldots, v_n$ such that $\deg(v_i) = d_i$ for each i with $1 \leq i \leq n$.

For convenience, we always order a degree sequences and write $(d_1, \ldots, d_n)$ such that $d_1 \leq \cdots \leq d_n$. We may also assume that in general all degrees are non-zero, since nodes with degree 0 are isolated so they can be ignored.

Definition 5.4 (Graphical degree sequence)**.** A degree sequence $\boldsymbol{d} = (d_1, \ldots, d_n)$ of non-negative integers is called *graphical* if it is the degree sequence for some graph. We say that a graph G *realizes* $\boldsymbol{d}$ if $\boldsymbol{d}$ is the degree sequence of G.

If $\boldsymbol{d}$ is a graphical degree sequence, it clearly holds that all degrees are smaller than n, *i.e.*, $d_i \leq n-1$. Moreover, the sum of the degrees $\sum_{i=1}^{n} d_i$ must be even. The question is if these conditions are enough. It turns out that for each degree sequence that satisfies these two conditions, there exists a graph that realizes this sequence. However, this graph can have loops or multiple edges between two nodes. Since we want to construct Tanner graphs without self-loops and double edges, we restrict ourselves to simple graphs. These are unweighted, undirected graphs containing no loops and no multiple edges. The question now becomes more difficult. For example, the sequence $\boldsymbol{d} = (3, 3, 1, 1)$ can be realized by the following graph, but there exists no simple graphs that realizes this sequence.



The question if there exists a simple graph G for a given degree sequence is more complicated. Therefore, from now on, the term *graphical degree sequences* is used for sequences that can be realized by a simple graph, but not by any graph. Fortunately, there are theorems that tells us necessary and sufficient conditions. The first one is a result independently found by both Havel [35] and Hakimi [36]. This theorem is later used by Erdős and Gallai to come up with a set of inequalities that have to be checked to see if a degree sequence is graphical or not [37].

Theorem 5.5 (Havel-Hakimi)**.** *A sequence of non-negative integers* $(d_1, \ldots, d_n)$ *with* $d_1 \geq d_2 \geq \cdots \geq d_n$ *and* $n \geq 3$ *is graphical if and only if the sequence*

$$(d_2 - 1, d_3 - 1, \ldots, d_{d_1} - 1, d_{d_1+1} - 1, d_{d_1+2} \ldots, d_n),$$

i.e., the sequence we get when we delete the first element and then subtract 1 from the first d_1 *elements, is graphical.*

Theorem 5.6 (Erdős-Gallai)**.** *A sequence of positive integers $(d_1, \ldots, d_n)$ with $d_1 \geq d_2 \geq \cdots \geq d_n$ is graphical if and only if $\sum_{i=1}^{n} d_i$ is even and for each integer k with $1 \leq k \leq n$ the following inequalities hold*

$$\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(k, d_i).$$

This last theorem gives us a method where we have to check n inequalities. Tripathi and Vijay [38] show that this number of inequalities can be even further reduced. Let s be the largest integer such that $d_s \geq s$. Then it is enough to check the inequalities for every k with $1 \leq k \leq s$. Furthermore, they prove that in case of multiple occurrences of numbers in the degree sequence, it suffices to check the inequality in Theorem 5.6 only at the end of each segment of duplicate values.

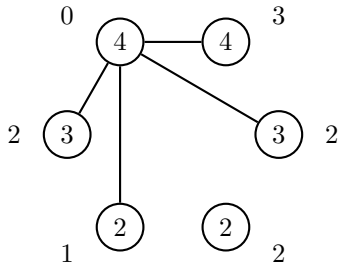
In some cases, there are more ways to see that a sequence is not graphical. We can for example directly see that the sequence $\boldsymbol{d} = (6, 6, 5, 4, 3, 3, 1)$ is not graphical, since $n = 7$, $d_1 = 6$ and $d_2 = 6$. This means that v_1 and v_2 must be connected to all other nodes, which contradicts with $d_7 = 1$. In general the Erdős-Gallai theorem and the Havel-Hakimi theorem are the most common, and easiest, ways to test the validity of the graphicality.

The next problem we encounter is the construction of graphs that realize graphical degree sequences. Theorem 5.5 can be used to design an algorithm to construct such a graph. This algorithm can be described with the following steps.

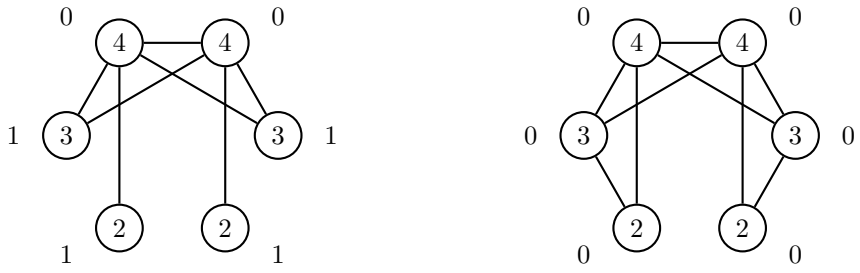
1. Pick the node with the highest degree Δ.
2. Connect this node to the next Δ nodes having highest degree. This node now has been exhausted. We now look at the truncated, residual sequence which we get after removing the first element and subtract 1 from all nodes it is attached to.
3. From this truncated sequence we again pick the node with the highest degree and connect it to the next the nodes with the highest residual degree. Repeat these steps until all nodes are exhausted.

Theorem 5.5 directly proves the correctness of this algorithm. If the algorithm fails in a step and cannot connect a node to the required other nodes, it means that the current truncated degree sequence is not graphical. Using several rounds of the Havel-Hakimi theorem, this would mean that the original sequence is not graphical, which gives a contradiction.

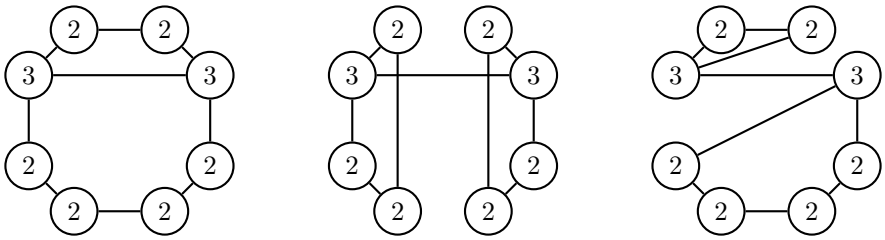
Example 5.7. We can use the algorithm to check if the degree sequence $\boldsymbol{d} = (4, 4, 3, 3, 2, 2)$ is graphical. And if so, the algorithm can find a graph G that realizes $\boldsymbol{d}$. We start with an empty graph G with 6 nodes. The first step is to connect the node of degree 4 with the four nodes with the highest degree, in this case degrees 4, 3, 3 and 2.



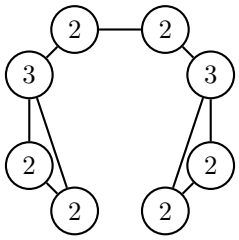
The residual degree sequence is now, after reordering, $(3, 2, 2, 2, 1)$. This means that in the next step we connect the node with residual degree 3 to the three nodes with residual degree 2. This results in the graph on the left. The residual degree sequence after this step is $(1, 1, 1, 1)$, hence in the last two steps we connect two nodes of residual degree 1 to each other. This can for example lead to the final graph on the right which has $\boldsymbol{d}$ as degree sequence. △



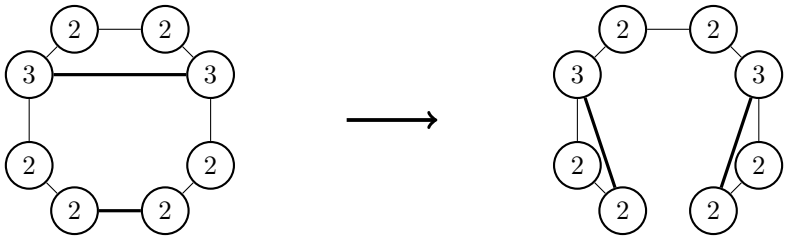
There are typically many different realizations for one graphical degree sequences. The sequence $(3, 3, 2, 2, 2, 2, 2, 2)$ is graphical. The following graphs are examples of graphs, found using the Havel-Hakimi algorithm, that realize this sequence.



The algorithm can return multiple graphs that are not equivalent. Unfortunately, the algorithm is unable to return every graph, as it will never give the next graph as output, because there is no edge between the two nodes of highest degree 3.



The question that arises now is how we can make all graphs realizing a graphical degree sequence $\boldsymbol{d}$. This can be done by swapping edges in a graph. Using this method we can turn a graph that can be constructed by the Havel-Hakimi algorithm to one that does not. For this edge swapping step we pick two distinct edges (u, v) and (u', v') and we swap these edges with the edges (u, u') and (v, v') or (u, v') and (v, u'). Note that the degree sequence of the graph remains unchanged.



This idea of edge swapping is also used in the proof of the Havel-Hakimi theorem. An important study by Petersen shows that if we have two graphs G_1 and G_2 with the same degree sequence, there exist a sequence of edge swaps to turn the one in the other [39]. He also showed that the number of swaps is $O(\sum_i d_i)$. This theorem helps us to study properties of graphs realizing a certain degree sequence. Starting at a realization

of a degree sequence and randomly swapping edges will generate a random graph with the same degree sequence. By generating numerous random graphs and performing statistical analysis on the outcomes we can answer questions like 'What is the average diameter of a graph with this degree sequence?' or 'What is the average chromatic number of a graph with this degree sequences'. Despite the fact that it does not sample a graph uniformly at random (it samples a graph with probability proportional to the number of possible edge swaps), we can use the known bias to make the sampling more uniform.

For graphical degree sequences with an unique realization it would be senseless to do this edge swapping process, since there is only one graph that realizes the sequence. Hence, it is interesting to study the characteristics of the sequences which have an unique realization. Koren [40] found that sequences that lie on the convex polytope formed by the inequalities of Theorem 5.6 are exactly the sequences that have an unique realization. Moreover, Li proves in [41] that if G realizes a sequence $\boldsymbol{d}$, G is an unique realization if and only if for all nodes $v \in V$ any node adjacent to v has a higher degree than any node that is not adjacent to v.

5.4 The bipartite degree realization problem

In order to construct LDPC codes, it is necessary to obtain Tanner graphs. These graphs need to have a bipartite structure. The *bipartite realization problem* asks whether there exist a bipartite graph that has a specific degree sequence, and if so, how to construct it. The general problem, where both the partition and the realizing graph need to be determined, is still not resolved [42]. However, for the purpose of constructing Tanner graphs for LDPC codes, we focus on the the simpler variant of the problem. This variant, where the partition is given, has been answered over 60 years ago by Gale and Ryser [43] [44]. The Gale-Ryser theorem can be used to solve the bipartite degree realization problem in polynomial time. Let $\boldsymbol{a}$ and $\boldsymbol{b}$ be non-negative integer sequences, then the sequence pair $(\boldsymbol{a}, \boldsymbol{b})$ is said to be *bigraphic* if there exist some simple bipartite graph $G = (V_1 \cup V_2, E)$ so that $\boldsymbol{a}$ represents the degrees of the nodes in V_1 and $\boldsymbol{b}$ the degree of nodes in V_2. We again say that this simple bipartite graph G *realizes* the pair $(\boldsymbol{a}, \boldsymbol{b})$. Gale and Ryser gave the following necessary and sufficient conditions for a sequence pair to be bigraphic.

Theorem 5.8 (Gale-Ryser)**.** *A pair of sequences of non-negative integers $(a_1, \ldots, a_p)$ and $(b_1, \ldots, b_q)$ with $a_1 \geq \cdots \geq a_p$ is bigraphic if and only if $\sum_i^p a_i = \sum_i^q b_i$ and the following inequality holds for k with $1 \leq k \leq p$:*

$$\sum_{i=1}^{k} a_i \leq \sum_{i=1}^{q} \min(b_i, k).$$

Note that only sequence $\boldsymbol{a}$ is ordered, it is not necessary to order $\boldsymbol{b}$ as well.

For the construction of bipartite graphs realizing a given bigraphic sequence pair, we propose an algorithm that is based on the Havel-Hakimi algorithm. We will give a justification of the algorithm as well by proving the following algorithm. Note that the roles of $\boldsymbol{a}$ and $\boldsymbol{b}$ can be reversed.

Theorem 5.9. *The pair of sequences $(\boldsymbol{a}, \boldsymbol{b})$ is bigraphic if and only if $(\boldsymbol{a'}, \boldsymbol{b'})$ is bigraphic, where $(\boldsymbol{a'}, \boldsymbol{b'})$ is obtained from $(\boldsymbol{a}, \boldsymbol{b})$ by deleting the largest element Δ from $\boldsymbol{b}$ and subtracting 1 from each of the Δ largest elements of $\boldsymbol{a}$.*
In other words, when a pair of sequences of non-negative integers $(a_1, \ldots, a_p)$ and $(b_1, \ldots, b_q)$ is ordered, so that $a_1 \geq \cdots \geq a_p$ and $b_1 \geq \cdots \geq b_q$, then $(\boldsymbol{a}, \boldsymbol{b})$ is bigraphic if and only if the sequence pair $(\boldsymbol{a'}, \boldsymbol{b'})$ with

$$\boldsymbol{a'} = (a_1 - 1, \ldots, a_\Delta - 1, a_{\Delta+1}, \ldots, a_p) \qquad \boldsymbol{b'} = (b_2, \ldots, b_q),$$

is bigraphic.

Proof. We start with the sufficiency condition. Given a bigraphic pair of sequences $(\boldsymbol{a'}, \boldsymbol{b'})$, let $G' = (V_1 \cup V_2, E)$ be a simple bipartite graph, such that $\boldsymbol{a}$ contains the degrees of V_1 and $\boldsymbol{b}$ the degrees of V_2. We construct the bipartite G the realizes $(\boldsymbol{a}, \boldsymbol{b})$ by adding a node to V_2 that is adjacent to the nodes in V_1 that have degree $a_1 - 1, \ldots, a_\Delta - 1$, as these are the Δ largest numbers in $\boldsymbol{a'}$. This bipartite graph G has $\boldsymbol{a}, \boldsymbol{b}$ as

degree sequences. Hence, the pair $(\boldsymbol{a}, \boldsymbol{b})$ is bigraphic.

To prove necessity, we start with a simple bipartite graph $G = (V_1 \cup V_2, E)$ that realizes the pair $(\boldsymbol{a}, \boldsymbol{b})$ and we produce a simple bipartite graph $G' = (V_1' \cup V_2', E)$ realizing the pair $(\boldsymbol{a'}, \boldsymbol{b'})$. We may assume that both $\boldsymbol{a}$ and $\boldsymbol{b}$ have been arranged in non-increasing order. Let v be a node of degree Δ in V_2 and let $X \subseteq V_1$ be a set of Δ nodes in V_1 having the desired degrees $a_1, \ldots, a_\Delta$. If the neighbours of v are exactly the nodes in X, *i.e.*, if $N(v) = X$, then we can delete v to obtain G'. Otherwise, there is a node in X that is not in $N(v)$. Our approach is to adjust G to increase $|N(v) \cap X|$, and this is achieved without changing any node degrees. By swapping edges we try to obtain $N(v) = X$. We choose $u \in X$ and $w \notin X$ so that $u \notin N(v)$ and $w \in N(v)$. Since $d(u) \geq d(w)$, there must be a node x adjacent to u, but not to w. After swapping the edges (v, w) and (u, x) with the edges (u, v) and (w, x) node w is now in both $N(v)$ and x and thus this step leads to an increase of $|N(v) \cap X|$. The size of $N(v)$ and X is both equal to Δ, so this step is repeated until $N(v) = X$.

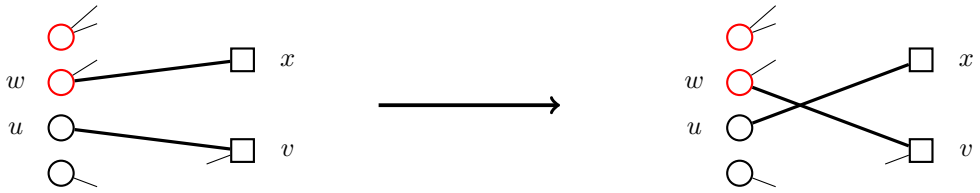


Figure 13: A part of a simple bipartite graph is showed to illustrate the edge swap procedure in the proof. The nodes in X are indicated in red. We find nodes u and w, so that u is a neighbour of v but not in X and w in X but not a neighbour of v. We detect node x as the neighbour in V_2 that is adjacent to w but not to u. We now swap the edges as indicated. Node u is replaced by w in $N(v)$ and thus $|N(v) \cap X|$ is increased. We continue until $N(v) = X$.

□

Example 5.10. Let $(\boldsymbol{a}, \boldsymbol{b})$ be a pair of sequences, with $\boldsymbol{a} = (2, 2, 1, 1, 1)$ and $\boldsymbol{b} = (3, 2, 2)$. The theorem states that the pair $(\boldsymbol{a}, \boldsymbol{b})$ is bigraphic if and only if $(\boldsymbol{a'}, \boldsymbol{b'})$ is graphical, where $\boldsymbol{a'} = (1, 1, 1, 1, 0)$ and $\boldsymbol{b'} = (2, 2)$. The pair $(\boldsymbol{a}, \boldsymbol{b})$ is bigraphic. Let $G = (V_1 \cup V_2, E)$ be a bipartite graph that realizes this sequence pair, with $V_1 = \{v_1, v_2, v_3, v_4, v_5\}$, $V_2 = \{c_1, c_2, c_3\}$ and $E = \{(v_1, c_3), (v_2, c_1), (v_2, c_2), (v_3, c_3), (v_4, c_1), (v_4, c_2), (v_5, c_3)\}$ as in the first graph of Figure 14. This bipartite graph realizes $(\boldsymbol{a}, \boldsymbol{b})$. To show that $(\boldsymbol{a'}, \boldsymbol{b'})$ is also bigraphic, a bipartite graph G' is constructed as in the proof of Theorem 5.9.

Node c_3 is the only check node of degree $\Delta = 3$ and $X = \{v_2, v_3, v_4\}$ is the set of variable nodes with degrees $2, 2, 1$, the Δ largest numbers in V_1. These nodes are indicated by red circles. It can be directly seen that $X \neq N(v)$, so some edge swaps need to be executed. We see that $v_2 \in X$ is not a neighbour of c_3, while $v_1 \notin X$ is. Thus we are going to swap edges attached to these nodes. Because $d(v_2) = 2 \geq 1 = d(v_1)$ we find check node c_1 which is adjacent to v_2, but not to v_1. Now we swap the edges (v_1, c_3) and (v_2, c_1) with (v_1, c_1) and (v_2, c_3) and this increases $|N(c_3) \cap X|$ from one to two. We can continue this process, now swapping edges (v_4, c_2) and (v_5, c_3) with (v_4, c_3) and (v_5, c_2). After this step $|N(c_3) \cap X| = 3$ and thus $N(c_3) = X$. We can now obtain the simple bipartite G' that realizes $(\boldsymbol{a'}, \boldsymbol{b'})$ by deleting check node c_3 and its adjacent edges.
△

The algorithm for constructing a simple bipartite graph that realizes a given bigraphic sequence pair follows immediately from Theorem 5.9. Observe that the roles of $\boldsymbol{a}$ and $\boldsymbol{b}$ can be reversed in the theorem. We can thus start with either connect one check node to its variable nodes, or conversely, with a variable node connecting it to check nodes. The algorithm is a greedy algorithm where the node with highest residual degree Δ is picked as either a variable node or a check node, and it is connected to the nodes on the other side with the highest Δ residual degrees. Theorem 5.9 proves the correctness of this algorithm.

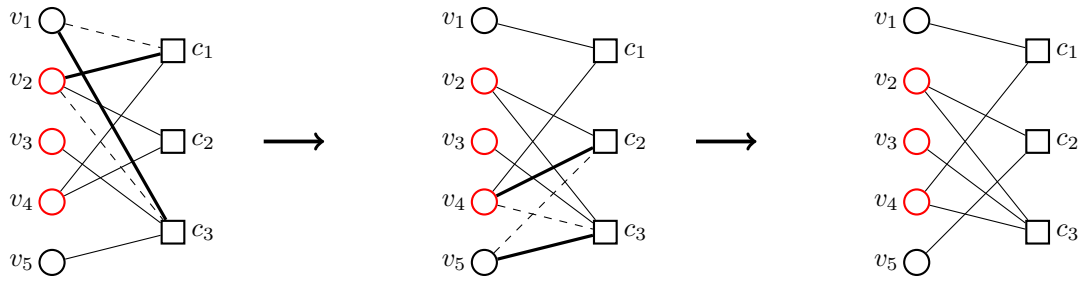


Figure 14: Three simple bipartite graphs, all realizing the sequence pair $((2,2,1,1,1),(3,2,2))$. Using edge swaps we can modify the graph until c_3, the check node with the highest degree, is connected to the $d(c_3) = 3$ variable nodes with the highest degree. We can use this final graph to show that also the sequence pair we get via Theorem 5.9, $((1,1,1,1,0),(2,2))$, is graphical. We do this by deleting check node c_3 with its attached edges, after which we have found the desired graph.

5.5 Forbidden connections

In this section we prove a slightly more generalized Hakimi-Havel theorem, which is largely formed from the work of [45], where they use this theorem to provide an algorithm to construct all simple graphs realizing a given graphical degree sequence $\boldsymbol{d}$. The theorem tells us how to check if a degree sequence is graphical if there are some 'forbidden edges' which are edges that are not allowed to exist in the graph. We will use this theorem to generate bipartite graphs with a large girth. Before presenting our main theorem, several definitions and observations are needed.

Definition 5.11. Let $A(i)$ be an increasingly ordered set of d_i distinct nodes associated with node i:

$$A(i) = \{a_k \mid a_k \in V, \quad a_k \neq i, \quad \forall k, \quad 1 \leq k \leq d_i\}.$$

This set $A(i)$ is called the *adjacency set* of i.

Definition 5.12. If for two adjacency sets $A(i) = \{\dots, a_k, \dots\}$ and $B(i) = \{\dots, b_k, \dots\}$ we have $a_k \leq b_k$ for all $1 \leq k \leq d_i$, we say that $A(i) \leq B(i)$.

Definition 5.13. Let $d_1 \geq d_2 \geq \cdots \geq d_n \geq 1$ be a graphical degree sequence, and let $A(i)$ be an adjacency set of node i. The degree sequence reduced by $A(i)$ is defined as

$$d'_k|_{A(i)} = \begin{cases} d_k - 1 & \text{if } k \in A(i) \\ d_k & \text{if } k \in [1,n] \backslash (A(i) \cup \{i\}) \\ 0 & \text{if } k = i. \end{cases}$$

In other words, the reduced degree sequence $\boldsymbol{d'}|_{A(i)}$ is obtained after removing node i together with all its edges from G.

It is not hard to see that if $\boldsymbol{d}$ is a non-increasing graphical sequence with $d_j > d_k$, $j,k \in \{1,\dots,n\}$, we can move an edge that is attached to d_j from d_j to d_k and thus the sequence $\{d_1,\dots,d_j-1,\dots,d_k+1,\dots,d_n\}$ is also graphical but not necessarily in non-increasing order. This observation can be used to prove the following lemma.

Lemma 5.14. *Let $\boldsymbol{d} = (d_1, d_2, \dots, d_n)$, be a non-increasing graphical sequence, and let $A(i)$, $B(i)$ be two adjacency sets for fixed node $i \in V$, such that $B(i) \leq A(i)$. If the degree sequence $\boldsymbol{d'}|_{A(i)}$ reduced by $A(i)$ is graphical, then the degree sequence $\boldsymbol{d'}|_{B(i)}$ reduced by $B(i)$ is also graphical.*

Let $B^m(i) = \{b_1,\dots,b_m,a_{m+1},\dots,a_{d_i}\}$, with induction on m we can prove that the degree sequence reduced by $B^m(i)$ is graphical for every $1 \leq m \leq d_i$. This is based on the fact that if $b_j < a_j$, then the degree of node b_j is larger than the degree of a_j, and thus we can move an edge from b_j to a_j. This means that if the degree sequence reduced by $B^{j-1}(i)$ is graphical, then so is $B^j(i)$.

Definition 5.15. Let $\boldsymbol{d} = (d_1, d_2, \ldots, d_n)$ be a non-increasing graphical sequence and let $F(i)$ be a subset of V for every $1 \leq i \leq d_i$. We define the *leftmost adjacency set* $L_F(i)$ of a node i as the set containing the d_i nodes with the largest degrees, that are not in a forbidden set $F(i)$.

Since $\boldsymbol{d}$ is non-increasing, the nodes in the leftmost adjacency set $L(i)$ are the first d_i nodes in the sequence that are not in the forbidden set $F(i)$. The set $L(i)$ is called the leftmost adjacency set, because it follows from the definition that for every $Y(i) = \{y_1, \ldots, y_{d_i}\}$ disjoint from $F(i) \cup \{i\}$ we have that $L(i) \leq Y(i)$.

We can now give the main theorem.

Theorem 5.16. *Let $\boldsymbol{d} = (d_1, \ldots, d_n)$ be a non-increasing graphical sequence. Let $i \in V$ be a fixed node, let $F(i)$ be a subset of V and let $L_F(i)$ be the leftmost adjacency set of i. Then the degree sequence $\boldsymbol{d} = (d_1, \ldots, d_n)$ can be realized by a simple graph $G(V, E)$ in which $(i, j) \notin E$, for all $j \in F(i)$, if and only if the degree sequence reduced by $L_F(i)$ is graphical.*

Proof. One direction of this proof is straightforward. If the degree sequence reduced by $L_F(i)$ is graphical, we can add node i and connect it with all nodes in $L_F(i)$. Since $L_F(i)$ and $F(i)$ are disjoint, this realizes a graphical realization of $\boldsymbol{d}$ in which there are no connections between node i and any node in $F(i)$.

For the other side we assume that $\boldsymbol{d}$ is graphical with a graphical realization where there are no edges between i and $F(i)$. We need to show that the sequence obtained from $\boldsymbol{d}$ by reduction via $L(i)$ is also graphical. Since $\boldsymbol{d}$ has a graphical representation G without edges between i and $F(i)$, the set $A(i)$, containing all nodes that are connected to node i in G, is disjoint from $F(i)$. With our previous observation this means that $L(i) \leq A(i)$ and thus Lemma 5.14 states that the degree sequence reduced by $L(i)$ is graphical. □

This is a generalized version of the Havel-Hakimi theorem, since we can take $F(i) = \emptyset$ for every $i \in V$, so that Theorem 5.16 is indentical to Theorem 5.5. The theorem can also be easily adjusted to bipartite graphs. To do so, some definitions have to be adjusted. The adjacency set $A(i)$ of a check node v_i consist of variable nodes, and vice versa. Moreover, the leftmost adjacency set of a check node i is the set containing the b_i variable nodes with the highest degree, such that the variable nodes are not in $F(i)$ which is the set of forbidden connections. Furthermore, if $(\boldsymbol{a}, \boldsymbol{b})$ is a bigraphic sequence pair, and $A(i)$ a set of variable nodes, adjacent to a check node, the sequence pair reduced by $A(i)$ is now defined as $(\boldsymbol{a'}_{A(i)}, \boldsymbol{b'}_{A(i)})$, with

$$a'_k|_{A(i)} = \begin{cases} a_k - 1 & \text{if } k \in A(i) \\ a_k & \text{if } k \notin A(i), \end{cases} \qquad b'_k|_{A(i)} = \begin{cases} b_k & \text{if } k \neq i \\ 0 & \text{if } k = i. \end{cases}$$

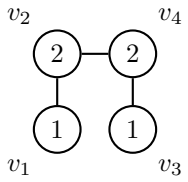
The theorem for bipartite graphs can now be stated as following.

Theorem 5.17. *Let $(\boldsymbol{a}, \boldsymbol{b})$ be a pair of degree sequences, with $\boldsymbol{a} = (a_1, \ldots, a_p)$ and $\boldsymbol{b} = (b_1, \ldots, b_q)$, both in non-increasing order. Let $i \in V_2$ be a fixed check node, let $F(i)$ be a subset of V_1 and let $L_F(i)$ be the leftmost adjacency set of i. Then the degree sequence pair (a, b) can be realized by a simple bipartite graph $G(V_1 \cup V_2, E)$ in which $(i, j) \notin E$, for all $j \in F(i)$, if and only if the degree sequence pair reduced by $L_F(i)$ is bigraphic.*

Theorem 5.16 and Theorem 5.17 can be used to detect if certain edges are 'allowed' in a graph G. Assume that we have a graphical degree sequence $\boldsymbol{d}$. Instead of generating a graph that realizes $\boldsymbol{d}$ via the Havel-Hakimi algorithm, we can add a random edge and check with Theorem 5.16 or Theorem 5.17 if the resulting degree sequence still can be realized by a simple graph avoiding the forbidden connections. The set of forbidden connections consists of all the connections that are already made since there cannot be multiple edges between two nodes.

Instead of picking edges at random, we can use a more clever way of choosing these edges, such that small cycles are avoided. Section 5.6 will explore this idea. First, we clarify this concept with an example. To keep the idea as simple as possible we start with an example for a single graphical degree sequence. The concept for a sequence pair of bipartite graphs is similar, albeit less trivial.

Example 5.18. Consider the graphical sequence $\boldsymbol{d} = (2, 2, 1, 1)$. This sequence can be realized by the following graph:



We will use Theorem 5.16 to see which edges are allowed in the construction. After the addition of each edge we need to check if the degree sequence is still graphical, taking into account that previous edges are forbidden connections. Assume we first add the edge (v_2, v_4), we can check that the residual sequence $(1, 1, 1, 1)$ is still graphical, even when the edge (v_2, v_4) is a forbidden connection. We can thus safely keep this edge since it does not break the graphicality of $\boldsymbol{d}$. Next, we can add the edge between v_1 and v_3. Now, we check if the residual sequence $(1, 1)$ is still graphical, considering the forbidden connections. Although the sequence itself is graphical, it cannot be realized, since an edge between v_2 and v_4 is needed. This means that adding the edge (v_1, v_3) breaks the graphicality and thus we need to continue with another connection. If v_1 is connected to v_2 or v_3, we can finish the graph by connecting the other two nodes.

△

The theorem to check if the residual degree sequences are still graphical without the forbidden connections, can detect failure in an early stage. A naive approach would be to add random edges until either every node got the desired result or until loops or double edges arise. The problem with this approach is that complications appear only at the end of the process. With this theorem we are capable of catching these obstacles earlier. In Example 5.18 we see that the algorithm detects the problem in the second last step. However, when a graph is much larger, this procedure can save a lot of time by detecting the problem at the earliest time possible.

5.6 Constructing Tanner graphs with a large girth

Using the idea of forbidden edges of Section 5.5, we introduce an algorithm in this section which can be considered to be used for the construction of Tanner graphs with a large girth. We do not give a formal proof about the results of the algorithm, but instead give an intuition on why this algorithm is valuable.

Our algorithm is motivated by the following notion. The Havel-Hakimi algorithm for constructing bipartite graphs is a greedy algorithm, which leads to many connections between nodes with high degrees. This procedure can quickly lead to small cycles. Let $G = (V_1 \cup V_2, E)$ be a Tanner graph, and suppose v_1, v_2 are the two variable nodes with the highest degrees in V_1 and c_1, c_2 are the two check nodes with the highest degree in V_2. It is not odd to assume that the degrees of v_1, v_2 and c_1, c_2 are strictly larger than other nodes in their vertex set. Assuming that $v_1 \geq 2$, the Havel-Hakimi algorithm connects v_1 with the $d(v_1)$ nodes with the highest degree, which includes c_1 and c_2. Since the degrees of c_1 and c_2 were strictly bigger than the other nodes, they are still on top of the list with their residual degree. This means that if we need to connect v_2 to the $d(v_2)$ nodes with the highest degree it is again connected to c_1 and c_2 and this leads to a cycles of length four.

Instead of connecting high degree nodes to other high degree nodes, we suggest to distribute the edges better among the nodes. We could try to do this by studying the complement graph. The complement graph G_c of a graph $G = (V, E)$ is defined on the same nodes, but with the complement of the edges, $G_c = (V, (V \times V) \backslash E)$. If there is a connection between two nodes u, v in G, there is no connection is G_c and vice versa. The idea is that if G has small cycles, G_c avoids these small cycles. Unfortunately, if high degree nodes are connected to other high degree nodes, this means that low degree nodes are also more likely to be connected to low degree nodes. These nodes have a high degree in the complement graph so we get a similar problem.

A more effective way would be to connect high degree nodes as much as possible to low degree nodes. This is how our algorithms operates. It starts with connecting high degree nodes in V_1 to low degree nodes V_2. We need to make sure that after every connection, the remaining degree sequences are still graphical, considering the forbidden connections. This is where we use Theorem 5.16 of Section 5.5. Algorithm 3 summarizes our proposed algorithm.

Algorithm 3 Constructing large girth Tanner graphs for irregular LDPC codes

Input: A bigraphic pair $(\boldsymbol{a}, \boldsymbol{b})$ of degree sequences $\boldsymbol{a} = (a_1, \ldots, a_p)$ and $\boldsymbol{b} = (b_1, \ldots, b_q)$.
Output: A simple bipartite graph $G = (V_1 \cup V_2, E)$ that realizes the pair $(\boldsymbol{a}, \boldsymbol{b})$.

Let $G = (V_1 \cup V_2, E)$ be a graph with p nodes in V_1, q nodes in V_2 and $E = \emptyset$.

while $\exists i \in \{1, \ldots, p\}$ so that $a_i \neq 0$ **do**
    Bring $\boldsymbol{a}$ and $\boldsymbol{b}$ in non-increasing order.
    Connect in G the variable node v_i with the highest degree in V_1 to the check node c_j which is not checked yet for v_i, with the lowest, non-zero, degree in V_2.
    Let $A(i)$ be the set containing the $a_1 - 1$ check nodes with the highest degrees, such that there are no forbidden connections, together with check node c_j.
    Let $(\boldsymbol{a'}, \boldsymbol{b'})$ be the degree sequence pair reduced by $A(i)$. Check with Theorem 5.8 if this sequence pair $(\boldsymbol{a'}, \boldsymbol{b'})$ is bigraphic, and thus if the connection between v_i and c_j is permitted.

    if $(\boldsymbol{a'}, \boldsymbol{b'})$ is bigraphic **then**
        Set $a_i = a_i - 1$ and $b_j = b_j - 1$. This is the residual degree sequence after connecting a_i to b_j.
        Make (v_i, c_j) a forbidden connection.
    else
        Remove the connection between v_i and c_j.
        Add check node c_j to the nodes that has been checked for variable node v_i.
    end if
end while

In this algorithm we connect the highest degree variable node to check nodes. We could have also reversed the sequences in the pair $(\boldsymbol{a}, \boldsymbol{b})$ such that the highest degree check node is connected to variable nodes. A combination of both is also possible. The question if the choice of picking either check nodes or variable nodes is important requires further investigation. Another opportunity to optimize this algorithm is to question if we need to check if every inequality of the Gale-Ryser theorem is needed or if some inequalities can be reduced, since similar inequalities could have been checked before. The optimizations of the inequalities as described in the note of Tripathi and Vijay [38] can help with this. For example, in this note, it is stated that we only need to check the first s inequalities, where s is the largest integer such that $d_s \geq s$. We observe that, after adding an edge, s decreases and thus less inequalities need to be checked.

6 Conclusion

To conclude, this thesis has focused on exploring the benefits of large girth Tanner graphs in generating irregular low-density parity-check (LDPC) codes with excellent performance, approaching Shannon's limit. The initial part of the thesis reviewed the theoretical background of LDPC codes. We have studied the encoding and decoding processes of LDPC codes, together with their optimizations. The goals of the thesis was first of all to explore the reasoning behind the usage of Tanner graphs with a large girth. The answer lies in the Belief Propagation (BP) decoding algorithm. We have seen that the BP decoder is optimal on Tanner graphs that are a tree, but since codes whose Tanner graph is a tree have a small minimum distance, these codes are not used. The BP decoder is near-optimal for Tanner graphs with a large girth, since they locally look like a tree. Therefore, the BP decoder is in practice very functional on large girth Tanner graphs.

The second goal of the thesis included the construction of large girth Tanner graphs. Luby showed that irregular LDPC codes have an even better performance than regular LDPC codes [9]. We therefore focused on generating large girth Tanner graphs for irregular graphs. The degree distributions of a code serve as a compact way to view the degrees of the nodes in the Tanner graph. To be able to construct a code with a given degree distribution we first of all need to know if there exists graphs that could represent this code. And if there exist multiple representations, we need to find a Tanner graph representation with a large girth. The first question is answered in Section 5.3 via the Erdős-Gallai theorem and the Havel-Hakimi theorem. The Gale-Ryser theorem is used to check if the degree sequences of a bipartite graph are graphical. For the second question we proposed an algorithm in Section 5.6 that uses forbidden connections as used in another context by Kim [45]. We do not know how effective the algorithm is, because it is not tested yet. In further work it would be important to see how the algorithm performs in real-world applications. Additionally, the algorithm can be possibly improved via multiple questions, as they are discussed in Section 5.6.

I hope this thesis will inspire further research in this exciting and rapidly growing field.

References

[1] Claude E Shannon. "A mathematical theory of communication". In: *The Bell system technical journal* 27.3 (1948), pp. 379–423.

[2] Robert Gallager. "Low-density parity-check codes". In: *IRE Transactions on information theory* 8.1 (1962), pp. 21–28.

[3] David JC MacKay and Radford M Neal. "Near Shannon limit performance of low density parity check codes". In: *Electronics letters* 33.6 (1997), pp. 457–458.

[4] Tom Richardson and Rüdiger Urbanke. "Efficient Encoding of Low-Density Parity-Check Codes". In: *Information Theory, IEEE Transactions on* 47 (Mar. 2001), pp. 638–656. DOI: 10.1109/18.910579.

[5] Tom Richardson and Rüdiger Urbanke. *Modern Coding Theory*. Cambridge University Press, 2008. DOI: 10.1017/CBO9780511791338.

[6] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988, pp. 143–194.

[7] Changyan Di et al. "Finite-length analysis of low-density parity-check codes on the binary erasure channel". In: *IEEE Transactions on Information theory* 48.6 (2002), pp. 1570–1579.

[8] Michael E O'Sullivan. "Algebraic construction of sparse matrices with large girth". In: *IEEE Transactions on Information Theory* 52.2 (2006), pp. 718–727.

[9] Michael Luby et al. "Analysis of low density codes and improved designs using irregular graphs". In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 1998, pp. 249–258.

[10] Shu Lin. *Costello. DJ Error control coding*. 2004.

[11] Yfke Dulek and Christian Schaffner. *Lecture notes Information Theory*. University of Amsterdam, Master of Logic, 2017.

[12] R. Tanner. "A recursive approach to low complexity codes". In: *IEEE Transactions on Information Theory* 27.5 (1981), pp. 533–547. DOI: 10.1109/TIT.1981.1056404.

[13] E. Berlekamp, R. McEliece, and H. van Tilborg. "On the inherent intractability of certain coding problems (Corresp.)" In: *IEEE Transactions on Information Theory* 24.3 (1978), pp. 384–386. DOI: 10.1109/TIT.1978.1055873.

[14] Luis Salamanca et al. "MAP decoding for LDPC codes over the binary erasure channel". In: *2011 IEEE Information Theory Workshop*. 2011, pp. 145–149. DOI: 10.1109/ITW.2011.6089364.

[15] Jonathan S Yedidia, William T Freeman, Yair Weiss, et al. "Understanding belief propagation and its generalizations". In: *Exploring artificial intelligence in the new millennium* 8.236-239 (2003), pp. 0018–9448.

[16] M.P.C. Fossorier, M. Mihaljevic, and H. Imai. "Reduced complexity iterative decoding of low-density parity check codes based on belief propagation". In: *IEEE Transactions on Communications* 47.5 (1999), pp. 673–680. DOI: 10.1109/26.768759.

[17] David JC MacKay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003, pp. 241–243.

[18] Tom Richardson. "Error floors of LDPC codes". In: *Proceedings of the annual Allerton conference on communication control and computing*. Vol. 41. 3. The University; 1998. 2003, pp. 1426–1435.

[19] Aiden Price and Joanne L. Hall. "A Survey on Trapping Sets and Stopping Sets". In: *CoRR* abs/1705.05996 (2017). arXiv: 1705.05996. URL: http://arxiv.org/abs/1705.05996.

[20] Hua Xiao and Amir H Banihashemi. "Improved progressive-edge-growth (PEG) construction of irregular LDPC codes". In: *IEEE Communications Letters* 8.12 (2004), pp. 715–717.

[21] Milos Ivkovic, Shashi Kiran Chilappagari, and Bane Vasic. "Eliminating trapping sets in low-density parity-check codes by using Tanner graph covers". In: *IEEE transactions on information theory* 54.8 (2008), pp. 3763–3768.

[22] Eirik Rosnes and Oyvind Ytrehus. "An algorithm to find all small-size stopping sets of low-density parity-check matrices". In: *2007 IEEE International Symposium on Information Theory*. IEEE. 2007, pp. 2936–2940.

[23] Sarah J Johnson and Steven R Weller. "Codes for iterative decoding from partial geometries". In: *IEEE Transactions on Communications* 52.2 (2004), pp. 236–243.

[24] Olgica Milenkovic, Emina Soljanin, and Philip Whiting. "Asymptotic spectra of trapping sets in regular and irregular LDPC code ensembles". In: *IEEE Transactions on Information Theory* 53.1 (2006), pp. 39–55.

[25] Tuvi Etzion, Ari Trachtenberg, and Alexander Vardy. "Which codes have cycle-free Tanner graphs?" In: *IEEE Transactions on Information Theory* 45.6 (1999), pp. 2173–2181.

[26] Andrew Kozlík. "Kódování a efektivita LDPC kódů". In: (2011).

[27] Alexander T Ihler et al. "Loopy belief propagation: convergence and effects of message errors." In: *Journal of Machine Learning Research* 6.5 (2005).

[28] D.J.C. MacKay. "Good error-correcting codes based on very sparse matrices". In: *IEEE Transactions on Information Theory* 45.2 (1999), pp. 399–431. DOI: 10.1109/18.748992.

[29] J. Campello, D.S. Modha, and S. Rajagopalan. "Designing LDPC codes using bit-filling". In: *ICC 2001. IEEE International Conference on Communications. Conference Record (Cat. No.01CH37240)*. Vol. 1. 2001, pp. 55–59. DOI: 10.1109/ICC.2001.936272.

[30] Jingyu Kang et al. "Quasi-cyclic LDPC codes: an algebraic construction". In: *IEEE Transactions on Communications* 58.5 (2010), pp. 1383–1396. DOI: 10.1109/TCOMM.2010.05.090211.

[31] Mohsen Bayati, Andrea Montanari, and Amin Saberi. "Generating random graphs with large girth". In: *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2009, pp. 566–575.

[32] Yu Kou, Shu Lin, and Marc PC Fossorier. "Low density parity check codes: Construction based on finite geometries". In: *Globecom'00-IEEE. Global Telecommunications Conference. Conference Record (Cat. No. 00CH37137)*. Vol. 2. IEEE. 2000, pp. 825–829.

[33] Sae-Young Chung et al. "On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit". In: *IEEE Communications letters* 5.2 (2001), pp. 58–60.

[34] T.J. Richardson, M.A. Shokrollahi, and R.L. Urbanke. "Design of capacity-approaching irregular low-density parity-check codes". In: *IEEE Transactions on Information Theory* 47.2 (2001), pp. 619–637. DOI: 10.1109/18.910578.

[35] Václav Havel. "A remark on the existence of finite graph (Hungarian)". In: *Casopis Pest., Mat.* 80 (1955), pp. 477–480.

[36] S Louis Hakimi. "On realizability of a set of integers as degrees of the vertices of a linear graph. I". In: *Journal of the Society for Industrial and Applied Mathematics* 10.3 (1962), pp. 496–506.

[37] T. Gallai P.Erdős. "Graphs with prescribed degree of vertices (Hungarian)". In: *Matematikai Lapok* 11 (1960), pp. 264–274.

[38] Amitabha Tripathi and Sujith Vijay. "A note on a theorem of Erdős & Gallai". In: *Discrete Mathematics* 265.1 (2003), pp. 417–420. ISSN: 0012-365X. DOI: https://doi.org/10.1016/S0012-365X(02)00886-5. URL: https://www.sciencedirect.com/science/article/pii/S0012365X02008865.

[39] Julius Petersen. "Die Theorie der regulären graphs". In: (1891).

[40] Michael Koren. "Extreme degree sequences of simple graphs". In: *Journal of Combinatorial Theory, Series B* 15.3 (1973), pp. 213–224. ISSN: 0095-8956. DOI: https://doi.org/10.1016/0095-8956(73)90037-3. URL: https://www.sciencedirect.com/science/article/pii/0095895673900373.

[41] Shuo-Yen R Li. "Graphic sequences with unique realization". In: *Journal of Combinatorial Theory, Series B* 19.1 (1975), pp. 42–68.

[42] Amotz Bar-Noy et al. "On Realizing a Single Degree Sequence by a Bipartite Graph". In: *18th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2022.

[43] David Gale et al. "A theorem on flows in networks". In: *Pacific J. Math* 7.2 (1957), pp. 1073–1082.

[44] Herbert J Ryser. "Combinatorial properties of matrices of zeros and ones". In: *Canad. J. Math.* (1957), pp. 371–377.

[45] Hyunju Kim et al. "On realizing all simple graphs with a given degree sequence". In: *Discrete Mathematics* (2008).