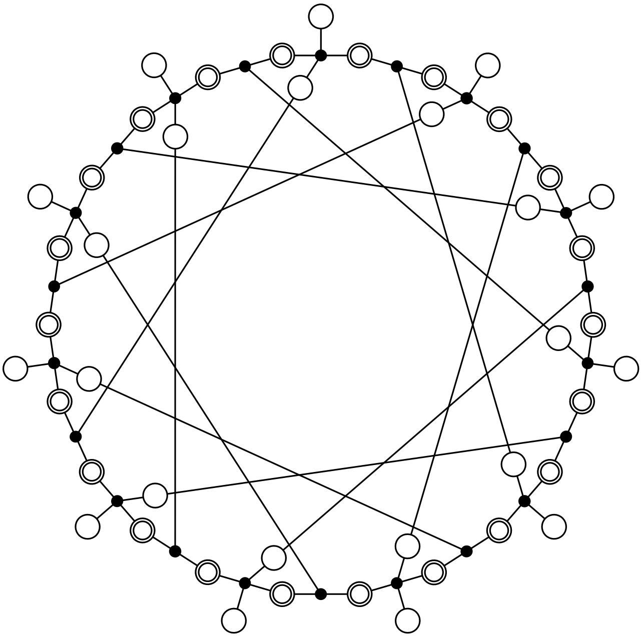
Linköping Studies in Science and Technology
Dissertation No. 440

Codes and Decoding on General Graphs

Niclas Wiberg



Department of Electrical Engineering
Linköping University, S-581 83 Linköping, Sweden

Linköping 1996

Linköping Studies in Science and Technology
Dissertation No. 440

Codes and Decoding on General Graphs

Niclas Wiberg

Department of Electrical Engineering
Linköping University, S-581 83 Linköping, Sweden

Linköping 1996

Corrections applied as of errata October 30, 1996

ISBN 91-7871-729-9
ISSN 0345-7524

To Kristin,
Isak, and Benjamin

Abstract

Iterative decoding techniques have become a viable alternative for constructing high performance coding systems. In particular, the recent success of turbo codes indicates that performance close to the Shannon limit may be achieved. In this thesis, it is showed that many iterative decoding algorithms are special cases of two generic algorithms, the min-sum and sum-product algorithms, which also include non-iterative algorithms such as Viterbi decoding. The min-sum and sum-product algorithms are developed and presented as generalized trellis algorithms, where the time axis of the trellis is replaced by an arbitrary graph, the “Tanner graph”. With cycle-free Tanner graphs, the resulting decoding algorithms (e.g., Viterbi decoding) are maximum-likelihood but suffer from an exponentially increasing complexity. Iterative decoding occurs when the Tanner graph has cycles (e.g., turbo codes); the resulting algorithms are in general suboptimal, but significant complexity reductions are possible compared to the cycle-free case. Several performance estimates for iterative decoding are developed, including a generalization of the union bound used with Viterbi decoding and a characterization of errors that are uncorrectable after infinitely many decoding iterations.

Acknowledgments

The best guarantee to a successful work, undoubtedly, is to work among intelligent and helpful people. Luckily for me, I had the opportunity to work closely with Andi Löliger and Ralf Kötter for almost four years. Andi, who has been my co-supervisor, has many times pointed me in directions that turned out to be very interesting (sometimes he had to push me to get me started). Furthermore, he was always (well, almost) available for fruitful and engaged discussions regarding my work. It is not an overstatement to say that this thesis would not have been written without him. Thank you, Andi.

The many discussions with Ralf Kötter have also been very valuable. Ralf's knowledge in algebra, graph theory, and other subjects important to me, combined with his great interest in iterative decoding, has often lead to important insights for both of us.

While a deep penetration of a subject is perhaps the essence of a doctoral thesis, it is equally important to obtain a wider understanding and perspective of a problem area. I am deeply thankful to my professor and supervisor Ingemar Ingemarsson for providing an environment that always stimulated me to learn more about communication theory in general. By involving me and other graduate students in the planning of undergraduate courses, and by introducing unconventional teaching methods, we were all forced to review our own knowledge again and again.

All my friends and colleagues at the divisions of information theory, image coding, and data transmission have helped me a lot by providing an inspiring and enjoyable atmosphere. It is a fact that the best way to learn something is to explain it to other people; many are the times when I, uninvited, have exploited the patience of my colleagues by describing my unsorted thoughts on the whiteboard in the coffee room and elsewhere. Thank you for listening—and please forgive me.

A special thanks goes to my friend Håkan Andersson, with whom I shared the office for some years. Apart from our many discussions, he has helped me a lot by reading my manuscripts and providing constructive criticism on all aspects of the thesis. Maurice Devenney also did a very good job in reading and correcting my writing.

Needless to say, my parents have been very important for me. Here, I would like to thank my Father, who was the first to bring me into mathematics and science. By having a programmable home computer at the age of 13, my career was practically settled.

I dedicate the thesis to my wife Kristin and my children Isak and Benjamin. Sharing with you the few hours that I did not work on the thesis made the long hours at work bearable. Thank you for your support, understanding, and love!

Linköping, April 1996
Niclas Wiberg

Contents

1 Introduction **1**

1.1 Decoding Complexity 2

1.2 Iterative Decoding Based on Tanner Graphs 3

1.3 Turbo Codes 4

1.4 Thesis Outline 4

2 Code Realizations Based on Graphs **6**

2.1 Systems, Check Structures and Tanner Graphs 7

2.2 Turbo Codes 10

3 The Min-Sum and Sum-Product Algorithms **12**

3.1 The Min-Sum Algorithm 13

3.2 The Sum-Product Algorithm 18

3.3 Updating Order and Computation Complexity 21

3.4 Optimized Binary Version of the Min-Sum Algorithm 22

3.5 Non-Decoding Applications 23

3.6 Further Unifications 24

4 Analysis of Iterative Decoding **25**

4.1 The Computation Tree 27

4.2 The Deviation Set 29

5 Decoding Performance on Cycle-Free Subgraphs **37**

5.1 Estimating the Error Probability with the Union Bound 37

5.2 Asymptotic Union Bound 40

5.3 Computing Statistical Moments of Final Costs 45

5.4 Gaussian Approximation of Log-Cost-Ratios 49

6 Decoding Performance with Cycles **52**

6.1 Cycle Codes 54

6.2 Tailbiting Trellises 57

6.3 The General Case 59

6.4 Turbo Codes 60

7 More on Code Realizations **61**

7.1 Realization Complexity 62

7.2 Cycle-Free Realizations 63

7.3 Realizations with Cycles 64

7.4 Modeling Complicated Channels 71

8 Conclusions **74**

A Proofs and Derivations **76**

A.1 Proof of Theorems 3.1 and 3.2 76

A.2 Derivations for Section 3.3 80

A.3 Derivation for Section 3.4 84

A.4 Derivation used in Section 5.4 85

A.5 Proofs for Section 6.1 88

A.6 Proof of Theorem 7.2 91

References **92**

Chapter 1

Introduction

This thesis deals with methods to achieve reliable communication over unreliable channels. Such methods are used in a vast number of applications which affect many people's everyday life, for example mobile telephony, telephone modems for data communication, and storage mediums such as compact discs.

A basic scheme for communicating over unreliable channels is illustrated in Figure 1.1. The message to be sent is encoded with a channel code before it is transmitted on the channel. At the receiving end, the output from the channel is decoded back to a message, hopefully the same as the original one. A fundamental property of such systems is Shannon's channel coding theorem, which states that reliable communication can be achieved as long as the information rate does not exceed the "capacity" of the channel, provided that the encoder and decoder are allowed to operate on long enough sequences of data (extensive treatments can be found in many textbooks, e.g. [1]).

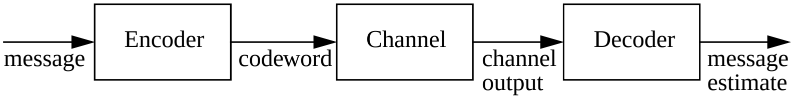


Figure 1.1 *Shannon's model for reliable communication on an unreliable channel.*

We will deal with the decoding problem, i.e. finding a good message estimate given the channel output. The problem can be solved, in principle, by searching through all possible messages and comparing their corresponding codewords with the channel output, selecting the message which is most likely to result in the observed channel output. While such a method can be made optimal in the sense of minimizing the error probability, it is useless in practice because the number of messages is too large to search through them all. The interesting (and difficult!) aspect of the decoding problem is to find methods with reasonable complexity that still give sufficiently good performance.

Decoding methods can be divided, roughly, in two classes: algebraic and "probabilistic". Algebraic methods are typically based on very powerful codes, but are only suited to relatively reliable channels. In such cases, however, they can provide virtually error-free communication. By "probabilistic" methods, we mean methods that are designed to use the channel output as efficiently as possible, approaching the performance of an optimal decoder. Such methods are better suited to highly unreliable channels than algebraic methods are. Unfortunately, probabilistic methods in general work only for relatively weak codes, and consequently, cannot completely avoid decoding errors. To achieve error-free communication on highly unreliable channels, one typically uses a combination of probabilistic and algebraic decoding methods.

We will consider probabilistic decoding methods in this thesis, with the goal of finding decoding methods that both use the channel output efficiently and are still capable of handling relatively powerful codes. It is assumed that the reader is familiar with the basics of communication theory.

1.1 Decoding Complexity

Until recently, the most important probabilistic decoding method has been the Viterbi algorithm [2]. A main reason for its success is that it is optimal, or maximum-likelihood, in the sense that it minimizes the probability of decoding error for a given code. The main drawback, on the other hand, is the computation complexity, which is very high for good codes (the number of operations grows exponentially with the minimum distance of the code, cf. [3]).

The reason behind this high complexity is to be found in the trellis code description, illustrated in Figure 1.2 for a small code, on which the Viterbi algorithm is based. In a trellis, the structure of the code is expressed by viewing the code as a dynamic system, introducing a time axis on which the codeword components are laid out. (Of course, having such a time axis is practical too, since the components must be transmitted in some order anyway. With a memoryless channel, however, the transmission order is irrelevant from a theoretical perspective; the point here is that the decoding algorithm *assumes* some time axis on which it operates.) With this dynamic-system view, all dependence between the past and the future (with respect to some instant on the time axis) is expressed in the present state. The problem is that good codes by necessity must have a high dependence between the codeword components, implying that the state space of the trellis must be excessively large (cf. [3]), which leads to a high decoding complexity.

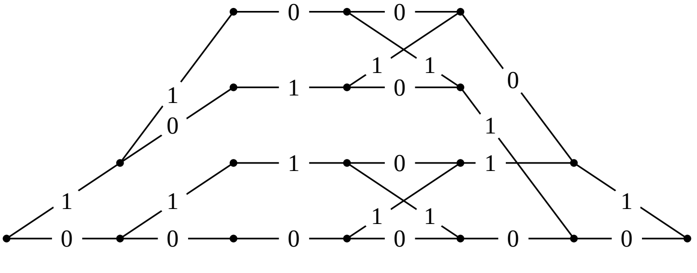


Figure 1.2 *A minimal trellis for a binary linear* (6, 3, 3) *code. The codewords are obtained by following paths from left to right and reading off the labels encountered.*

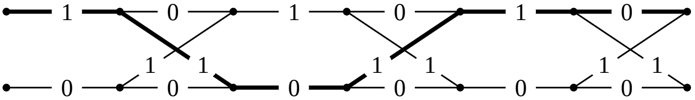


Figure 1.3 *A tailbiting trellis for a binary linear* (6, 3, 3) *code. The marked path corresponds to the codeword* 011011.

On the other hand, it has been known for a long time that *tailbiting* trellises can in some cases be much smaller than any ordinary trellis for the same code [4]. In a tailbiting trellis there are multiple starting states and equally many ending states (an ordinary trellis has single starting and ending states), and a path is required to start and end in the same state. Figure 1.3 illustrates a two-state tailbiting trellis for the same code as in Figure 1.2. In this case, the maximal number of states is reduced from four to two.

Loosely speaking, tailbiting trellises are based on a circular time axis. This is also the reason for the lower complexity: since the dependence between any two halves of the codewords may be expressed in *two* states, one in each direction on the time axis, the size of each of these state spaces may be smaller than if there were only one state space.

1.2 Iterative Decoding Based on Tanner Graphs

One of the major goals behind this thesis has been to search for code descriptions, or "realizations", with lower complexity than trellises. The other major goal was to investigate how such realizations could be exploited by decoding algorithms to achieve low decoding complexity. In particular, a promising direction seemed to be "iterative" decoding, i.e. decoding algorithms that operate on some internal state which is altered in small steps until a valid codeword is reached.

The main result is a framework for code realizations based on "Tanner graphs", which have the role of generalized time axes, and two generic iterative decoding algorithms that apply to any such realization. While our framework was developed as a combination and generalization of trellis coding and Gallager's low-density parity-check codes [5], the basic ideas were all present in Tanner's work "A recursive approach to low complexity codes" [6], including the two mentioned algorithms. Our main contributions to the framework is to explicitly include trellis-type realizations and to allow more general "metrics", or "cost functions"; the latter makes it possible to model, e.g., non-uniform a priori probability distributions, or channels with memory.

While this framework appeared interesting in itself, additional motivation for our research arose from an unexpected direction: turbo codes.

1.3 Turbo Codes

Undoubtedly, the invention of turbo codes [7] is a milestone in the development of communication theory. Compared to other coding systems, the improvement in performance obtained with turbo coding is so big that, for many applications, the gap between practical systems and Shannon's theoretical limit is essentially closed, a situation which was probably not predicted by anyone before the invention of turbo codes.

On the negative side, the turbo code construction is largely based on heuristics, in the sense that no theoretical analysis exists as of yet that can predict their amazing performance. More precisely, it is the decoding algorithm that remains to be analyzed; a relatively successful analysis of the theoretical code performance is given in [8].

As it turned out, turbo codes and their decoding algorithm fit directly into our general framework for codes and decoding based on graphs. This relation provided us with an additional research goal: to understand the turbo codes and their decoding performance, using our framework. Consequently, a lot of the material in this thesis is highly related to turbo codes and their decoding algorithms, and we have tried to make this connection explicit in many places.

1.4 Thesis Outline

The following two chapters present the graph-based framework. Chapter 2 provides a formal definition of code realizations based on graphs. While the basic ideas are due to Tanner [6], our definitions are more general and based on a different terminology. Chapter 3 presents the two decoding algorithms, the "min-sum" and the "sum-product" algorithms. We give a general formulation of these two algorithms, which are extended versions of Tanner's algorithms A and B, and we show their optimality when applied to realizations with cycle-free Tanner graphs (such as trellises). In both of these chapters we demonstrate explicitly how trellises and turbo codes fit into the framework. Another important example that appears throughout the thesis is Gallager's low-density parity-check codes [5].

The material in Chapter 2 and Chapter 3 is relatively mature and has been presented earlier, in [9], along with some parts of Chapter 6 and Chapter 7.

Chapters 4 through 6 are devoted to iterative decoding, i.e. the application of the min-sum and sum-product algorithms to realizations with cycles. In Chapter 4 we develop some fundamental results for performance analysis of iterative decoding; these are used in the following two chapters. Chapter 5 is focused on the decoding performance after the first few decoding iterations, before the cycles have affected the computation. In Chapter 6, we consider the performance obtained after this point, when the cycles do affect the computation. For a limited class of realizations, we analyze the asymptotic performance after infinitely many decoding iterations. The applicability of this result to turbo codes is also discussed.

In Chapter 7, we return to code realizations and complexity issues. A precise interpretation of the above complexity reasoning, regarding trellises and tailbiting trellises, will be given. We will also give a few more examples of code realizations, both with and without cycles. Some of these examples are closely related to turbo codes; in fact, they may point out a convenient way of constructing good turbo-like realizations with moderate block lengths. In addition, we will show how the framework can incorporate more complicated situations involving, e.g., channels with memory.

Chapter 8, finally, contains our conclusions.

Chapter 2

Code Realizations Based on Graphs

The central theme of this thesis is to describe codes by means of “equation systems”, whose structure are the basis for decoding algorithms. By *structure* we mean the relation between the variables and the equations. More precisely, the equation system defines a *bipartite graph* with vertices both for the variables and for the equations; an edge indicates that a particular variable is present in a particular equation.

Example 2.1 Figure 2.1 illustrates a linear equation system of six variables as well as its structure in the form of the mentioned bipartite graph. Assuming binary variables, this particular equation system defines a binary linear (6, 3, 3) code, with the equations corresponding to the rows of a parity-check matrix. ✻

While Example 2.1 and the term “equation system” conveys some of our ideas, the definitions that we will soon give include a lot more than just linear equations. For maximal generality, we allow an “equation” on a set of variables to be any subset of the possible value combinations; the “equation system” is then the intersection of these subsets. We hope that the reader will not be repelled by this rather abstract viewpoint. As an aid, we provide several examples that are familiar to the reader to show how our definitions unify many different code descriptions and decoding algorithms. In Chapter 7, we provide some examples of new code realizations too.

$$\begin{cases} x_1 + x_2 + x_3 = 0 \\ x_3 + x_4 + x_5 = 0 \\ x_5 + x_6 + x_1 = 0 \\ x_2 + x_4 + x_6 = 0 \end{cases}$$

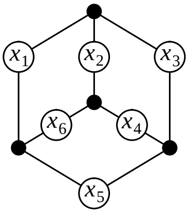


Figure 2.1 *An equation system and the corresponding bipartite graph. Each filled dot together with its neighbors corresponds to an equation with its variables.*

2.1 Systems, Check Structures and Tanner Graphs

A *configuration space* is a direct product $W = \prod_{s \in N} A_s$, where $\{A_s\}_{s \in N}$ is a collection of *alphabets* (or *state spaces*). We will usually assume that the *index set* N, as well as all alphabets A_s, $s \in N$, are finite. (Neither of these assumptions is essential, though.) Elements of W will be called *configurations*. Elements of N will be referred to as *sites*.

In many of our examples (e.g. 2.1), the index set N corresponds directly to the codeword components, i.e. $N = \{1, \ldots, n\}$, and the site alphabets are the binary field, i.e. $A_s = F_2$ for all $s \in N$. This means that the configuration space is the familiar space F_2^n of binary *n*-tuples. However, more general configuration spaces are also useful, e.g., for trellis-based constructions, where some of the sites do not correspond to codeword components but rather to "cuts" of the time axis; the corresponding site alphabets A_s are then state spaces of the trellis.

The components of a configuration $x \in W$ will be denoted by x_s, $s \in N$. More generally, the restriction (projection) of $x \in W$ to a subset $R \subseteq N$ of sites will be denoted by x_R. For a set of configurations $X \subseteq W$ and a site subset $R \subseteq N$ we will use the notation $X_R \triangleq \{x_R : x \in X\}$.

> **Definition 2.1** A *system* is a triple (N, W, B), where N is a set of sites, W is a configuration space, and $B \subseteq W$ is the *behavior.* (This "behavioral" notion of a system is due to Willems [10], cf. also [11].) The members of B will be called *valid configurations*. A system is *linear* if all alphabets A_s are vector spaces (or scalars) over the same field, the configuration space W is the direct product of the alphabets, and the behavior B is a sub*space* of W.

> **Definition 2.2** A *check structure* for a system (N, W, B) is a collection Q of subsets of N (*check sets*) such that any configuration $x \in W$ satisfying $x_E \in B_E$ for all check sets $E \in Q$ is valid (i.e., in B). The restriction B_E of the behavior to a check set E is called the *local behavior* at E. A configuration x is *locally valid* on E if $x_E \in B_E$. Note that a configuration is valid if and only if it is locally valid on all check sets.

The bipartite graph corresponding to a check structure Q for a system (N, W, B) is called a *Tanner graph* [6] for that system. Tanner graphs will be visualized as in Figure 2.1, with sites represented by circles and check sets by filled dots which are connected to those sites (circles) that they check.

The definitions 2.1 and 2.2 are "axiomatic" in the sense that they specify required properties for Q to be a check structure. Actual constructions are usually built in the opposite way, by specifying a check structure Q and the corresponding local behaviors B_E ("checks"), so that a desired behavior B is obtained. This was illustrated in Example 2.1, which can be seen as a system (N, W, B) with $N = \{1, \ldots, 6\}$ and W the six-dimensional binary vector space. The check structure Q = {{1, 2, 3}, {3, 4, 5}, {5, 6, 1}, {2, 4, 6}} and the local behaviors $B_E = \{000, 110, 101, 011\}$ (for all check sets $E \in Q$) together define the behavior B = {000000, 110001, 011100, 000111, 101101, 110110, 011011, 101010}.

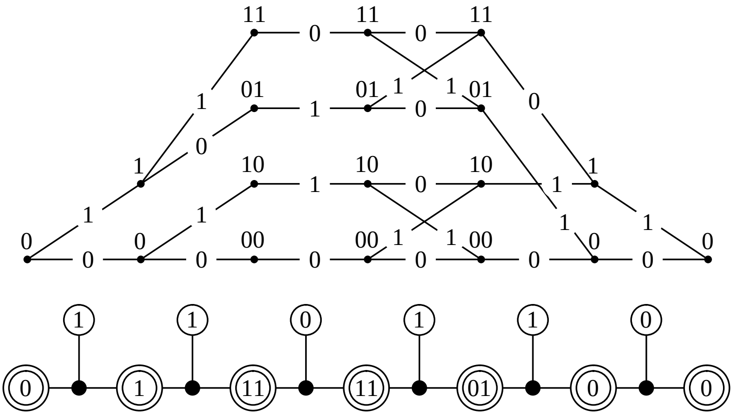


Figure 2.2 A trellis (top) for a (6,3,3) *code, and the corresponding Tanner graph. The values in the sites form a* ***valid configuration****, i.e., a* ***path****, which can be seen by checking locally in each trellis section.*

Any binary block code C of length n may be viewed as a system (N, W, B), where $N = \{1, 2, \ldots, n\}$, $W = F_2^n$, and $B = C$ is the set of codewords. For linear codes, a parity check matrix H (i.e., a matrix H such that $Hx^T = 0$ if and only if $x \in C$) defines a check structure with one check set for each row of H, containing those sites that have a “one” in that row. The corresponding local behaviors are simple parity checks. Of special interest is the case when the check sets have a small, fixed size k, and the sites are contained in a small, fixed number j of check sets. Such systems, which were introduced by Gallager in [5], are referred to as (j, k) *low-density parity-check codes*. When $j = 2$, i.e., when sites belong to exactly two check sets, the codes are referred to as *cycle codes* [12, pp. 136-138], since these codes are generated by codewords whose support corresponds to cycles in the Tanner graph. (Example 2.1 is a cycle code.)

So far, we have only considered systems where all sites correspond to components of the codewords. However, it is often useful to allow *hidden* sites, which do not correspond to codeword components but only serve to give a suitable check structure. The most familiar example of such descriptions is the trellis, as illustrated by the following example.

Example 2.2 Figure 2.2 illustrates the minimal trellis for the same binary linear (6,3,3) block code as in Example 2.1. The trellis is a system (N, W, B) with two types of sites: *visible* sites (corresponding to codeword components) and *hidden* sites (corresponding to the “cuts” between the trellis sections). Hidden sites are illustrated by double circles. The visible site alphabets are all binary, but the hidden site alphabets (the state spaces) contain one, two or four states. A configuration is an assignment of states and output symbols, one from each site alphabet. Such a configuration is valid (i.e. a path) if and only if each local configuration of left state, output symbol, and right state is valid, i.e., a branch. ✻

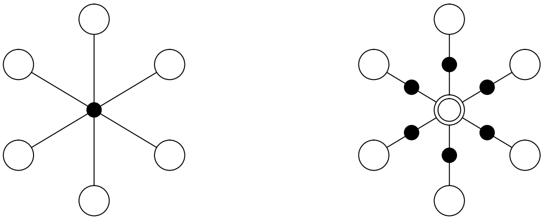


Figure 2.3 *Two trivial realizations that apply to any code (of length six).*

(As mentioned in the example, hidden sites will be depicted by double circles.) In general, if (N, W, B) is a system with hidden sites, and $V \subseteq N$ is the set of visible sites, then a *codeword* of the system is the restriction x_V of a valid configuration $x \in B$ to V. The *visible behavior* or *output code* of the system is $B_V \triangleq \{x_V : x \in B\}$. We consider a system (N, W, B) with a check structure Q to be a *description* or *realization* of the corresponding output code B_V.

The motivation for introducing hidden sites, and indeed for studying systems with check structures at all, is to find code descriptions that are suitable for decoding. There are many different realizations for any given code, and the decoding algorithms that we will consider in Chapter 3 can be applied, in principle, to all of them. However, both the decoding complexity and the performance will differ between the realizations.

One important property of a realization is its structural complexity. In the decoding algorithms that we will consider, all members of the site alphabets A_s and of the local behaviors B_E are considered explicitly during the decoding process, in some cases even several times. For this reason, the site alphabets and the local behaviors should not be too large; in particular, trivial check structures such as the one with a single check set $Q = \{N\}$, and the one with a single hidden site whose alphabet has a distinct value for each valid configuration, are unsuitable for decoding (see Figure 2.3).

Another important property lies in the structure of the Tanner graph. As we will see in Chapter 3, the decoding algorithms are optimal when applied to realizations with cycle-free Tanner graphs. For realizations with cycles in the Tanner graphs, very little is actually known about the decoding performance; however, most indications are that it is beneficial to avoid short cycles. (This will be discussed later.)

So why do we consider realizations with cycles at all, when the decoding algorithms are optimal for cycle-free realization? The advantage of introducing cycles is that the structural complexity may be much smaller in such realizations, allowing for a smaller decoding complexity. What happens, roughly, is that a single trellis state space of size m is split into a number of hidden sites of size m_s such that $\prod_s m_s \approx m$. This will be discussed in Chapter 7, where we will continue the discussion of graph-based realizations. Here we just give one further example of a realization with many cycles in the Tanner graph, an example that deserves special attention.

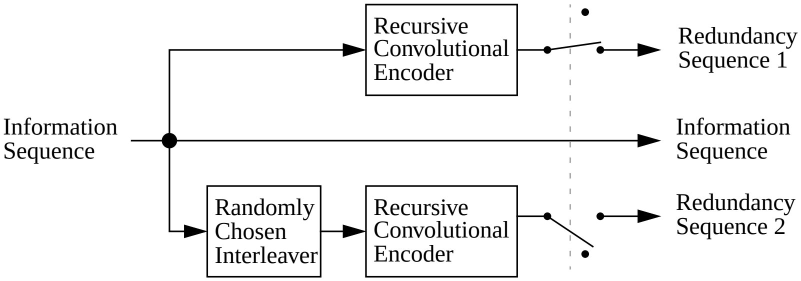

Recursive Convolutional Encoder
Redundancy Sequence 1
Information Sequence
Information Sequence
Randomly Chosen Interleaver
Recursive Convolutional Encoder
Redundancy Sequence 2

Figure 2.4 *The turbo codes as presented by Berrou et al* [7].

2.2 Turbo Codes

The turbo codes of Berrou et al. [7] are famous for their amazing performance, which beats anything else that has been presented so far. Unfortunately, the performance has only been demonstrated by simulations, and not by theoretical results. In particular, the decoding algorithm proposed in [7] remains to be analyzed (the theoretical code performance was analyzed to some extent in [8]).

The conventional way of presenting turbo codes is to describe the encoder. As illustrated in Figure 2.4, the information sequence is encoded twice by the same recursive systematic convolutional encoder, in one case with the original symbol ordering and in the other case after a random interleaving of the information symbols. (At the output, the redundancy sequences are often punctured in order to achieve the overall rate $^1/_2$.) The convolutional encoders are usually rather simple; the typical encoder memory is 4 (i.e., there are 16 states in the trellis).

The corresponding Tanner graph (Figure 2.5) makes the structure somewhat more apparent, consisting of two trellises that share certain output symbols via an interleaver (i.e., the order of the common symbols in one trellis is a permutation of the order in the other trellis). It is well known (cf. [8]) that the amazing performance of turbo codes is primarily due to the interleaver, i.e., due to the cycle structure of the Tanner graph.

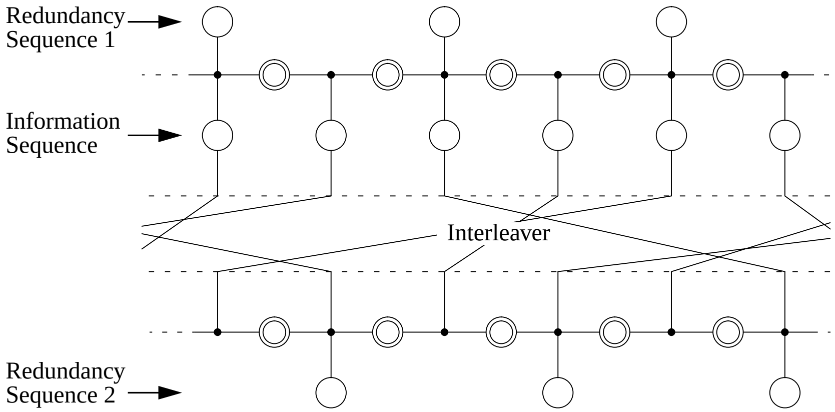


Figure 2.5 *The Tanner graph of the turbo codes.*

Chapter 3

The Min-Sum and Sum-Product Algorithms

We will describe two generic decoding algorithms for code realizations based on Tanner graphs, as described in the previous chapter. The structure of the algorithms matches the graphs directly. It will be convenient to think of these algorithms as parallel processing algorithms, where each site and each check is assigned its own processor and the communication between them reflects the Tanner graph. (In fact, this "distributed" viewpoint was one of the motivations for developing the framework. However, in many cases a sequential implementation is actually more natural.)

The algorithms come in two versions: the min-sum algorithm and the sum-product algorithm. The ideas behind them are not essentially new; rather, the algorithms are generalizations of well-known algorithms such as the Viterbi algorithm [2] and other trellis-based algorithms. Another important special case is Gallager's algorithm for decoding low-density parity-check codes [5]. A relatively general formulation of the algorithms was also given by Tanner [6] (the relation between Tanner's work and ours is discussed in Section 1.2 of the introduction).

There are other generic decoding algorithms that apply to our general framework for code descriptions. In a previous work [13] we discussed the application of Gibbs sampling, or simulated annealing for decoding graph-based codes.

The overall structure of the algorithms, and the context in which they apply, is illustrated in Figure 3.1. As shown, the algorithms do not make decisions, instead they compute a set of *final cost functions* upon which a final decision can be made. The channel output enters the algorithms as a set of *local cost functions*, and the goal of the algorithms is to concentrate, for each site, all information from the channel output that is relevant to that site.

Formally, there is one local cost function for each site $s \in N$, denoted by $\gamma_s : A_s \to \mathcal{R}$ (where $\mathcal{R}$ denotes the real numbers), and one for each check set $E \in Q$, denoted by $\gamma_E : W_E \to \mathcal{R}$. Similarly, there is one final cost function for each site $s \in N$, denoted by $\mu_s : A_s \to \mathcal{R}$, and one for each check set, denoted by $\mu_E : W_E \to \mathcal{R}$. (In our applications,

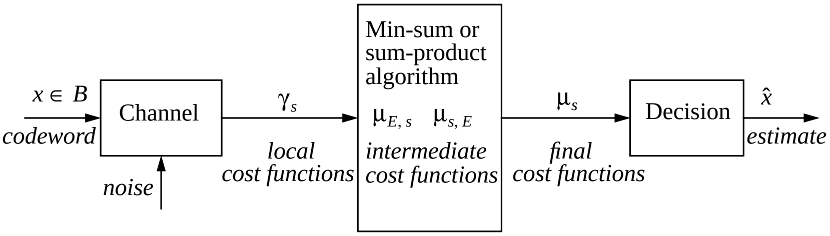


Figure 3.1 *Typical decoding application of the min-sum or sum-product algorithm. The channel output takes the form of* ***local cost functions*** γ_s *("channel metrics") which are used by the min-sum or sum-product algorithm to compute* ***final cost functions*** μ_s*, upon which the final decisions are based. During the computation process, the algorithms maintain a set of intermediate cost functions.*

the check cost functions γ_E and μ_E are often not used; they are most interesting when the codewords are selected according to some non-uniform probability distribution, or, as discussed in Section 7.4, when dealing with channels with memory.)

During the computation, the algorithms maintain a set of intermediate cost functions: for each pair (s, E) of adjacent site and check set (i.e., $s \in E$), there is one check-to-site cost function $\mu_{E,s} : A_s \to \mathcal{R}$ and one site-to-check cost function $\mu_{s,E} : A_s \to \mathcal{R}$. These cost functions are best thought of as having a direction on the Tanner graph. For instance, we will often call $\mu_{E,s}$ the "contribution" from the check set E to the site s. (In the cycle-free case, this will be given a precise interpretation.) See Figure 3.2.

3.1 The Min-Sum Algorithm

The min-sum algorithm is a straightforward generalization of the Viterbi algorithm [2]. (The resulting algorithm is essentially Tanner's Algorithm B [6]; Tanner did not, however, observe the connection to Viterbi decoding.) Hagenauer's low-complexity turbo decoder [14] fits directly into this framework. A well-known decoding algorithm for generalized concatenated codes [15] is also related, as is threshold decoding [16]. Before going into the general description, we encourage the reader to go through the example in Figure 3.3 on pages 14–15, where the decoding of a (7,4,2) binary linear code using the min-sum algorithm is performed in detail.

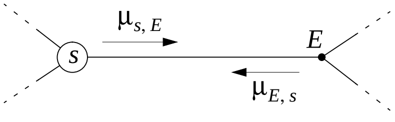
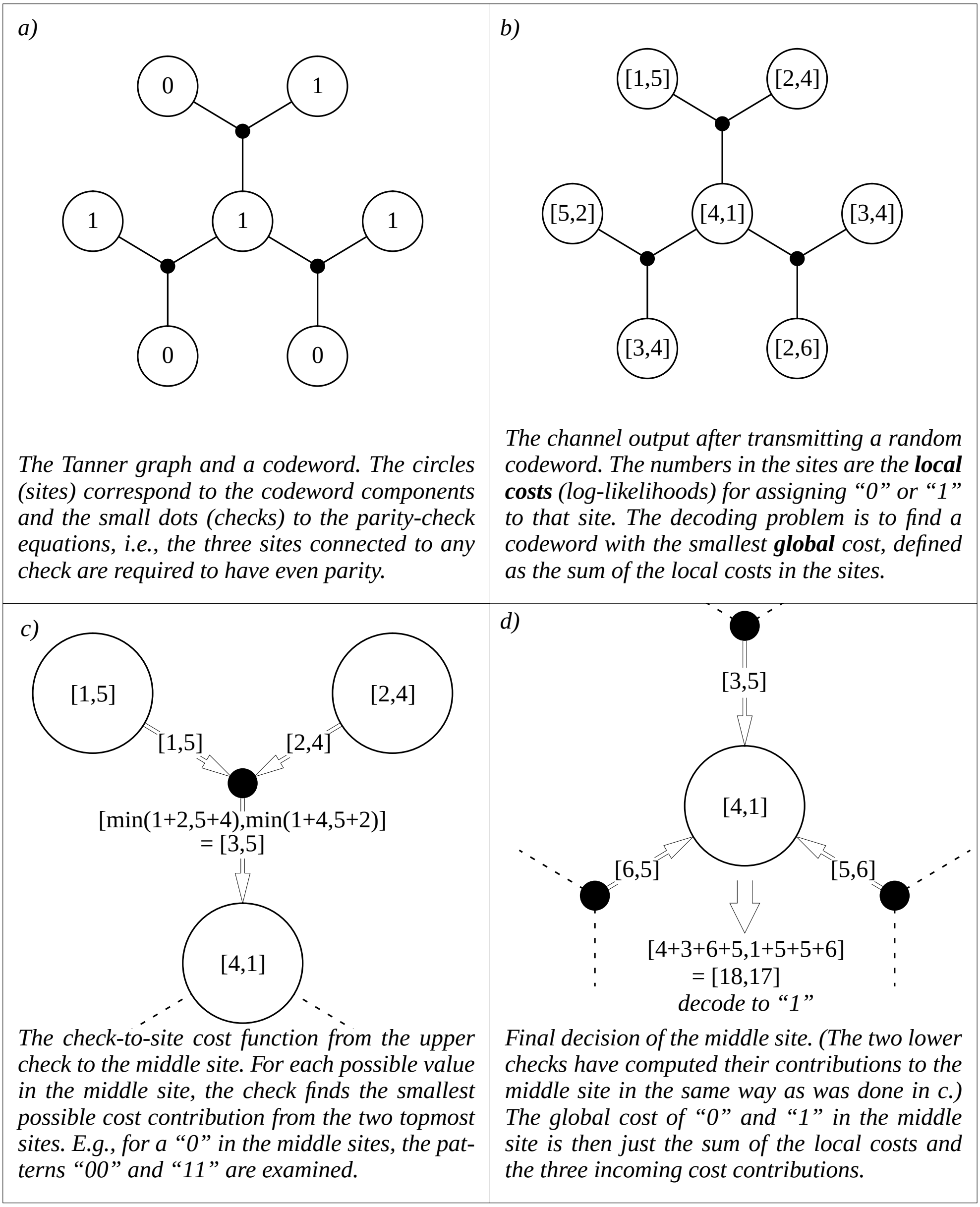


Figure 3.2 *The intermediate cost functions* $\mu_{s,E} : A_s \to \mathcal{R}$ *and* $\mu_{E,s} : A_s \to \mathcal{R}$.



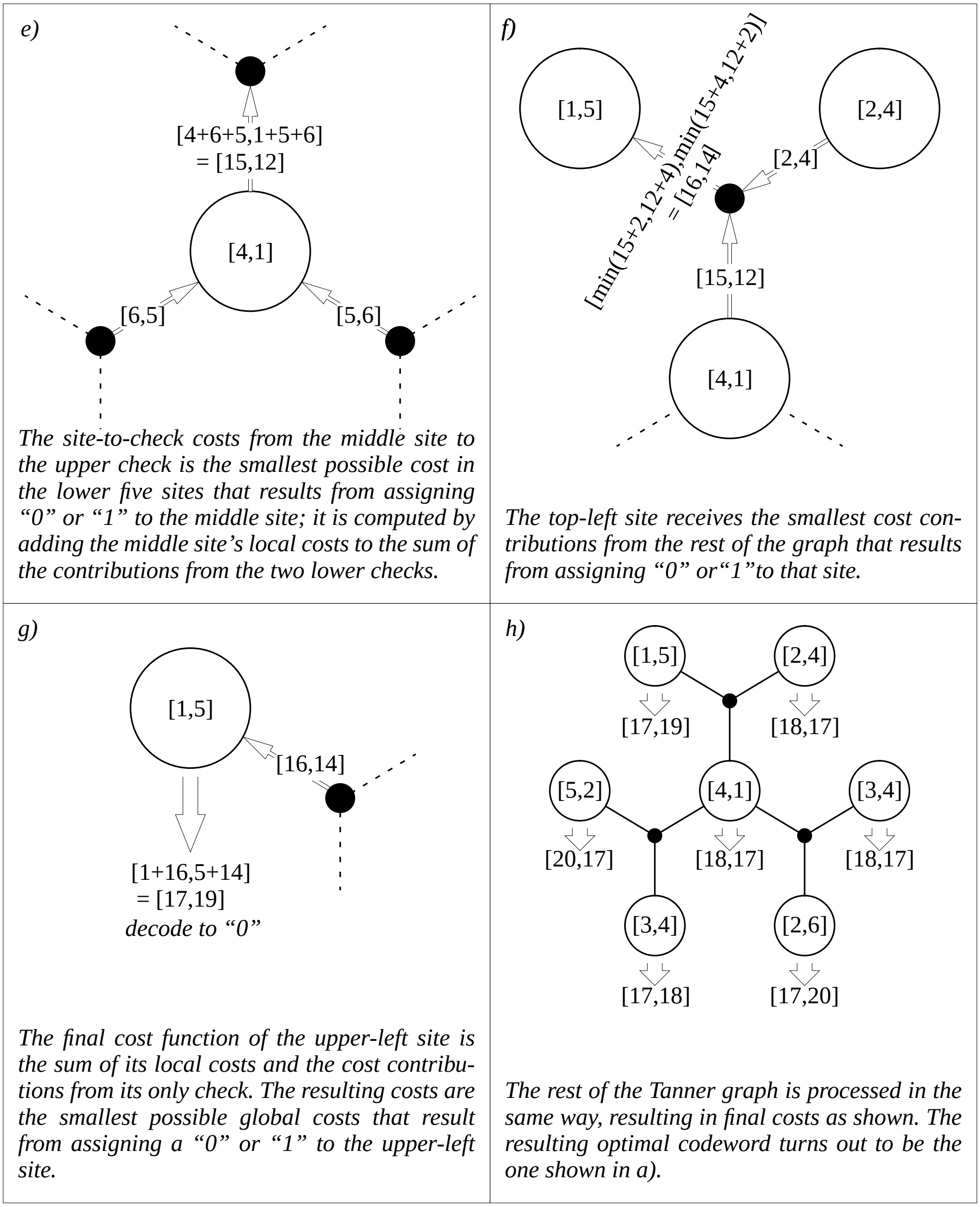
The Tanner graph and a codeword. The circles (sites) correspond to the codeword components and the small dots (checks) to the parity-check equations, i.e., the three sites connected to any check are required to have even parity.

*The channel output after transmitting a random codeword. The numbers in the sites are the **local costs** (log-likelihoods) for assigning "0" or "1" to that site. The decoding problem is to find a codeword with the smallest **global** cost, defined as the sum of the local costs in the sites.*

The check-to-site cost function from the upper check to the middle site. For each possible value in the middle site, the check finds the smallest possible cost contribution from the two topmost sites. E.g., for a "0" in the middle sites, the patterns "00" and "11" are examined.

Final decision of the middle site. (The two lower checks have computed their contributions to the middle site in the same way as was done in c.) The global cost of "0" and "1" in the middle site is then just the sum of the local costs and the three incoming cost contributions.

Figure 3.3 *The min-sum algorithm applied to a binary linear (7,4,2) code, whose Tanner graph is shown in a). The decoding problem is to find the codeword with the smallest **global cost** (or "metric"), defined as the sum over the codeword components of the corresponding **local** costs, which are indicated in b). The local costs are typically channel log-likelihoods (such as Hamming distance or squared Euclidean distance to the received values).*



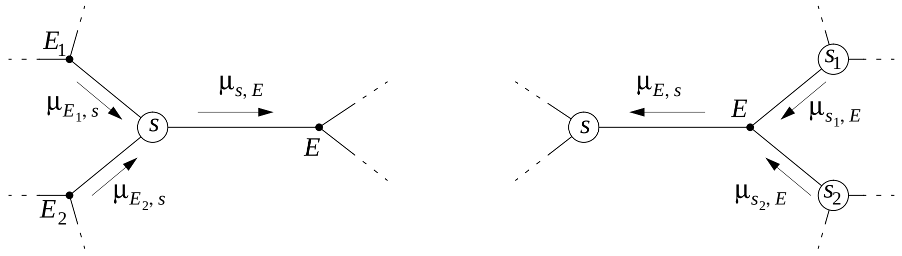
The site-to-check costs from the middle site to the upper check is the smallest possible cost in the lower five sites that results from assigning “0” or “1” to the middle site; it is computed by adding the middle site’s local costs to the sum of the contributions from the two lower checks.

The top-left site receives the smallest cost contributions from the rest of the graph that results from assigning “0” or“1”to that site.

The final cost function of the upper-left site is the sum of its local costs and the cost contributions from its only check. The resulting costs are the smallest possible global costs that result from assigning a “0” or “1” to the upper-left site.

The rest of the Tanner graph is processed in the same way, resulting in final costs as shown. The resulting optimal codeword turns out to be the one shown in a).

Figure 3.3 *(continued) Boxes c) –g) illustrate the computation of the intermediate and final cost functions for a few of the sites. In h), the final cost functions of all sites are shown.*



$$\mu_{s,E}(a) \ := \ \gamma_s(a) + \mu_{E_1,s}(a) + \mu_{E_2,s}(a) \qquad \mu_{E,s}(a) \ := \ \min_{\substack{x_E \in B_E \\ x_s = a}} [\gamma_E(x_E) + \mu_{s_1,E}(x_{s_1}) + \mu_{s_2,E}(x_{s_2})]$$

Figure 3.4 *The updating rules for the min-sum algorithm.*

As discussed in the example (Figure 3.3), the goal of the min-sum algorithm is to find a valid configuration $x \in B$ such that the sum of the local costs (over all sites and check sets) is as small as possible. When using the min-sum algorithm in a channel-decoding situation with a memoryless channel and a received vector y, the local check costs $\gamma_E(x_E)$ are typically omitted (set to zero) and the local site costs $\gamma_s(x_s)$ are the usual channel log-likelihoods $-\log p(y_s|x_s)$ (for visible sites; for hidden sites they are set to zero). On the binary symmetric channel, for example, the local site cost $\gamma_s(x_s)$ would be the Hamming distance between x_s and the received symbol y_s.

The algorithm consists of the following three steps:

- *Initialization*. The local cost functions γ_s and γ_E are initialized as appropriate (using, e.g., channel information). The intermediate cost functions $\mu_{E,s}$ and $\mu_{s,E}$ are set to zero.
- *Iteration*. The intermediate cost functions $\mu_{s,E}$ and $\mu_{E,s}$ are alternatingly updated a suitable number of times as follows (cf. also Figure 3.4). The site-to-check cost $\mu_{s,E}(a)$ is computed as the sum of the site's local cost and all contributions coming into s except the one from E:

$$\mu_{s,E}(a) \ := \ \gamma_s(a) + \sum_{\substack{E' \in Q: \\ s \in E',\, E' \neq E}} \mu_{E',s}(a)\,. \tag{3.1}$$

The check-to-site cost $\mu_{E,s}(a)$ is obtained by examining all locally valid configurations on E that match a on the site s, for each summing the check's local cost and all contributions coming into E except the one from s. The minimum over these sums is taken as the cost $\mu_{E,s}(a)$:

$$\mu_{E,s}(a) \ := \ \min_{x_E \in B_E \,:\, x_s = a} \left\lfloor \gamma_E(x_E) + \sum_{s' \in E \,:\, s' \neq s} \mu_{s',E}(x_{s'}) \right\rfloor. \tag{3.2}$$

- *Termination*. The final cost functions μ_s and μ_E are computed as follows. The final site cost $\mu_s(a)$ is computed as the sum of the site's local cost and all contributions coming into *s*, i.e.,

$$\mu_s(a) := \gamma_s(a) + \sum_{E' \in Q : s \in E'} \mu_{E',s}(a), \tag{3.3}$$

and the final check cost $\mu_E(a)$ for a local configuration $a \in B_E$ is computed as the sum of the check's local cost and all contributions coming into *E*, i.e.,

$$\mu_E(a) := \gamma_E(a) + \sum_{s' \in E} \mu_{s',E}(a_{s'}). \tag{3.4}$$

As we mentioned above, the goal of the min-sum algorithm is to find a configuration with the smallest possible cost sum. To formulate this precisely, we define the *global cost* of a valid configuration $x \in B$ as

$$G(x) \triangleq \sum_{E \in Q} \gamma_E(x_E) + \sum_{s \in N} \gamma_s(x_s). \tag{3.5}$$

In a typical channel-decoding situation where the local check costs $\gamma_E(x_E)$ are set to zero and the local site costs are $\gamma_s(x_s) \triangleq -\log p(y_s | x_s)$, the global cost $G(x)$ becomes the log-likelihood $-\log p(y|x)$ for the codeword *x*; then maximum-likelihood decoding corresponds to finding a valid configuration $x \in B$ that minimizes $G(x)$. As we will see, the min-sum algorithm does this minimization when the check structure is cycle-free. In addition, it is also possible to assign nonzero values to the check costs $\gamma_E(x_E)$ in order to include, e.g., an a priori distribution $p(x)$ over the codewords: if we define the local check costs γ_E such that $-\log p(x) = \sum_{E \in Q} \gamma_E(x_E)$, then the global cost will be $G(x) \triangleq -\log p(x) - \log p(y|x) = -\log p(x|y) - \log p(y)$, and minimizing $G(x)$ will be equivalent to maximizing the a posteriori codeword probability $p(x|y)$. (See the next section for more about a priori probabilities.)

The following theorem is the fundamental theoretical property of the min-sum algorithm:

Theorem 3.1 If the check structure is finite and cycle-free, then the cost functions converge after finitely many iterations, and the final cost functions become

$$\mu_s(a) = \min_{x \in B : x_s = a} G(x) \tag{3.6}$$

and

$$\mu_E(a) = \min_{x \in B : x_E = a} G(x). \tag{3.7}$$

(The proof is given in Appendix A.1.) For Tanner graphs that contain cycles, there is no general result for the final cost functions, or for the decoding performance. This issue is discussed in the following chapters.

As we mentioned earlier, the min-sum algorithm only computes the final costs; no decision is made. Usually, we also want to find a configuration that minimizes the global cost. Such a configuration is obtained by taking, for each site *s*, a value $x_s \in A_s$ that minimizes the final cost $\mu_s(x_s)$. It may then happen that for some sites several values minimize the final cost; then it may be a nontrivial problem to find a valid configuration that minimizes μ_s at all sites. For a cycle-free check structure, however, there is a straightforward procedure for solving this problem: start in a leaf site *s* (one that belongs to a single check set) and choose an optimal value for x_s; then extend the configuration successively to neighboring sites, always choosing site values that are both valid and minimize the final cost.

In a practical implementation it is important to handle numerical issues properly. Typically, the cost functions $\mu_{E,\,s}$ and $\mu_{s,\,E}$ grow out of range quickly. To overcome this, an arbitrary normalization term may be added to the updating formulas without affecting the finally chosen configuration. Since the algorithm only involves addition and minimization (i.e., no multiplication), fixed precision arithmetic can be used without losing information (the only place where precision is lost is in the initial quantization of the local costs).

3.2 The Sum-Product Algorithm

The sum-product algorithm is a straightforward generalization of the forward-backward algorithm of Bahl et al [17] for the computation of per-symbol a posteriori probabilities in a trellis. Two other special cases of the sum-product algorithm are the classic turbo decoder by Berrou et al. [7] and Gallager's decoding algorithms for low-density parity-check codes [5]. The general case was outlined by Tanner [6], who did not, however, consider a priori probabilities.

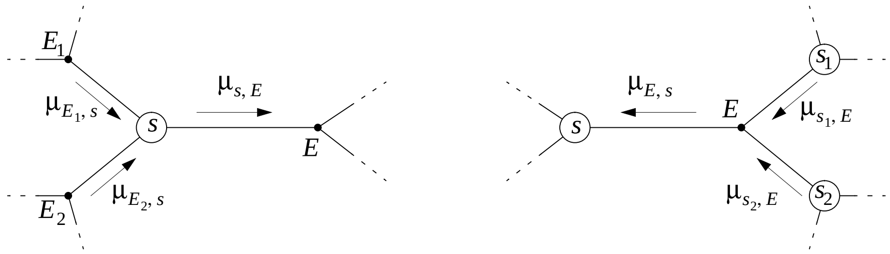
In the sum-product algorithm, the local cost functions γ_s and γ_E have a multiplicative interpretation: we define the global "cost" for any configuration $x \in W$ as the product

$$G(x) \triangleq \prod_{E \in Q} \gamma_E(x_E) \prod_{s \in N} \gamma_s(x_s). \tag{3.8}$$

The term "cost" is somewhat misleading in the sum-product algorithm, since it will usually be subject to maximization (rather than minimization as in the min-sum algorithm); we have chosen this term to make the close relation between the two algorithms transparent. The algorithm does not maximize *G* directly; it merely computes certain "projections" of *G*, which in turn are natural candidates for maximization.

When discussing the sum-product algorithm, it is natural not to consider the behavior *B* explicitly, but to instead require that the check costs $\gamma_E(x_E)$ are zero for local configurations that are non-valid and positive otherwise. Hence, we require

$$\gamma_E(x_E) \geq 0 \quad \text{with equality if and only if } x_E \notin B_E\,. \tag{3.9}$$



$$\mu_{s,E}(a) := \gamma_s(a)\,\mu_{E_1,s}(a)\,\mu_{E_2,s}(a) \qquad\qquad \mu_{E,s}(a) := \sum_{\substack{x_E \in W_E \\ x_s = a}} \gamma_E(x_E)\,\mu_{s_1,E}(x_{s_1})\,\mu_{s_2,E}(x_{s_2})$$

Figure 3.5 *The updating rules for the sum-product algorithm.*

If we also require the local site costs $\gamma_s(x_s)$ to be strictly positive for all x_s, then it is easy to see that the global cost (3.8) of a configuration x is strictly positive if x is valid, and zero otherwise. In particular, a configuration that maximizes G is always valid (provided that B is nonempty).

In the typical channel-decoding situation, with a memoryless channel and a received vector y, the local site costs $\gamma_s(x_s)$ are set to the channel likelihoods $p(y_s|x_s)$ (for visible sites; for hidden sites γ_s is set to one), and the local check costs are chosen according to the a priori distribution for the transmitted configuration x, which must be of the form $p(x) = \prod_{E \in Q} \gamma_E(x_E)$. This form includes Markov random fields [18], Markov chains, and, in particular, the uniform distribution over any set of valid configurations. (The latter is achieved by taking γ_E as the indicator function for B_E, with an appropriate scaling factor.) With this setup, we get $G(x) = p(x)p(y|x) \propto p(x|y)$, i.e., $G(x)$ is proportional to the a posteriori probability of x. We will see later that if the check structure is cycle-free, the algorithm computes the a posteriori probability for individual site (symbol) values $p(x_s|y) = \sum_{x' \in B : x'_s = x_s} G(x')$, which can be used to decode for minimal symbol error probability.

The sum-product algorithm consists of the following three steps:

- *Initialization*. The local cost functions γ_s and γ_E are initialized as appropriate (using, e.g., channel information and/or some known a priori distribution). The intermediate cost functions $\mu_{E,s}$ and $\mu_{s,E}$ are set to one.
- *Iteration*. The intermediate cost functions $\mu_{E,s}$ and $\mu_{s,E}$ are updated a suitable number of times as follows (cf. also Figure 3.5). The site-to-check cost $\mu_{s,E}(a)$ is computed as the product of the site's local cost and all contributions coming into s except the one from E:

$$\mu_{s,E}(a) := \gamma_s(a) \prod_{\substack{E' \in Q: \\ s \in E', E' \neq E}} \mu_{E',s}(a). \tag{3.10}$$

The check-to-site cost $\mu_{E,s}(a)$ is obtained by summing over all local configurations on E that match a on the site s, each term being the product of the check's local cost and all contributions coming into E except the one from s:

$$\mu_{E,s}(a) := \sum_{x_E \in W_E : x_s = a} \gamma_E(x_E) \prod_{s' \in E : s' \neq s} \mu_{s',E}(x_{s'}). \tag{3.11}$$

Note that the sum in (3.11) actually only runs over B_E (the locally valid configurations) since $\gamma_E(x_E)$ is assumed to be zero for $x_E \notin B_E$.

- *Termination.* The final cost functions μ_s and μ_E are computed as follows. The final site cost $\mu_s(a)$ is computed as the product of the site's local cost and all contributions coming into s, i.e.,

$$\mu_s(a) := \gamma_s(a) \prod_{E' \in Q : s \in E'} \mu_{E',s}(a), \tag{3.12}$$

and the final check cost $\mu_E(a)$ is computed as the product of the check's local cost and all contributions coming into E, i.e.,

$$\mu_E(a) := \gamma_E(a) \prod_{s' \in E} \mu_{s',E}(a_{s'}). \tag{3.13}$$

The fundamental theoretical result for the sum-product algorithm is the following:

Theorem 3.2 If the check structure is finite and cycle-free, then the cost functions converge after finitely many iterations and the final cost functions become

$$\mu_s(a) = \sum_{x \in B : x_s = a} G(x) \tag{3.14}$$

and

$$\mu_E(a) = \sum_{x \in B : x_E = a} G(x). \tag{3.15}$$

(The proof is essentially identical with that of Theorem 3.1, which is given in Appendix A.1.) An important special case of Theorem 3.2 is when the cost functions correspond to probability distributions:

Corollary 3.3 If the global cost function $G(x)$ is (proportional to) some probability distribution over the configuration space, then the final cost functions are (proportional to) the corresponding marginal distributions for the site

> values and the local configurations. In particular, if $G(x)$ is proportional to the a posteriori probability $p(x|y)$, then the final cost μ_s is proportional to the per-symbol a posteriori probability $p(x_s|y)$ and the final cost μ_E is proportional to the per-check a posteriori probability $p(x_E|y)$.

As with the min-sum algorithm, there is no guarantee for the decoding performance when the Tanner graph contains cycles. (We will come back to this case in later chapters.)

In a decoding (estimation) application, an estimate for the site value x_s is obtained by choosing the value x_s that maximizes the final cost $\mu_s(x_s)$. With a cycle-free check structure, this minimizes the probability of symbol error.

As with the min-sum algorithm, it is important to handle numerical issues properly. Typically, it is necessary to include a normalization factor in the updating formulas in order to prevent the costs from going out of numerical range. This normalization factor does not influence the final maximization.

3.3 Updating Order and Computation Complexity

So far, we have only described the formulas that are used to compute the cost functions, and not in which order they should be computed. In the introduction to this chapter, we mentioned that it is convenient to view the algorithms as “parallel”, with one processing unit for each site and each check. Then all check-to-site cost functions would be updated simultaneously, and similarly for the site-to-check cost functions. This updating order is used in Gallager’s sum-product decoding algorithm for low-density parity-check codes.

With a cycle-free check structure, however, each intermediate cost function need only be computed once, namely when all its incoming cost functions have been computed. Not surprisingly, trellis-based algorithms (such as the forward-backward algorithm [17] and the soft-output Viterbi algorithm [19]) use “sequential” updating orders, where the entire trellis is processed first from left to right and then from right to left, thereby updating each cost function exactly once. It is possible to find such efficient updating orders for any cycle-free check structure in the following way. Select any site as the “root” site. Then start updating the cost functions at the leaves, working towards the root site, but always finishing all the subgraphs leading into a check or site before proceeding with its outgoing cost function. For each site and check, only the cost function pointing towards the root is updated. After the root site is reached (from all directions), the updating process proceeds “outwards” again.

With the min-sum algorithm and a cycle-free realization, the “outwards” updating phase may be avoided and replaced by a simpler backtracking procedure, if we only want to find a lowest-cost realization. This requires that we remember in the first phase what particular local configurations were optimal for each check. This is exactly what is done in the traditional version of the Viterbi algorithm [2] (i.e., not the soft-output version), when the best incoming branch is stored for each state in a section.

Even with a check structure that is not cycle-free, certain updating orders may be more natural (or efficient) than others. In the classic turbo decoder [7] (cf. Figure 2.5), for example, the two trellises are processed alternatingly in a forward-backward manner, and the connected sites are updated between processing of the two trellises.

From the updating rules of the min-sum and sum-product algorithms, it is relatively clear that the computation complexity is closely related to the size of the site alphabets and the local behaviors, i.e., the *realization complexity.* For cycle-free realizations, we have to compute either all cost functions once, or (with the min-sum algorithm) only the cost functions that point towards some “root” site. For realizations with cycles, each cost function will generally be computed several times, so the number of operations depends on the number of decoding iterations.

In Appendix A.2, a detailed analysis of the number of binary operations (such as addition or multiplication of two numbers, or finding the smallest of two numbers) is performed for these two cases. The number of operations needed to compute each cost function once is

$$\sum_{E \in Q} |B_E|(|E|^2 - |E|) + \sum_{s \in V} |A_s|(2|s| - 5) + \sum_{s \in N \setminus V} |A_s|(2|s| - 6) \ , \tag{3.16}$$

where $|s|$ is the number of check sets that s belongs to, and V is the set of visible sites. The number of operations needed to compute the cost functions pointing towards a root site is

$$\sum_{s \in I} |A_s||s| + \sum_{s \in H} |A_s|(|s| - 1) + \sum_{E \in Q} |B_E|(|E| - 1) + |A_r| - 1 \ , \tag{3.17}$$

where H are the hidden sites, I are the interior sites (not leaves), and r is the root site.

Naturally, both the alphabet sizes $|A_s|$ and the size of the local behaviors $|B_E|$ influence the computation complexity. However, many relatively large local behaviors can be handled efficiently with “smart” updating rules, obtaining a lower complexity; a typical example is the parity check. This is issue is discussed in Appendix A.2, where we point out that such smart updating rules can often be viewed as working on a *refined* realization, where a check is replaced by a few sites and checks, and applying the straightforward updating rules to this realization. The resulting complexity is then given by (3.16) or (3.17), but applied to the refined realization. However, some smart updating rules cannot be viewed conveniently in this way, as we will see in the next section.

3.4 Optimized Binary Version of the Min-Sum Algorithm

We have mentioned that normalization is important to keep the costs within available numerical range, and that such a normalization (additive in the min-sum case and multiplicative in the sum-product case) does not influence more than an overall constant in the final costs. In effect, this means that a “q-ary” cost function actually only has $q - 1$ degrees of freedom. This fact can be exploited to simplify the computations, by representing the cost functions with only $q - 1$ values. For binary site alphabets and parity checks, the savings obtained in this way are significant, as shown already by Gallager [5] for the sum-product algorithm. Here, we derive a similar simplification for the min-sum algorithm.

Since we are now only interested in the difference between the “1” cost and the “0” cost, we will use a simplified notation for cost functions. The local cost difference $\gamma_s(1) - \gamma_s(0)$ at the site s will be denoted by γ_s, and the final cost difference $\mu_s(1) - \mu_s(0)$ will be denoted by μ_s. The same simplification will be used for the intermediate cost functions. Then it is easy to see that the updating rule (3.1) for the site-to-check costs $\mu_{s,E}$ may be expressed as

$$\mu_{s,E} = \gamma_s(1) - \gamma_s(0) + \sum_{\substack{E' \in Q: \\ s \in E', E' \neq E}} [\mu_{E',s}(1) - \mu_{E',s}(0)] \tag{3.18}$$

$$= \gamma_s + \sum_{\substack{E' \in Q: \\ s \in E', E' \neq E}} \mu_{E',s}, \tag{3.19}$$

and, similarly, the updating rule (3.3) for the final cost functions μ_s may be written as

$$\mu_s = \gamma_s + \sum_{E \in Q : s \in E} \mu_{E,s}. \tag{3.20}$$

Finally, the updating rule for the check-to-site cost functions has the form

$$\mu_{E,s} = \prod_{s' \in E : s' \neq s} [\text{sign}(\mu_{s',E}) \min_{s' \in E : s' \neq s} |\mu_{s',E}|]. \tag{3.21}$$

The derivation behind (3.21) can be found in Appendix A.3.

3.5 Non-Decoding Applications

Although our main focus is on decoding, it is clear that the min-sum and sum-product algorithms have more general applications (this is one reason why we used the term “cost function” instead of the more common “metric”). This is perhaps most obvious for the min-sum algorithm (applied to cycle-free Tanner graphs), which is used a lot in optimization, where it is referred to as “dynamic programming”, cf. e.g. [20].

A useful non-decoding application of the sum-product algorithm is to compute weight distributions. Incidentally, we will use the sum-product algorithm for this purpose in Chapter 5. The idea is to compute the weight enumerator function by performing the sum-product algorithm *symbolically*.

Consider a system with binary-only visible sites and a cycle-free check structure. Let all local costs equal 1 except for the visible sites where $\gamma_s(1) = w$ and $\gamma_s(0) = 1$. Then, from (3.8), the global cost of a configuration x is $G(x) = w^{\text{weight}(x)}$. By applying the sum-product algorithm and using Theorem 3.2, the final cost $\mu_s(a)$ is (after a suitable number of updates) the weight enumerator for the subset of B with $x_s = a$. Finally, the weight enumerator for the whole behavior B is obtained as $\sum_{a \in W_s} \mu_s(a)$.

3.6 Further Unifications

The reader may have noted that (for both algorithms presented) the updating formulas for the site-to-check cost functions, on the one hand, and the check-to-site cost functions, on the other, are very similar (see, for example, Equation 3.1 vs. 3.2). In fact, sites could have been treated as a special kind of checks, which would have formally unified the update formulas. It seems, however, that the present notation is more natural and easier to understand.

It is also possible to unify (and possibly generalize) the min-sum and sum-product algorithms formally, using an "universal algebra" approach with two binary operators $\oplus$ and $\otimes$. For the sum-product algorithm the operator $\oplus$ is addition and the operator $\otimes$ is multiplication, while for the min-sum algorithm $\oplus$ is minimization (taking the smallest of two numbers) and $\otimes$ is addition. The theorems 3.1 and 3.2 have a counterpart in this universal setting provided that the operators $\oplus$ and $\otimes$ are associative and commutative, and that the operator $\otimes$ is distributive over $\oplus$, i.e., $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$. The proof for such a general theorem is readily obtained by substituting $\oplus$ for "min" and $\otimes$ for "+" in the proof of Theorem 3.1, cf. Appendix A.1.

Chapter 4

Analysis of Iterative Decoding

Iterative decoding is a generic term for decoding algorithms whose basic operation is to modify some internal state in small steps until a valid codeword is reached. In our framework, iterative decoding occurs when the min-sum or sum-product algorithm is applied to a code realization with cycles in the Tanner graph. (There are other iterative decoding methods too, cf. e.g. our previous work [13].) As opposed to the cycle-free case, there is no obvious termination point: even when the cost contributions from each site have reached all other sites, there is no guarantee that this information is used optimally in any sense. Typically, the updating process is continued after this point, updating each intermediate cost function several times before computing the final costs and making a decision.

Since the theorems of Chapter 3 do not apply when there are cycles in the Tanner graph, it is somewhat surprising that, in fact, iterative decoding often works quite well. This is primarily demonstrated by simulation results, the most well-known being the turbo codes of Berrou et. al. [7]. A much older (and lesser known) indication of this is Gallager's decoding method for low-density parity-check codes [5].

A natural approach to analyzing iterative decoding is simply to disregard the influence of the cycles. In fact, if the decoding process is terminated after only a few decoding iterations, the algorithm will have operated on cycle-free subgraphs of the Tanner graph; consequently, the theorems of Chapter 3 apply to these "subgraph codes", a fact that can be used to estimate the decoding performance. Put another way, the intermediate cost functions coming into each site (or check) are statistically independent as long as the cycles have not "closed".

One aspect of this approach is to consider the performance obtained after infinitely many cycle-free decoding iterations. In particular, the error probability may in some cases tend to zero when the number of decoding iterations increases. Of course, the number of cycle-free decoding iterations for any fixed realization is limited by the length of the shortest cycles (the *girth*) of the Tanner graph, so such an analysis is strictly applicable only to infinite realization families having asymptotically unbounded girth. By considering such families, however, it is possible to construct coding systems with arbitrarily low probability of decoding error for a given channel.

On the other hand, even if the number of decoding iterations is too large for a strict application of this analysis, the initial influence of the cycles is probably relatively small (in fact, Gallager states in [5] that "the dependencies have a relatively minor effect and tend to cancel each other out somewhat"); in any case, it appears highly unlikely that the actual performance would be significantly *better* than the estimate obtained from such a cycle-free analysis.

Chapter 5 is devoted to performance analyses based on cycle-free subgraphs.

For any fixed code, the decoding performance is limited by the theoretical code performance obtained with an optimal decoder. Ideally, an iterative decoding algorithm would approach this performance when the number of iterations increases. Very little is actually known about the asymptotic behavior of iterative decoding for fixed codes. Our results in this area, along with a deeper discussion of the problem, are presented in Chapter 6.

The rest of this chapter contains some fundamental observations that are used in both Chapter 5 and Chapter 6. In particular, we generalize the concept of trellis "detours", (i.e., paths that diverge from the all-zero path exactly once) to the case of arbitrary realizations. Most of the analysis applies only to the min-sum algorithm, since we have found its operation easier to characterize. For further simplification, we will also assume the following:

- The code is binary and linear, and all codewords are equally probable. In our setting, this implies that the local check costs can be neglected, and that there are site costs on binary sites only (we do allow nonbinary *hidden* sites, though, including trellis state spaces).
- The channel is stationary, memoryless, and has binary input; the transition probabilities are symmetric (i.e., $p(\gamma_s(0)|x_s = 0) = p(\gamma_s(1)|x_s = 1)$ and $p(\gamma_s(0)|x_s = 1) = p(\gamma_s(1)|x_s = 0)$).

These restrictions allow us to assume (without losing further generality) that the all-zero codeword was transmitted. Furthermore, we will assume (also with full generality) that the local site cost functions are normalized so that $\gamma_s(0) = 0$ for all s, and use the shorthand notation $\gamma_s \triangleq \gamma_s(1)$. Then the global cost may be written as

$$G(x) = \sum_{s \in N} \gamma_s(x_s) = \sum_{s \in \operatorname{supp}(x)} \gamma_s, \tag{4.1}$$

where $\operatorname{supp}(x)$ is the *support* of x, i.e., the number of nonzero visible sites in x. Under these restrictions, we will be interested in the following problems:

- What combinations of local costs γ_s (error patterns) lead to a decoding error?
- For a given channel, what is the probability of decoding error?

By decoding error, we will usually consider *symbol* error for some site s, i.e., the event $\mu_s(0) \le \mu_s(1)$. This may seem a little odd since we are analyzing the min-sum algorithm rather than the sum-product algorithm. This has been the most fruitful approach, however. In addition, while it is true for cycle-free systems that the min-sum algorithm minimizes the block error probability and the sum-product algorithm minimizes the symbol error probability, this need not be the case for systems with cycles.

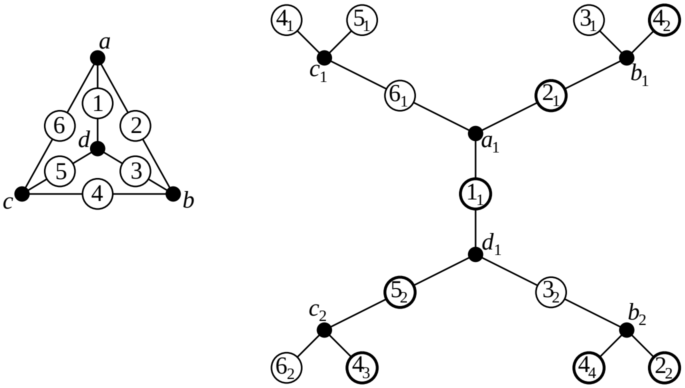


Figure 4.1 *The Tanner graph (left) of a small low-density code and the computation tree (right) leading up to the computation of the final cost function* μ_1*. The sites and check sets in the computation tree have, in this figure, been denoted with subscripts to distinguish between multiple occurrences. The marked sites in the computation tree correspond to the "ones" in a particular valid tree configuration. Note that this configuration does not correspond to a codeword since the site values differ in some cases for tree sites that correspond to the same original site.*

For simplicity, we will use Gallager's low-density parity-check codes as the main example. The discussion will be held on a fairly general level, though, and we will discuss the applicability of our results to trellises and turbo codes, too.

4.1 The Computation Tree

Assume that either the min-sum or the sum-product algorithm is applied to a system (N, W, B) with the check structure Q (cycle-free or not). The algorithm is run as described in the previous chapter: the intermediate cost functions are initialized, the updating rules are applied a suitable number of times (in some order), and the final cost functions are computed.

Consider the final cost function μ_s for some site s. By examining the updatings that have occurred, we may trace recursively back through time the computation of this cost function and all the cost functions which it depends on. This traceback will form a *tree graph* consisting of interconnected sites and checks, in the same way as the original Tanner graph. The site s will be the root of the tree, and if we follow the tree towards the leaves, we will encounter sites and checks with earlier versions of their cost functions; at the leaves, we will encounter the initial cost functions (we will always assume that the leaves are sites (and not checks), i.e., that the updating process starts by computing all site-to-check cost functions). In general, the same sites and checks may occur at several places in the computation tree. The depths in the tree at which the leaves occur depend on the updating order: in the case of parallel updating, the leaves are all of the same depth; otherwise, the depths may be different for different leaves. See Figure 4.1.

Computation trees provide a way to describe the operation of the algorithms by means of a cycle-free graph instead of the original Tanner graph with cycles. More precisely, if the min-sum or the sum-product algorithm were applied directly to the computation tree (with a suitable updating order), then exactly the same operations would occur as when the algorithms are applied to the original Tanner graph. The crucial observation is that the absence of cycles in the computation tree makes it possible to apply the theorems of Section 3.1 and 3.2 to describe the result of the computation.

We will denote the sites and the check structure in the computation tree by $\mathcal{N}$ and $\mathcal{Q}$, respectively, treating possible multiple occurrences distinctly. It follows that there is a mapping $\eta : \mathcal{N} \to N$ such that for any tree site $t \in \mathcal{N}$, $\eta(t)$ is the corresponding site in the original system. There is also a mapping $\theta : \mathcal{Q} \to Q$ such that for any tree check set $F \in \mathcal{Q}$, $\theta(F) \triangleq \{\eta(t) : t \in F\}$ is the corresponding check set in the original Tanner graph. Note that a site (or check set) in the original Tanner graph may in general have multiple corresponding tree sites (or check sets).

Associated with the computation tree is a *tree system* $(\mathcal{N}, \mathcal{W}, \mathcal{B}$ with the site alphabets and local behaviors taken from the original system (N, W, B). More precisely, $\mathcal{W}_t = W_{\eta(t)}$ for all $t \in \mathcal{N}$, and the behavior $\mathcal{B}$ is defined by assuming that $\mathcal{Q}$ is a check structure and that $\mathcal{B}_F = B_{\theta(F)}$ for all $F \in \mathcal{Q}$. The local site and check cost functions are also taken directly from the original system, i.e., $\gamma_t = \gamma_{\eta(t)}$ and $\gamma_F = \gamma_{\theta(F)}$. Finally, we define the *global tree cost function* $\mathcal{G} : \mathcal{W} \to \mathcal{R}$ as

$$\mathcal{G}(u) \triangleq \sum_{F \in \mathcal{Q}} \gamma_F(u_F) + \sum_{t \in \mathcal{N}} \gamma_t(u_t \tag{4.2}$$

for the min-sum algorithm, and for the sum-product algorithm as

$$\mathcal{G}(u) \triangleq \prod_{F \in \mathcal{Q}} \gamma_F(u_F) \prod_{t \in \mathcal{N}} \gamma_t(u_t. \tag{4.3}$$

With these definitions, we have the following corollary of Theorems 3.1 and 3.2.

Corollary 4.1 When the min-sum or sum-product algorithms are applied to any system with a Tanner graph (cycle-free or not), and the corresponding tree system is defined as above, the min-sum algorithm computes

$$\mu_s(a) = \min_{u \in \mathcal{B} : u_s = a} \mathcal{G}(u), \tag{4.4}$$

and the sum-product algorithm computes

$$\mu_s(a) = \sum_{u \in \mathcal{B} : u_s = a} \mathcal{G}(u). \tag{4.5}$$

(Here and henceforth we use the convention that the particular site s refers both to the root site in$\mathcal{N}$ and to the corresponding site in N.)

Proof. The tree system is clearly cycle-free; applying the min-sum or sum-product algorithms to this system gives the final costs above (Theorems 3.1 and 3.2). But the operations performed in that case are exactly the same as when the algorithms are applied to the original system, and hence the results of the computation must be the same. ❑

To summarize this in words: the min-sum and sum-product algorithms always operate as if on a cycle-free Tanner graph, providing optimal decision based on the tree system defined by this graph. If the original Tanner graph is cycle-free, then this tree system will eventually coincide with the original system. In other cases, the tree system may be quite different from the original one, as was illustrated in Figure 4.1.

4.2 The Deviation Set

We now restrict the discussion to the min-sum algorithm. From Corollary 4.1, it is clear that the site s will be decoded incorrectly (i.e., to a “one”) if and only if some valid tree configuration $u \in \mathcal{B}$ with $u_s = 1$ has a lower global cost than all valid tree configurations with $u_s = 0$. Introducing the notations $\mathcal{B}^0 \triangleq \{u \in \mathcal{B} : u_s = 0\}$ and $\mathcal{B}^1 \triangleq \{u \in \mathcal{B} : u_s = 1\}$, we thus have a necessary and sufficient condition for decoding error (at the site s) as

$$\min_{\imath \in \mathcal{B}^0} \mathcal{G}(u) \geq \min_{u \in \mathcal{B}^1} \mathcal{G}(u). \tag{4.6}$$

The condition (4.6) is difficult to handle directly. A considerably simpler *necessary* condition is provided through the concept of *deviations*, which is a direct generalization of trellis detours [21]:

Definition 4.1 Given a tree system $(\mathcal{N}, \mathcal{W}, \mathcal{B})$ rooted at s, a configuration $e \in \mathcal{B}^1$ is called a *minimal valid deviation* or, simply, a *deviation* if it does not cover any other nonzero valid configurations, i.e., if there is no $u \in \mathcal{B}$, $u \neq 0$, such that $\text{support}(u) \subset \text{support}(e)$, where the support is defined over the visible (binary) sites only. We also define the *deviation set* $\mathcal{E}$ of a tree system as the set of all deviations.

Figure 4.2 illustrates two valid tree configurations in $\mathcal{B}^1$, one of which is a deviation. Using the concept of deviations, the condition (4.6) may be simplified (weakened) into:

Theorem 4.2 A necessary condition for a decoding error to occur is that $\mathcal{G}(e) \leq 0$ for some $e \in \mathcal{E}$.

Before proving Theorem 4.2, we introduce another lemma, which states a useful property of the deviation set.

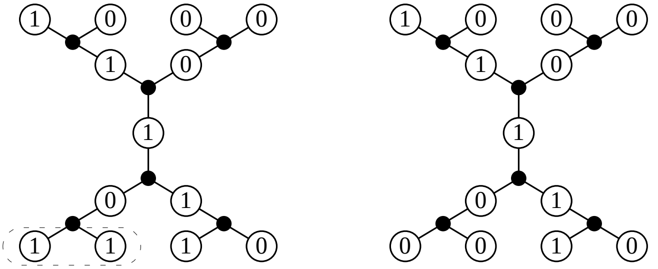


Figure 4.2 *Tree systems for a low-density parity-check code (actually, a cycle code). To the left, a valid configuration in $\mathcal{B}^1$ is shown; however, that configuration is not a* ***deviation****, since it covers a valid configuration (with "ones" only in the marked region). A deviation, shown to the right, is obtained by assigning zeros to those two sites.*

Lemma 4.3 Let $\mathcal{E}$ be the deviation set for a tree system $(\mathcal{N}, \mathcal{W}, \mathcal{B}$. Any configuration $v \in \mathcal{B}^1$ may be decomposed as $v = u + e$, where $u \in \mathcal{B}^0$ and $e \in \mathcal{E}$, and u and e have disjoint support. We will call this a *deviation decomposition* (note, however, that a deviation decomposition need not be unique). We also note that $u \neq 0$ unless $v \in \mathcal{E}$.

Proof of Lemma 4.3. Let $v \in \mathcal{B}^1$. If v is a deviation, then we have $v = 0 + e$, and we are done. On the other hand, if v is not a deviation, then, by Definition 4.1, v covers some nonzero valid configuration x, and we can decompose v as $v = x + y$ where v also covers y, and x and y have disjoint support. Furthermore, since $v_s = x_s + y_s$, exactly one of x and y is in $\mathcal{B}^1$, say $x \in \mathcal{B}^1$. By repeating this decomposition process recursively on x, we get v as a sum of configurations with disjoint support, only one of which is in $\mathcal{B}^1$. The process clearly stops when this configuration is also in $\mathcal{E}$. ❑

Proof of Theorem 4.2. Assume that a decoding error occurred. Then, from (4.6), we know that there is a $\hat{v} \in \mathcal{B}^1$ such that $\mathcal{G}(\hat{v}) \leq \mathcal{G}(u)$ for all $u \in \mathcal{B}^0$. Let $\hat{v} = e + u$ be a deviation decomposition of $\hat{v}$. Then $\mathcal{G}(\hat{v}) = \mathcal{G}(e + u) = \mathcal{G}(e) + \mathcal{G}(u)$, and thus $\mathcal{G}(e) = \mathcal{G}(\hat{v}) - \mathcal{G}(u) \leq 0$. ❑

As an immediate result of Theorem 4.2, we have

Corollary 4.4 The probability of decoding the bit s incorrectly is upper bounded by $\Pr\{\mathcal{G}(e) \leq 0 \text{ for some } e \in \mathcal{E}\}$.

Since Theorem 4.2 provides a necessary condition for decoding error, it is natural to ask whether it is also a sufficient condition. This is not the case: even if $\mathcal{G}(e) \leq 0$ for some $e \in \mathcal{E}$, we may have a $u \in \mathcal{B}^0$ with $\mathcal{G}(u) < \mathcal{G}(e)$.

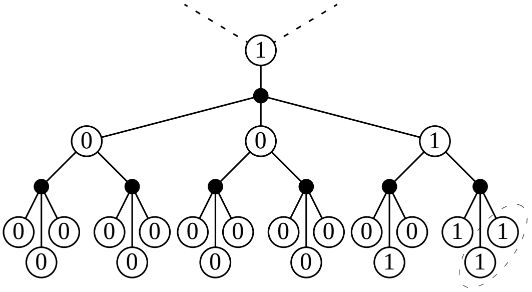


Figure 4.3 *Part of the computation tree for a (3,4) low-density parity-check code. The configuration illustrated is valid. However, it is not a deviation, since it covers a valid configuration with zeros everywhere except, e.g., in the two marked sites. Loosely speaking, there are "too many" ones in that local configuration.*

Example 4.1 (trellises: detours vs. deviations) In a minimal trellis, a *detour* is defined as a path segment that starts in the zero state and ends when it first reaches the zero state again (if ever) [21]. If we consider the computation tree induced by the Viterbi algorithm (recall that the Viterbi algorithm is a special case of the min-sum algorithm), then the detours are just the nonzero parts of the configurations in the deviation set $\mathcal{E}$. ✻

We will be interested in characterizing deviation sets for various code descriptions in order to use Theorem 4.2 and Corollary 4.4 to characterize the error-correcting capability. It turns out that all deviations can be constructed by starting with a nonzero value at the root site (and zeros everywhere else), and extending the support along the branches of the computation tree, but no more than necessary to make the tree configuration valid. Whenever a zero has been assigned to a site (binary or not), the entire subtree emanating from that site will be zero. This process resembles that of constructing detours in a trellis, where we start with a nonzero symbol and extend the path as long as we do not reach the zero state. However, there is a difference between the general case and the special case of detours: in a detour, all valid branches are allowed to follow a nonzero state; in the general case, we do not allow local configurations that may be locally decomposed in a way similar to what is discussed in Lemma 4.3. In other words, a deviation has the "deviation property" not only globally, but also on each check set. See Figure 4.3. This property is formalized in the following lemma.

Lemma 4.5 Let $v \in \mathcal{E}$ be a deviation of the tree system $(\mathcal{N}, \mathcal{W}, \mathcal{B})$ rooted at s. Let $F \in \mathcal{Q}$ be a check set in which the site t is closest to s. Then there does not exist a nonzero $u_F \in \mathcal{B}_F$ with $u_t = 0$ such that $v_F = u_F + e_F$ where $e_F \in \mathcal{B}_F$, and e_F and u_F have disjoint support (in the binary sites).

Proof. Assume to the contrary that there is such a locally valid configuration u_F. Then, by definition of $\mathcal{B}_F$, u_F corresponds to a globally valid configuration u. In particular, by the definition of check structure, u may be taken as equal to v on the subtrees that emanate from F through the nonzero sites of

u_F, and zero everywhere else. But then $u \in \mathcal{B}^0$ and $\mathrm{supp}(u) \subseteq \mathrm{supp}(v)$, contradicting the fact that v is a deviation. ❑

To recapitulate, Lemma 4.5 states that for a given deviation e, all local configurations e_F, $F \in \mathcal{Q}$, look like deviations too. In fact, the converse is also true, i.e., a configuration which looks locally like a deviation on all check sets is in fact a deviation, as formalized in the lemma:

Lemma 4.6 Let $\mathcal{N}$, $\mathcal{W}$, $\mathcal{B}$ be a tree system rooted at $s \in \mathcal{N}$, and let $\mathcal{E}$ be the corresponding deviation set. Let $\mathcal{Q}$ be the check structure for $\mathcal{N}$, $\mathcal{W}$, $\mathcal{B}$. Then $\mathcal{Q}$ is also a check structure for the system $(\mathcal{N}, \mathcal{W}, \mathcal{E}$.

Proof. By definition of a check structure (Definition 2.2 on page 29), we have to show that $v_F \in \mathcal{E}_F$ for all $F \in \mathcal{Q} \Rightarrow v \in \mathcal{E}$. Thus, we assume $v \notin \mathcal{E}$ and proceed to show that there is some $F \in \mathcal{Q}$ for which $v_F \notin \mathcal{E}_F$. Now, if $v \notin \mathcal{B}^1$ then $v_F \notin \mathcal{B}_F^1$ for some F, since $\mathcal{Q}$ is obviously a check structure for $\mathcal{B}^1$, and we are done. So we assume $v \in \mathcal{B}^1$ and write $v = u + e$ where $u \in \mathcal{B}^0$ and $e \in \mathcal{E}$ have disjoint support. Then there is a check set F with the site $t \in F$ closest to the root site s that has $u_t = 0$, but for which $u_F \neq 0$, and we have $v_F = u_F + e_F$. But, according to Lemma 4.5, there is no deviation for which this is true, and thus $v_F \notin \mathcal{E}_F$. ❑

The importance of Lemma 4.6 is twofold. Firstly, it tells us that we may construct all deviations (and only deviations) by starting with a nonzero value at the root site and extending the configuration with local configurations that are local deviations. Secondly, the lemma allows us to use the sum-product algorithm to compute the weight enumerator $T(w)$ for the deviation set $\mathcal{E}$, as discussed in Section 3.5 of Chapter 3; the latter will be useful to provide a union bound for the error probability.

Example 4.2 (deviations for low-density parity-check codes) For parity checks, the nonzero local configurations of the deviations, i.e. the "local deviations" of the checks, are easy to determine (cf. Figure 4.3): apart from the "one" in the innermost site of the check, there is a "one" in exactly one of the other sites. Let e be a deviation in the computation tree rooted at s of a (j, k) low-density parity-check code. The "one" at s gives rise to exactly one additional "one" in each of the j parity check sets that s belongs to, and each of these new "ones" gives rise to $j-1$ new "ones", and so on. Each such required "one" may be placed in any of the $k-1$ available sites of the check set. Thus, the support of e forms a regular subtree of the computation tree, extending out to the leaves. The root site has j children in this subtree and the other sites have $j-1$ children each. For a computation tree of depth m, i.e., after m complete updates, the number of "ones" in the deviation is $\mathrm{weight}(e) = 1 + j\sum_{i=1}^{m}(j-1)^i$, and since each of these "ones" (except the one at the root) may be placed in any of $k-1$ places, the number of different deviations is $(k-1)^{j\sum_{i=1}^{m}(j-1)^i}$. For cycle codes, the deviations are particularly simple: since each site is contained in only two check sets, each “one” in the deviation only gives rise to one additional “one”, and the support takes the form of a straight walk starting in a leaf site, going through the root site, and ending in another leaf site. ✻

It is a noteworthy property of low-density parity-check codes that the weight of their deviations increases without bounds when the computation tree grows. This makes it interesting to consider the asymptotic decoding performance of such codes when the number of iterations increases. Not all systems have the property that the weight of their deviations increases when the computation tree grows. In particular, trellises contain detours of fixed weight (see Example 4.1), and thus the asymptotic performance of trellis codes (with a fixed trellis structure) is not as interesting.

We conclude this section with a discussion of the computation tree of the turbo codes.

4.2.1 Computation Tree of Turbo Codes

The Tanner graph for the turbo codes of Berrou et al. [7] is illustrated in Figure 2.5 of Chapter 2 (page 11). We consider the computation tree for turbo codes, rooted at an information site (i.e., a connected visible site). If the min-sum (or sum-product) algorithm was applied in a parallel manner, alternatingly updating all check-to-site cost functions and all site-to-check cost functions, we would get a computation tree of the form illustrated in Figure 4.4, with all leaves occurring at the same depth. The decoder proposed in [7], however, does not use such an updating scheme, but instead processes the two trellises alternatingly, in a forward-backward fashion, and updates in between the cost functions in the connections between the two trellises. This results in a computation tree of the form illustrated in Figure 4.5, where the number of “trellis levels” is determined by the number of decoding iterations, but each trellis occurs in its entirety.

We now consider the corresponding deviation set. Assigning a “one” to the root site of the tree system means that an information bit is “one” in both of the trellises, which results in a detour in each of the trellises. Since the convolutional encoders are recursive (cf. [7] and Section 2.2), a detour with a single “one” among the information bits extends indefinitely (or to the end of the trellis) in either one or both directions of the trellis, and thus have infinite weight. An additional “one” among the information bits (connected sites) of the detour may result in the next trellis state being zero, thus ending that detour; but the detour may also continue, depending on where the additional “one” is placed. In either case, this “one” results in a new detour in the trellis at the next level of the computation tree. Thus, the support of a deviation always extends out to the leaves of the computation tree, making it interesting to consider asymptotic decoding performance. (Note that this is a consequence of the recursive convolutional encoders; feed-forward encoders would result in deviations of finite weight.)

Figure 4.6 illustrates the support of a few possible deviations. Some of the deviations have support that resembles those of a cycle code, i.e., walks on the computation tree; these deviations generally have the lowest weight among all deviations.

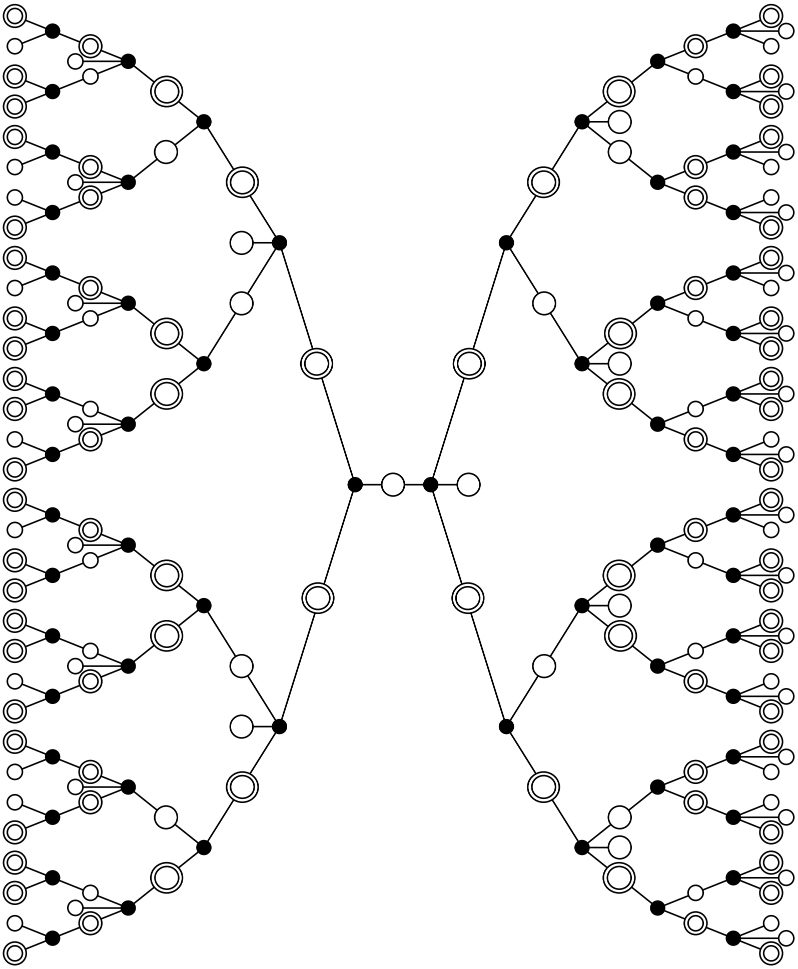


Figure 4.4 *Computation tree for the turbo codes of Berrou et al. after a few* ***parallel*** *updates of all cost functions.*

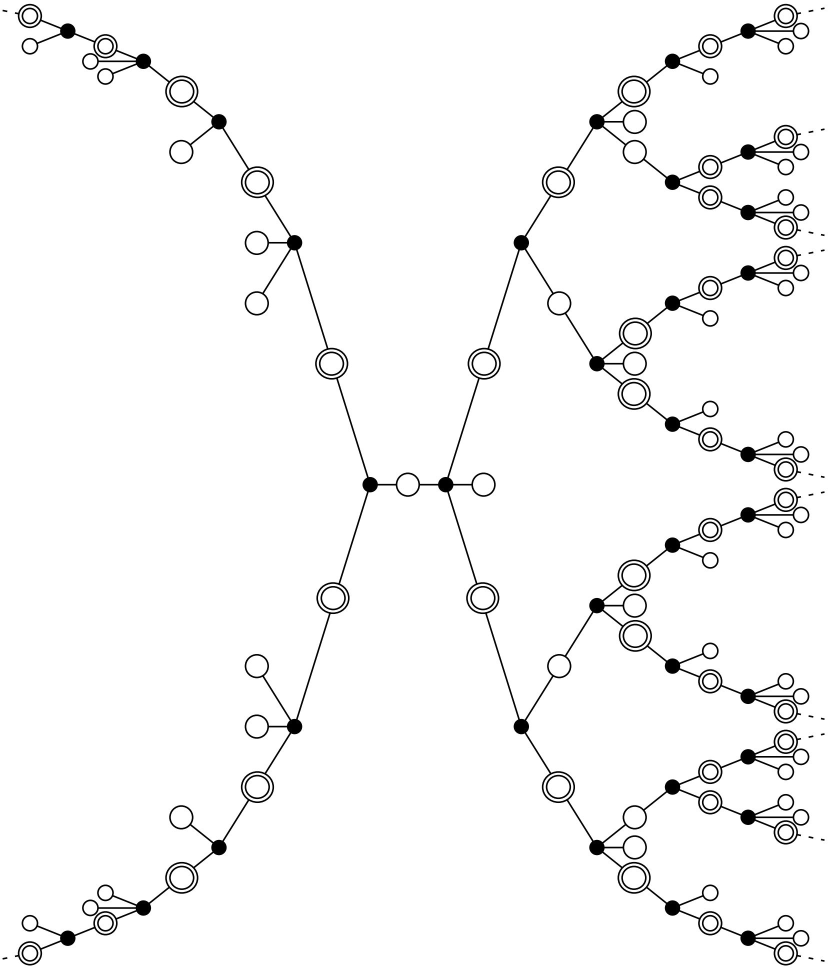


Figure 4.5 *Computation tree for the turbo codes after completely updating the two trellises one time each. The dashed lines indicate that the computation tree extends outside of the figure (to the end of the trellises).*

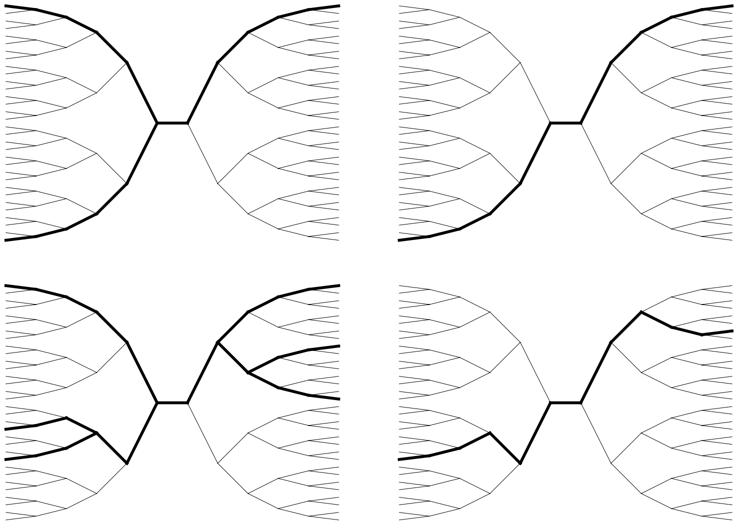


Figure 4.6 *Schematic illustration of (the support of) a few deviations in the turbo-code computation tree. Top left: only one information bit (the root site) is "one"; one of the two resulting detours extends in both directions, and the other extends in only one direction. Top right: only one information bit is "one", but the resulting detours extend in one direction only. Bottom left: additional "ones" in the information bits give rise to more detours; in some cases the detours are "cancelled" by such "ones", i.e. the zero state is reached. Bottom right: all "ones" in the information bits are placed such that their detour reaches the zero state.*

Chapter 5

Decoding Performance on Cycle-Free Subgraphs

Since cycle-free Tanner graphs provide optimal decoding, it is natural to try to minimize the influence of the cycles even in those cases where there are cycles in the graphs. This is the approach taken both for low-density parity-check codes in [5] and for turbo codes in [7]. By making the cycles long enough, the algorithms may run several iterations without being affected by them, in practice working only on cycle-free subgraphs of the Tanner graph.

Based on this observation, Gallager [5] obtained a performance estimate for sum-product decoding of low-density parity-check codes. We will discuss other estimates that apply before the cycles are closed. In particular, the standard tools for analyzing Viterbi decoding (union bound over the trellis detours) can be applied to the min-sum algorithm for any realization. This is performed both for the general case and for the special cases of low-density parity-check codes and turbo codes.

Two kinds of performance estimates are interesting. Given a specific realization with known girth $2k + 1$, it may be interesting to estimate the probability of decoding error after k iterations of the min-sum (or sum-product) algorithm. We will be more interested in the asymptotic error probability when the number of decoding iterations tends to infinity, requiring realization families with asymptotically unbounded girth (and, consequently, asymptotically unbounded block length). More specifically, we will investigate under what circumstances (i.e., for what channels) the error probability tends to zero when the girth tends to infinity.

5.1 Estimating the Error Probability with the Union Bound

From Corollary 4.4 we have

$$\Pr\{\text{decoding error}\} \le \Pr\{\mathcal{G}(e) \le 0 \text{ for some } e \in \mathcal{E}\} \le \sum_{e \in \mathcal{E}} \Pr\{\mathcal{G}(e) \le 0\} \ . \tag{5.1}$$

Here we can apply the standard techniques found in textbooks on communications, for example [21]. Since the channel is stationary and memoryless, we can write $\Pr\{\mathcal{G}(e) \le 0\} = p_0(w)$, where w is the weight of e and $p_0(w)$ is the probability that a particular codeword of weight w is more likely than the zero codeword given the received sequence. If A_w is the number of configurations in $\mathcal{E}$ with weight w, we may thus write $\sum_{e \in \mathcal{E}} \Pr\{\mathcal{G}(e) \le 0\} = \sum_{w=0}^{\infty} A_w p_0(w)$. For some channels, the bound may be even further simplified (weakened) using a bound of the form $p_0(w) \le \beta^w$. In particular, for the binary symmetric channel with crossover probability ε, we have $\beta = 2\sqrt{\varepsilon(1-\varepsilon)}$, while for the additive white gaussian noise channel with signal-to-noise ratio P/σ^2, we have $\beta = e^{-P/2\sigma^2}$ (see [21]). For such channels, we obtain the elegant formula

$$\Pr\{\text{decoding error}\} \le T(w)\big|_{w=\beta}, \tag{5.2}$$

where $T(w) \triangleq \sum_{i=0}^{\infty} A_i w^i$ is the *weight enumerator* for the deviation set.

Example 5.1 (low-density parity-check codes) The weight distribution of the deviation set for a (j, k) low-density parity-check code was determined in Example 4.2. If the decoding is terminated after m steps without closing any cycle, then the bit error probability is upper bounded by

$$\Pr\{\text{decoding error}\} \le (k-1)^{j\sum_{r=1}^{m}(j-1)^{r-1}} \beta^{1 + j\sum_{r=1}^{m}(j-1)^{r-1}} \tag{5.3}$$

$$= \beta e^{(\ln(k-1) + \ln\beta) j \sum_{r=1}^{m}(j-1)^{r-1}} . \tag{5.4}$$

Figure 5.1 illustrates this bound for a few values of (j, k) in the case of a Gaussian channel. Both from (5.4) and the figure it is clear that the bound tends to zero when m increases if and only if $\beta < 1/(k-1)$, and the speed at which this happens depends heavily on j (faster for larger j). Considering that the rate of a low-density parity-check code is $R \ge 1 - j/k$ (see [5]), it is rather surprising that the range of asymptotically "tolerable" channels is independent of j.

A different bound was obtained by Gallager [5] for the case of a binary symmetric channel and $j \ge 3$ (i.e., not cycle codes) and the sum-product algorithm. A comparison between the union bound (for the min-sum algorithm) and Gallager's bound (for the sum-product algorithm) is illustrated in Figure 5.2. The union bound gives only trivial bounds for channels with a high noise level but has a fast convergence for better channels; Gallager's bound has a much smoother behavior. Even though the bounds apply to two different decoding algorithms, it appears that the union bound is weak for channels with high noise levels. ✻

A general approach to computing the weight enumerator and the union bound of the error probability is provided by Lemma 4.6 and the sum-product algorithm. With the aid of a computer program for symbolic expression manipulation, it is possible to use the sum-product algorithm applied to the deviation set $\mathcal{E}$ to compute the weight enumerator symbolically,

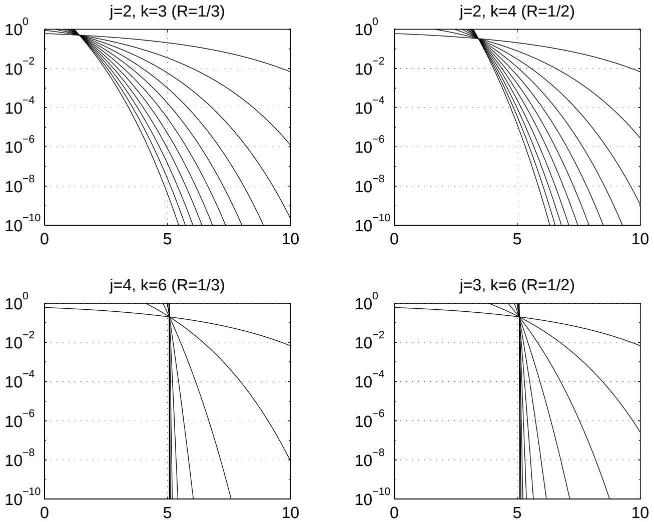


Figure 5.1 *Union bound estimate for a few low-density parity-check codes and the min-sum algorithm. The diagrams, which are obtained from Equation 5.4, show the union upper bound of the bit error probability versus the signal-to-noise ratio (in dB) of a Gaussian channel, for 0 to 10 cycle-free decoding iterations. (In the bottom two diagrams, the curves cannot be distinguished for more than the first few iterations.)*

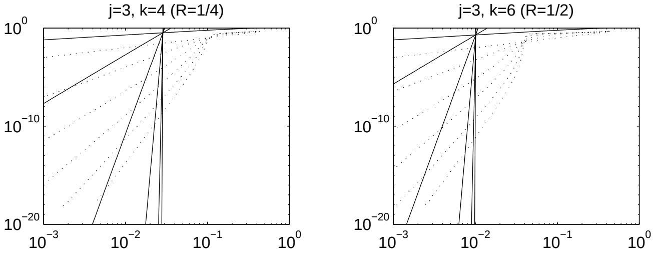


Figure 5.2 *Comparison between union bound estimate (solid) and Gallager's estimate (dotted), for 0, 2, 4, 6, 8, and 10 decoding iterations on a binary symmetric channel. The diagrams show the probability of decoding error versus the crossover probability of the channel.*

in general resulting in a huge polynomial $T(w)$, which can be used to compute upper bounds for the error probability for different channels. If the goal is to compute $T(w)$ numerically for a particular value $w = \beta$ (i.e., for a particular channel), then it may be more efficient to substitute β for w directly in each site, and perform the sum-product algorithm numerically. We are, however, more interested in the asymptotic behavior of $T(w)$ when the number of iterations increases.

5.2 Asymptotic Union Bound

For many graph-based code constructions, it is possible to construct infinite code families with asymptotically infinite girth in the Tanner graph. For such families, it is interesting to consider the asymptotic error probability when the number of decoding iterations increases; in particular, this error probability may tend to zero if the channel is not too noisy. By applying the techniques discussed above, we can compute the weight enumerator $T^{(i)}(w)$ corresponding to i iterations, and evaluate for what values (if any) of the channel parameter β the union bound $T^{(i)}(\beta)$ tends to zero as the number of iterations tend to infinity (as we did in Example 5.1).

For complicated realizations it may be difficult to obtain a closed-form expression for $T^{(i)}(w)$. This is not necessary, however. Instead, we can determine directly from the structure of the realization for what values of β we have $T^{(i)}(\beta) \to 0$ as $i \to \infty$. The idea is to analyze how $T(w)$ changes when one more decoding iteration is performed, i.e., when the computation tree grows.

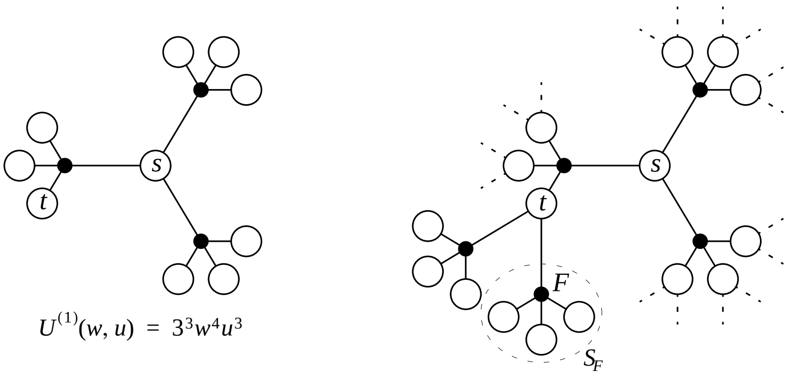
For (j, k) low-density parity-check codes with parallel updates, increasing the number of decoding iterations by one corresponds to increasing the depth of the computation tree by one, i.e., to append $j-1$ parity checks to each leaf site and $k-1$ sites to each new such parity check. For turbo codes, we will consider one “decoding iteration” to be a complete updating of an entire trellis, so that increasing the number of decoding iterations by one corresponds to appending a whole trellis to the “information” sites that are leaves in the current computation tree. (There are some problems with this view, which we will return to in Example 5.3.)

Consider an infinite sequence $\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, ..$ of larger and larger computation trees, where $\mathcal{T}_i$ is the computation tree obtained after i iterations with the min-sum algorithm ($\mathcal{T}_0$ is the computation tree after zero iterations, containing only the root site). For any such computation tree $\mathcal{T}_i$, we define its *extendable leaf sites* as those leaf sites that occur as interior sites of some larger computation tree. We require, for simplicity, that all extendable leaf sites are binary, and that they are extended in exactly the same way (i.e., the subtrees emanating from all extendable leaf sites have the same set of valid tree configurations). This is true for low-density parity-check codes; it is also true for unpunctured turbo codes (but not for punctured turbo codes, whose extendable leaf sites differ depending on whether they extend through a punctured trellis section or not).

Let $\mathcal{E}_i$ be the deviation set of $\mathcal{T}_i$, and define the *extendable weight enumerator* $U^{(i)}(w, u)$ for $\mathcal{E}_i$ as

$$U^{(i)}(w, u) \triangleq \sum_{p, q} U^{(i)}_{p, q} w^p u^q, \tag{5.5}$$

where $U^{(i)}_{p, q}$ is the number of deviations in $\mathcal{E}_i$ with weight q among the extendable leaf sites, and weight p among all visible sites. In other words, the variable u corresponds to “ones” in the extendable leaf sites. Note that a site can be both visible and an extendable leaf site at the



$$U^{(1)}(w, u) = 3^3 w^4 u^3$$

$$V(w, u) = 3wu$$

$$\begin{aligned} U^{(2)}(w, u) &= U^{(1)}(w, V^2(w, u)) \\ &= 3^9 w^{10} u^6 \end{aligned}$$

Figure 5.3 *The process of "extending" an extendable leaf site. Left: the computation tree corresponding to one decoding iteration on a* (3, 4) *low-density parity-check code: s is the root site and t is an extendable leaf site; the extendable weight enumerator is shown below the tree. Right: the computation tree after one further decoding iteration (only a part is shown). S are the sites of the extension tree of the site t through the check set F. As shown below the tree, the new extendable weight enumerator is obtained from the old by substituting, for u, the squared extendable weight enumerator for the extension tree (the reason for the squaring is that each extendable leaf site is extended with* ***two*** *extension trees).*

same time; a "one" in such a site corresponds to a factor wu in a term of $U^{(i)}(w, u)$. The ordinary weight enumerator $T^{(i)}(w)$ for the deviation set $\mathcal{E}_i$ is obtained by simply discarding u, i.e., as

$$T^{(i)}(w) = U^{(i)}(w, 1) . \tag{5.6}$$

Next, let t be an extendable leaf site in $\mathcal{T}_i$, and let F be a check set in $\mathcal{T}_{i+1}$ that is not also in $\mathcal{T}_i$ but that is adjacent to t (in other words, t is the innermost site of F). The subtree of $\mathcal{T}_{i+1}$ emanating from F will be called the *extension tree* of t through F. Let d be the number of such extension trees of the extendable leaf site t. (For example, $d = j - 1$ for (j, k) low-density parity-check codes, except for the root site, for which $d = j$). Let S_F be the sites of the extension tree from t through F. Let $V(w, u)$ be the extendable weight enumerator (for the deviation set) on the extension tree, defined as

$$V(w, u) \triangleq \sum_{p, q} V_{p, q} w^p u^q, \tag{5.7}$$

where $V_{p,q}$ is the number of local deviations $e_S \in (\mathcal{E}_{i+1})_{S_F}$ with weight q among all extendable leaf sites in S_F and weight p among all visible sites in S_F (note that $t \notin S_F$). The point is that $U^{(i+1)}(w, u)$ can be obtained from $U^{(i)}(w, u)$ by simply substituting, for u, $V^d(w, u)$ ($V(w, u)$ raised to d). In other words, we have

$$U^{(i+1)}(w, u) = U^{(i)}(w, V^d(w, u)) \ . \tag{5.8}$$

(This follows from the properties of deviations: the presence of a factor u in a term of $U^{(i)}(w, u)$ indicates that an extendable leaf site is “one” in a corresponding deviation e. But such a “one” forces a nonzero configuration (a deviation) in the subtrees of that leaf site.)

Using (5.8), the extendable weight enumerator $U^{(i)}(w, u)$ can be computed for any i, and therefore also the ordinary weight enumerator $T^{(i)}(w)$. To determine $\lim_{i \to \infty} T^{(i)}(\beta)$, we note that $U^{(i)}(w, u)$ is a polynomial in w and u with nonnegative coefficients. This implies that $U^{(i)}(w, u)$ is a monotone function in u, so that

$$U^{(i)}(\beta, u) < U^{(i)}(\beta, 1) \text{ if and only if } u < 1 \ . \tag{5.9}$$

Using (5.6) and (5.8), we can write the weight enumerator function $T^{(i+1)}(w)$ as

$$T^{(i+1)}(w) = U^{(i+1)}(w, 1) = U^{(i)}(w, V^d(w, 1)) \ , \tag{5.10}$$

and we obtain, from (5.9),

$$T^{(i+1)}(\beta) < T^{(i)}(\beta) \text{ if and only if } V^d(\beta, 1) < 1 \ . \tag{5.11}$$

From (5.11) it follows that a necessary condition for $T^{(i)}(\beta)$ to approach zero when $i \to \infty$, is that $V(\beta, 1) < 1$. This is not a sufficient condition, however, since $T^{(i)}(\beta)$ may be bounded from below by some positive constant. (In fact, this is exactly what happens when there are deviations of fixed, finite weight regardless of the number of iterations, which is the case for detours in a trellis.) A precise condition for convergence to zero is given by the following theorem.

Theorem 5.1 The upper bound $T^{(i)}(\beta)$ for the error probability tends to zero as the number i of iterations tends to infinity if $V(\beta, 1) < 1$ and $V(w, u)$ contains a factor u.

Proof. Assume that the required conditions are met. Since $U^{(0)}(w, u) = wu$ it follows by induction that for all i, $U^{(i)}(w, u)$ contains a factor u, so we can write $U^{(i)}(w, u) = u\tilde{U}^{(i)}(w, u)$ for some polynomial $\tilde{U}^{(i)}(w, u)$ with nonnegative coefficients. From (5.6) and (5.8) we obtain

$$T^{(i+1)}(\beta) = U^{(i+1)}(\beta, 1) = U^{(i)}(\beta, V^d(\beta, 1)) = V^d(\beta, 1)\tilde{U}^{(i)}(\beta, V^d(\beta, 1)) \ . \tag{5.12}$$

Since $\tilde{U}^{(i)}(w, u)$ is a polynomial with nonnegative coefficients and since $V(\beta, 1) < 1$, we get from (5.11)

$$\tilde{U}^{(i)}(\beta, V^d(\beta, 1)) \le \tilde{U}^{(i)}(\beta, 1) , \tag{5.13}$$

which can be inserted into (5.12) to obtain the upper bound

$$T^{(i+1)}(\beta) \le V^d(\beta, 1)\tilde{U}^{(i)}(\beta, 1) \tag{5.14}$$

$$= V^d(\beta, 1)U^{(i)}(\beta, V^d(\beta, 1)) \tag{5.15}$$

$$= V^d(\beta, 1)T^{(i)}(\beta) , \tag{5.16}$$

and thus we have obtained

$$T^{(i+1)}(\beta) \le \alpha T^{(i)}(\beta) \tag{5.17}$$

for $\alpha = V^d(\beta, 1) < 1$. Thus, $\lim_{i \to \infty} T^{(i)}(\beta) = 0$. ❑

Note that the range of β for which $T^{(i)}(\beta) \to 0$ is independent of *d*, but a larger *d* gives a faster convergence.

Example 5.2 (low-density parity-check codes) For a (j, k) low-density parity-check code, we have (from Example 4.2) $V(w, u) = (k-1)wu$ and $d = j-1$ (except for $i = 0$, when $d = j$). Since V contains a factor *u*, it follows from Theorem 5.1 that the error probability tends to zero if $V(\beta, 1) < 1$, i.e., if $(k-1)\beta < 1$. This is the same result as obtained in Example 5.1. ✲

A specific requirement for this approach was that all extendable leaf sites should extend in the same way. This rules out the classic turbo codes which extend differently through punctured and unpunctured sections. It is possible to handle a more general situation by using an extendable weight enumerator with several variables $u_1, u_2, \ldots$, one for each type of extendable leaf site. It is also possible to handle nonbinary extendable leaf sites, such as trellis state spaces, by assigning a unique variable for each nonzero value of the leaf sites. In the following example, we avoid this complexity by considering an unpunctured turbo code.

Example 5.3 (union bound for turbo decoding) We consider a simple unpunctured turbo code based on the four-state component trellis illustrated in Figure 5.4 (since it is unpunctured, it has the rate $R = 1/3$), with the usual forward-backward updating scheme, alternating between the two trellises, which we assume to have infinite length. This corresponds

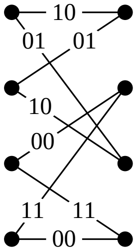


Figure 5.4 One section of the component trellis for a simple turbo code. The leftmost bit is shared between the two trellises.

to a computation tree similar to the one in Figure 4.5, but with redundancy sites in all sections. At the same time, we will assume that there are no multiple sites in the computation tree, i.e. no cycles have closed during the computation.

These assumptions make the analysis convenient, since there is only one type of extendable leaf sites in each computation tree, namely the visible connected sites (the information bits of the trellis), and since these leaf sites are binary. Extending such a site amounts to appending a complete trellis to the computation tree, as discussed in Section 4.2.1. The drawback of these assumptions is that they exclude a finite realization (or block length) even for a single decoding iteration, since each trellis has infinite length in itself. Thus, the result to be presented in this example does not strictly correspond to any family of finite realizations with asymptotically infinite girth. (The small state space of the trellis code suggests, however, that the "effective length" of the trellis is relatively short, so that the influence of cycles in a single trellis should be small. Strict results could be obtained by having as extendable leaf sites both visible connected sites and trellis state sites, thus extending the computation tree in two ways simultaneously.)

Adopting these assumptions, we get the extendable weight enumerator for the extension tree (5.7) as a version of the usual path weight enumerator for trellis codes; the difference is that we have to treat the information bits (the extendable leaf sites) specially. More precisely, $V(w, u) \triangleq \sum_{p,q} V_{p,q} w^p u^q$ where $V_{p,q}$ is the number of detours with overall weight p and weight q in the connected "information" positions (not counting the root site of the detours). This function can be obtained using the conventional flowchart technique (cf, e.g. [21]), obtaining (for the trellis section of Figure 4.5)

$$V(w, u) = \frac{2w^5u - 2w^7u + 3w^4u^2 - 5w^6u^2 - 8w^5u^3 + 4w^7u^3 + 7w^6u^4 - 2w^7u^5}{1 - 2w^2 + w^4 - 4wu + 4w^3u + 6w^2u^2 - 2w^4u^2 - 4w^3u^3 + w^4u^4}. \quad (5.18)$$

Since $V(w, u)$ has a factor u, we can use Theorem 5.1 to determine a range on β for which the error probability tends to zero with an increasing number of cycle-free iterations. On a Gaussian channel, this results in the range $\mathrm{SNR} > 2.4$ dB, which should be compared to the Shannon limit at -2.3dB (for the rate $1/3$), i.e., the bound is 4.7dB above the channel capacity curve. Apparently this is a very weak result, since simulations on the same code indicate that the error probability decreases already at about -1 dB, cf. Figure 5.5. ✲

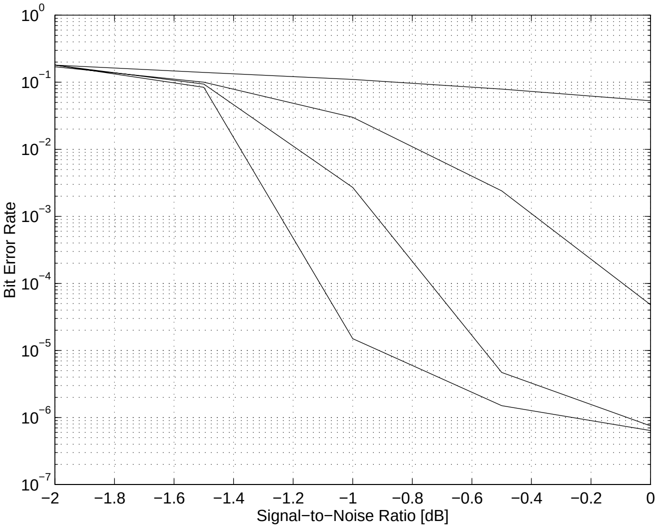


Figure 5.5 *Simulation results showing bit error rate on a Gaussian channel for the rate 1⁄3, 4-state turbo code discussed in the text.*

From these two examples (low-density and turbo codes), it appears that estimates based on the union bound are rather weak. This is not surprising, since many deviations are highly overlapping (for example, there are deviations which differ only in a few leaf sites), a fact which is discarded in the union bound argument.

We now consider a different approach which is not based on the union bound.

5.3 Computing Statistical Moments of Final Costs

An alternative approach for estimating the performance of iterative decoding on cycle-free Tanner graphs is to consider the statistical properties of the cost functions. We will describe two such methods. The first has the “potential” of being applicable to any code description and of giving strict upper bounds to the error probability; unfortunately, it seems that the bounds obtained are only trivial, i.e., larger than 1. The second method, which is based on insights obtained from the first, is unfortunately only approximate. However, it seems to give fairly accurate predictions of the error probability for some systems.

The first method applies only to the sum-product algorithm (rather than the min-sum algorithm, to which all other discussion in this chapter applies). The method is based on the observation that it is easy to compute statistical moments of the final costs computed by the sum-product algorithm, provided that the corresponding moments of the local costs are known. If we know the first and second moments of the final costs $\mu_s(0)$ and $\mu_s(1)$, then it is also easy to compute the mean and variance of their difference $D = \mu_s(0) - \mu_s(1)$, and we can apply well-known bounds to compute the error probability $\Pr\{D < 0\}$.

Recall from Section 3.2 that the final costs computed by the sum-product algorithm for cycle-free Tanner graphs are of the form $\mu_s(a) = \sum_{x \in B : x_s = a} G(x)$, where the global cost is defined as $G(x) = \prod_{s \in N} \gamma_s(x_s)$ (we assume, for simplicity, that there are no local check cost functions). On a memoryless channel, the local costs $\gamma_s(x_s)$ are statistically independent for different sites s, and thus the statistical moments of $G(x)$ can be computed as

$$E[(G(x))^k] = \prod_{s \in N} E[(\gamma_s(x_s))^k] \quad . \tag{5.19}$$

The first moment of the final costs, i.e., their mean, is particularly simple to compute. Since the final cost $\mu_s(a)$ is just the sum of the global costs for all x with $x_s = a$, we have

$$E[\mu_s(a)] = \sum_{x \in B : x_s = a} E[G(x)] = \sum_{x \in B : x_s = a} \prod_{s \in N} E[\gamma_s(x_s)] \quad , \tag{5.20}$$

which means that the mean of the final costs from the sum-product algorithm can be computed recursively with the sum-product algorithm, using as local costs the mean of the actual local costs. In fact, even the second (and other) moments of the final costs can be computed with the sum-product algorithm; however, the computation in those cases has to be carried out on a larger system (N, W', B'), where the alphabets are products of the original alphabets, i.e., $W'_s = W_s \times W_s$, and the local behaviors are products of the original local behaviors (in fact, this is true only for the second moments; for higher moments even larger systems are required). We will not go into the details of this computation since the results are not encouraging anyway. Instead, we will look at the results for a particular example.

Consider a cycle code with three bits in each parity check. Let the all-zero codeword be transmitted on a Gaussian channel with noise variance σ^2 using antipodal signaling at the signal levels ± 1. The sum-product algorithm is applied with the local costs defined as $\gamma_s(0) = \exp(-z_s^2/2\sigma^2)$ and $\gamma_s(1) = \exp(-(z_s - 2)^2/2\sigma^2)$, where z_s is the noise component at site s. The mean of the local costs may then be computed as $E[\gamma_s(0)] = 1/\sqrt{2}$, $E[\gamma_s(1)] = \exp(-1/\sigma^2)/\sqrt{2}$, and the second moments as $E[(\gamma_s(0))^2] = 1/\sqrt{3}$, $E[\gamma_s(0)\gamma_s(1)] = \exp(-4/3\sigma^2)/\sqrt{3}$, and $E[(\gamma_s(1))^2] = \exp(-4/3\sigma^2)/\sqrt{3}$. Using the method outline above, we have computed the mean and standard deviation of the "discriminator" $D \triangleq \mu_s(0) - \mu_s(1)$ for $\sigma^2 = 1/2$, i.e., for signal-to-noise ratio 3dB, and the result is displayed in Table 5.6 for 1 to 5 decoding iterations. Clearly, the ratio between the standard deviation and the mean grows quickly when the number of decoding iterations increases, and consequently, it is impossible to obtain a nontrivial estimate of the error probability $\Pr\{D \le 0\}$ without making assumptions on the probability distribution of D.

Table 5.6 also displays the actual (simulated) error probability for the sum-product algorithm in this case, and it is obvious that the approach based on first and second moments of the final costs is useless. The reason for this can be found in Figure 5.7, which illustrates the empirical distribution of the discriminator D after one iteration with the sum-product algorithm, obtained from simulation of 10000 decoding attempts. The distribution is heavily asymmetric; the mean is mainly determined by a large peak just to the right of the decision

Table 5.6 *Mean and variance of sum-product "discriminator"* $\mu_s(0) - \mu_s(1)$.

Number of Decoding Iterations	Mean	Standard Deviation	Simulated Error Probability
1	0.18	0.18	0.021
2	0.012	0.026	0.0069
3	5.0×10^{-5}	3.6×10^{-4}	0.0024
4	8.9×10^{-10}	6.0×10^{-8}	0.0010
5	2.8×10^{-19}	1.6×10^{-15}	0.00046

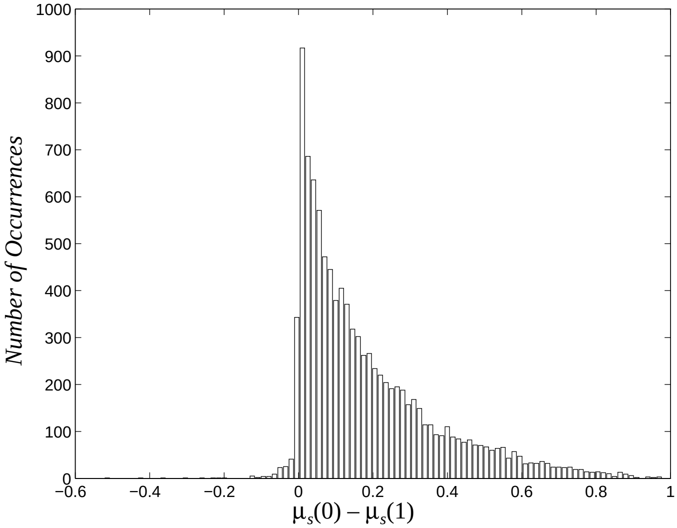


Figure 5.7 *Histogram of "discriminator"* $\mu_s(0) - \mu_s(1)$ *after one iteration with the sum-product algorithm on a cycle code.*

boundary at $D = 0$, while the relatively few samples with large positive value are the reason for the large variance. The few negative samples, which correspond to the error events, influence the moments very little. (After further decoding iterations, this situation is even more exaggerated.)

To better understand the situation, we have plotted the empirical distribution of the final costs $\mu_S(0)$ and $\mu_S(1)$ in Figure 5.8. After a few decoding iterations, the final costs appear approximately Gaussian on a logarithmic scale, i.e., the log-costs $\log\mu_S(0)$ and $\log\mu_S(1)$

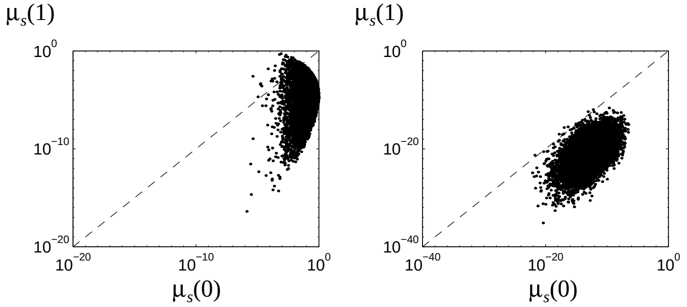


Figure 5.8 *Distribution of final costs after one iteration (left) and four iterations (right) with the sum-product algorithm on a cycle code. Each dot is one decoding attempt. The dashed line is the decision boundary, i.e., where* $\mu_s(0) = \mu_s(1)$.

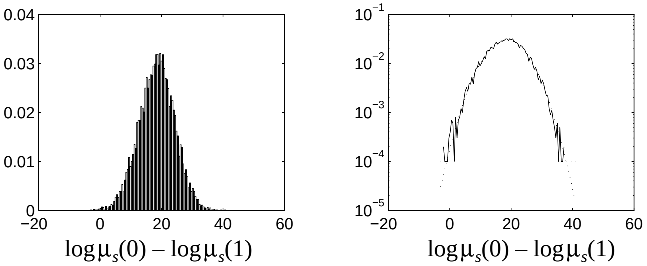


Figure 5.9 *Distribution of log-cost-ratio* $\log\mu_s(0) - \log\mu_s(1)$ *after four iterations with the sum-product algorithm on a cycle code. The left diagram shows a histogram on a linear scale, while the right diagram compares the histogram with a Gaussian distribution on a logarithmic scale.*

are almost Gaussian. In particular, the empirical distribution of the “log-cost-ratio” $\log\mu_s(0) - \log\mu_s(1)$ approximates a Gaussian distribution relatively well, as illustrated in Figure 5.9.

It seems natural, then, to *assume* that the log-cost-ratio is truly Gaussian, and to use this assumption to obtain an approximate estimate of the error probability. It turns out that, with the additional assumption that the difference $\log\mu_s(0) - \log\mu_s(1)$ is statistically independent from the sum $\log\mu_s(0) + \log\mu_s(1)$—an assumption that is also justified by simulation results such as Figure 5.8—it is possible to express the final Gaussian distribution of $\log\mu_s(0) - \log\mu_s(1)$ directly in terms of the first and second moments of $\mu_s(0)$ and $\mu_s(1)$. Unfortunately, the estimates obtained in this manner do not conform at all with simulation results. While we have not investigated this approach much further, it appears that the highly asymmetric “log-normal” distributions of the final costs $\mu_s(0)$ and $\mu_s(1)$ are inherently difficult to handle, and that it may be easier to deal directly with the distribution of the log-cost ratios of the intermediate and final cost functions.

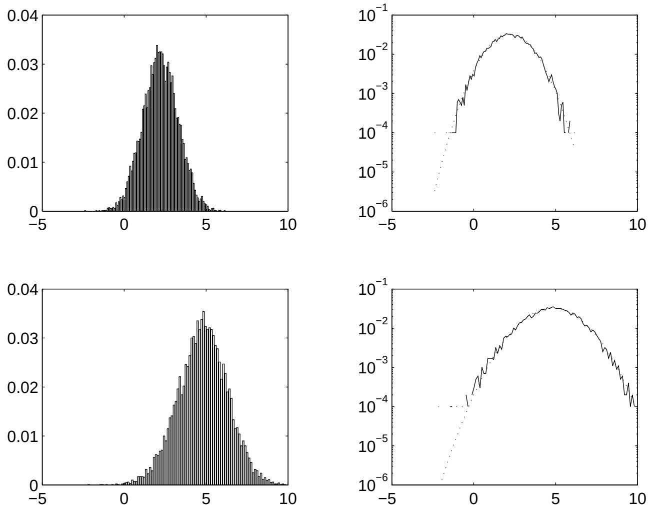


Figure 5.10 *Distribution of final cost difference $\mu_s(1) - \mu_s(0)$ from min-sum algorithm as applied to a cycle code used on a Gaussian channel. The upper diagrams were obtained after one decoding iteration and the lower after four decoding iterations. The dotted curves are true Gaussian distributions with the same mean and variance as the empirical distributions.*

5.4 Gaussian Approximation of Log-Cost-Ratios

As mentioned in the previous section, the empirical distribution of the final "log-cost-ratio" $\log\mu_s(0) - \log\mu_s(1)$ for the sum-product algorithm is very close to a Gaussian distribution. It turns out that this is almost always the case, both for final and intermediate cost functions. Furthermore, the same holds for the cost functions in the min-sum algorithm, i.e., the empirical distribution of cost differences such as $\mu_s(0) - \mu_s(1)$ and $\mu_{E,s}(0) - \mu_{E,s}(1)$ are very close to being Gaussian, especially if the local costs are Gaussian distributed, i.e., if the channel is Gaussian. See Figure 5.10.

In this section, we will discuss performance estimates for the min-sum algorithm, based on the assumptions that the cost differences (of both final and intermediate cost functions) are truly Gaussian. Since this is only an approximation (albeit often a rather good one), the results obtained here are only approximate.

We will only consider low-density parity-check codes. For these codes, the min-sum updating rules have the following simple form (cf. Section 3.4):

$$\mu_s = \sum\nolimits_{E \in Q : s \in E} \mu_{E,s} \tag{5.21}$$

$$\mu_{E,s} = \prod_{s' \in E : s' \neq s} \operatorname{sign}(\mu_{s',E}) \min_{s' \in E : s' \neq s} |\mu_{s',E}| \tag{5.22}$$

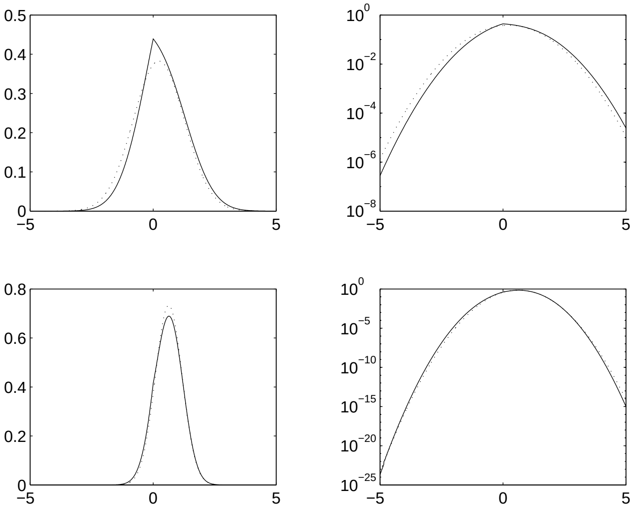


Figure 5.11 *Probability distribution of check-to-site cost* $\mu_{E,s} = \mu_{E,s}(1) - \mu_{E,s}(0)$ *as computed by the min-sum algorithm on a cycle code with three bits per parity check. The site-to-check costs* $\mu_{s',E} = \mu_{s',E}(1) - \mu_{s',E}(0)$ *are assumed to have Gaussian distribution with mean 1. In the upper two diagrams, the variance of* $\mu_{s',E}$ *is 2, and in the lower two diagrams, the variance is 1/2. The solid curves are the actual probability distributions; the dotted curves are Gaussian distributions with the same mean and variance.*

$$\mu_{s,E} = \sum_{E' \in Q : s \in E', E' \neq E} \mu_{E',s} . \tag{5.23}$$

Clearly, if the check-to-site costs $\mu_{E,s}$ were truly Gaussian, then so would the final costs μ_s and the site-to-check costs $\mu_{s,E}$, since they are computed as sums of the former (Equations 5.21 and 5.23). Moreover, if the check-to-site costs are “close” to Gaussian, then, according to the central limit theorem, the other costs will be even “closer” to Gaussian. So, the critical point is obviously the distribution of the check-to-site costs, as computed according to (5.22). Figure 5.11 illustrates the distribution of the check-to-site costs $\mu_{E,s} = \mu_{E,s}(1) - \mu_{E,s}(0)$ for a cycle code with three bits per parity check, indicating that the distribution is approximated relatively well by a Gaussian distribution in this case.

In fact, for cycle codes with three bits per parity check, it is possible to derive the mean and variance of the check-to-site costs $\mu_{E,s} = \mu_{E,s}(1) - \mu_{E,s}(0)$ analytically, assuming that the site-to-check costs are independent and Gaussian distributed. The derivation is rather lengthy (it can be found in Appendix A.4), and leads to the formulas

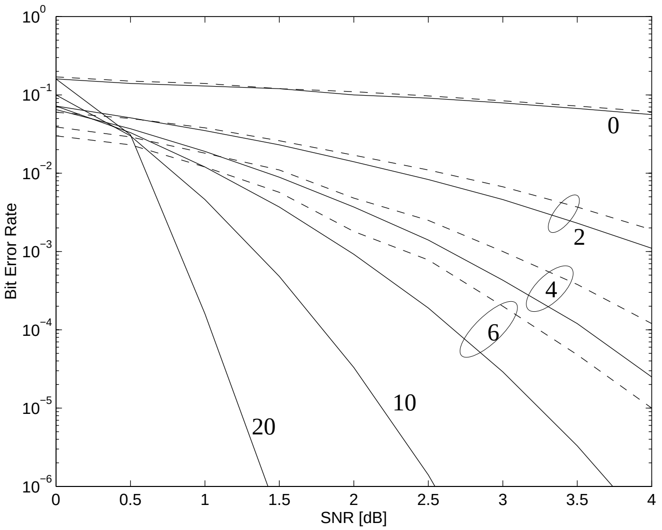


Figure 5.12 *Bit error rate with cycle code and the min-sum algorithm. The solid curves are the estimates obtained with the Gaussian assumption, while the dashed curves are obtained from simulations. The number of decoding iterations are shown in the diagram.*

$$E[\mu_{E,s}] = mQ(\sqrt{2}\frac{m}{\sigma}) - \frac{e^{-\frac{m^2}{\sigma^2}}\sigma}{2\sqrt{\pi}} \tag{5.24}$$

$$E[(\mu_{E,s})^2] = m^2 + \sigma^2 - \frac{2\sigma^2}{\pi}e^{-\frac{m^2}{\sigma^2}} - \frac{2m\sigma}{\sqrt{\pi}}\mathrm{erf}(\frac{m}{\sigma}) \ , \tag{5.25}$$

where m and σ are the mean and standard deviation of the site-to-check costs $\mu_{s',E}$, respectively, and $Q(\cdot)$ and $\mathrm{erf}(\cdot)$ are the usual “error functions”.

So, using this cycle code on a Gaussian channel, it is easy to compute estimates of the error probability, under the assumption that the check-to-site cost functions are almost Gaussian. In Figure 5.12, we have plotted the estimates obtained this way along with simulation results. As can be seen in the figure, the estimate is somewhat optimistic.

For more interesting realizations, such as turbo codes, it appears difficult to obtain a theoretical expression for the mean and variance of the log-cost ratios obtained during the computations. By simulating the decoding process for a single trellis, it would be possible to approximate the relation between the distributions of the incoming and outgoing log-cost ratios numerically.

Chapter 6

Decoding Performance with Cycles

We now turn to the case where the cycles do in fact influence the decoder, i.e., when there are multiple copies of some sites (and checks) in the computation tree. It is not at all obvious that the performance will continue to improve if the decoding process goes on after this point. However, simulation results strongly indicate that this may in fact be the case, at least for certain code constructions. This was already observed by Gallager in [5]; it is also clear that the dramatic performance of turbo codes [7] is achieved when the cycles have closed. Gallager explained this behavior by stating that "the dependencies have a relatively minor effect and tend to cancel each other out somewhat".

While the analysis of Chapter 5 does not apply in this case, the concept of computation trees and deviation sets, and in particular Theorem 4.2, still applies: a necessary condition for a decoding error to occur is that the global cost of some deviation $e \in \mathcal{E}$ is negative. However, the cost of a deviation is not a sum of independent local costs, as it is in the cycle-free case. Instead, some of the local costs are the same (since they correspond to the same sites). Therefore, the probability that the cost is negative depends not only on the weight of *e*, but also on the number of multiple occurrences. Still, it is possible to formalize the situation in a nice way, as follows.

Thus, let (N, W, B) be a system with $n = |N|$ sites, where the visible sites $V \subseteq N$ are all binary, and let $\mathcal{N}, \mathcal{W}, \mathcal{B})$ be a corresponding tree system rooted at the site *s*. Then we have

Definition 6.1 The *multiplicity vector* of a tree configuration $u \in \mathcal{W}$ is an integer-valued *n*-tuple $[u] \in \mathcal{Z}^n$, where $[u]_s$ is the number of tree sites *t* with $u_t = 1$ that correspond to *s* (for a visible site *s*).

With this definition, the global cost of the tree configuration *u* may be expressed as

$$\mathcal{G}(u) = \sum_{s \in V} [u]_s \gamma_s . \tag{6.1}$$

We now assume that the channel is Gaussian and (as usual) that the all-zero codeword was transmitted (using antipodal signaling); this means that the local costs γ_s are mutually independent and normally distributed with mean 1 and variance σ^2. Furthermore, from (6.1) we see that $\mathcal{G}(u)$ is normally distributed with mean $E[\mathcal{G}(u)] = \sum_{s \in V}[u]_s$ and variance $V[\mathcal{G}(u)] = \sigma^2 \sum_{s \in V}[u]_s^2$, giving

$$\Pr\{\mathcal{G}(u) \le 0\} = Q\left(\frac{\sum_{s \in V}[u]_s}{\sigma\sqrt{\sum_{s \in V}[u]_s^2}}\right), \tag{6.2}$$

where $Q(\cdot)$ is the usual error function.

The expression in (6.2) closely resembles the corresponding expression $Q(\sqrt{\text{weight}(u)}/\sigma)$ in the cycle-free case, making it natural to define the *generalized weight* $\omega(u)$ of a tree configuration u as the quantity

$$\omega(u) \triangleq \frac{\left(\sum_{s \in V}[u]_s\right)^2}{\sum_{s \in V}[u]_s^2}, \tag{6.3}$$

obtaining the convenient formulation

$$\Pr\{\mathcal{G}(u) \le 0\} = Q\left(\frac{\sqrt{\omega(u)}}{\sigma}\right). \tag{6.4}$$

Note that the generalized weight of u is in inverse proportion to a kind of “normalized empirical variance” of $[u]$, so that $\omega(u)$ is large when the components of $[u]$ are similar, whereas $\omega(u)$ is small when some components dominate over the rest, i.e., when a few sites occur more often than others in the support of u.

We now consider the deviation set and the probability of decoding error. Corollary 4.4 applies regardless of the presence of cycles, and we have the error probability bound

$$\Pr\{\text{decoding error}\} \le \Pr\{\mathcal{G}(e) \le 0 \text{ for some } e \in \mathcal{E}\}. \tag{6.5}$$

From (6.4), it is clear that $\omega(e)$ should be as large as possible to give a small error probability. This indicates that there should be no short cycles in the Tanner graph, because otherwise it is possible for a deviation to concentrate a lot of its support on such a short cycle and thus obtain a small generalized weight, thereby contributing to a large error probability. This idea will be formalized into a strict result, regarding cycle codes, below.

The upper bound on the error probability (6.5) may, in turn, be upper-bounded with the usual union bound, giving

$$\Pr\{\text{decoding error}\} \le \sum_{e \in \mathcal{E}} \Pr\{G(e) \le 0\} = \sum_{e \in \mathcal{E}} Q\left(\frac{\sqrt{\omega(e)}}{\sigma}\right) . \tag{6.6}$$

In parallel with the cycle-free case, it is even possible to define a "generalized weight enumerator" $T(\Omega) \triangleq \sum_{\omega} A_{\omega} \Omega^{\omega}$, where $A_{\omega} \triangleq |\{e \in \mathcal{E} : \omega(e) = \omega\}|$ is the number of deviations with generalized weight ω, and the sum in runs over all generalized weights ω that occur in $\mathcal{E}$. This would make it possible to write (a weakened version of) the bound (6.6) in the compact form $\Pr\{\text{decoding error}\} \le T(\beta)$, where, as usual, $\beta = e^{-1/2\sigma^2}$ for a Gaussian channel. Unfortunately, no method is known for computing the generalized weight distribution A_{ω}, apart from going through all deviations explicitly. In particular, it is not possible to apply the sum-product algorithm in the straightforward fashion described in Section 5.1, since the generalized weight of a configuration e is not additive over disjoint site subsets (as is the conventional weight).

For a limited class of systems, however, it is possible to analyze in more detail the behavior of the deviations when the number of decoding iterations tends to infinity. This is possible with systems for which the deviations have a "chain-like" structure, i.e., their support forms a single walk through the computation tree, starting and ending in two leaf sites and going through the middle (root) site. There are two interesting kinds of systems in this class: cycle codes and tailbiting trellises.

The idea is, essentially, that the lowest-cost deviation will have a periodic structure: the walk formed by such a deviation will spend most of its time in the lowest-cost part of the graph. We begin with the simplest case, namely cycle codes.

6.1 Cycle Codes

Recall that a cycle code is a low-density parity-check code with each bit checked by two parity checks. We noted in Example 4.2 that the support of a deviation $e \in \mathcal{E}$ forms a "walk" on the computation tree. More precisely, such a *walk* is a sequence of sites $s_1, s_2, \ldots$ in the original system, corresponding to a connected sequence of alternating sites and checks on the Tanner graph, with the property that the same edge is never utilized twice in a row (these requirements follow from the updating rules of the min-sum algorithm).

We need a few definitions. A finite walk $s_1 \ldots s_n$ will be called *closed* if the site sequence formed by concatenating the walk with itself, i.e., $s_1, \ldots, s_n, s_1, \ldots, s_n$, is also a walk. A walk $s_1, s_2, \ldots$ is *reducible* if, for some i and some $j > i$, the segment $s_i, \ldots, s_j$ is a closed walk and the remaining part $s_1, \ldots, s_{i-1}, s_{j+1}, \ldots$ is still a walk; walks that are not reducible are called *irreducible*. Clearly, any walk may be "factored" into a collection of irreducible closed walks (not necessarily distinct) and a single remaining irreducible walk.

A cycle on a Tanner graph is an irreducible closed walk, but there may be irreducible closed walks that are not cycles. The difference is that, in a cycle, all sites are distinct, whereas in a general irreducible closed walk, the same site may appear several times. See Figure 6.1 for an example of a closed irreducible walk that is not a cycle.

We are now ready for the main result about cycle codes.

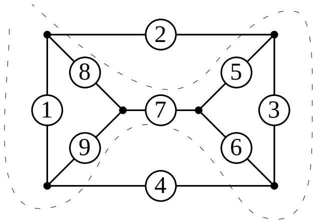


Figure 6.1 *A cycle code with an irreducible closed walk* (8,7,5,3,6,7,9,1) *that is not a cycle. The walk consists of two interconnected cycles, forming a "necktie".*

Theorem 6.1 After sufficiently many decoding iterations, a necessary condition for a decoding error to occur somewhere, i.e., $\mu_s(0) \geq \mu_s(1)$ for some s is that $\sum_{i=1}^{k} \gamma_{s_i} \leq 0$ for some irreducible closed walk $s_1, \ldots, s_k$.

(The proof is given in Appendix A.5.) Note that the theorem provides a necessary condition for decoding error, but not a sufficient condition. As for the other direction, we have the following weaker result.

Theorem 6.2 If, for all sites s and check sets E with $s \in E$, it holds that $\mu_{E,s}(1) - \mu_{E,s}(0) \to \infty$ as the number of updates tends to infinity, then there is no irreducible closed walk $s_1, \ldots, s_k$ for which $\sum_{i=1}^{k} \gamma_{s_i} \leq 0$.

(The proof is given in Appendix A.5.) The theorem only implies that if the necessary condition for decoding error that was given in Theorem 6.1 is actually met, then at least some intermediate cost function will not tend towards infinity, i.e., not provide a very clear "zero" decision. This does not rule out a correct decision, however, since a correct decision (i.e., deciding that the all-zero codeword was transmitted) only requires the final cost functions to be positive.

We now return to Theorem 6.1 and the condition for decoding error. The condition is met if the cost sum along some cycle is negative, since a cycle is an irreducible closed walk. We would not expect the decoder to decode correctly in that case, though, since even a maximum-likelihood decoder would make a decoding error. Consequently, if such errors (i.e., corresponding to cycles) were the only possible ones, then the min-sum algorithm would in fact find the maximum-likelihood codeword for cycle codes. However, as was illustrated by Figure 6.1, there may be irreducible closed walks that are not cycles. So, it is natural to ask if it is possible to have a negative cost sum along such a walk, while having positive cost sum along all cycles. This is in fact the case, as illustrated by Figure 6.2.

It is thus interesting to study irreducible closed walks that are not cycles, in order to estimate the decoding performance (compared to maximum-likelihood) of the min-sum algorithm for cycle codes. In particular, in view of (6.4), we would like to study the multiplicity vectors, (or generalized weights) of such irreducible closed walks. For any particular cycle code, there is a finite number of such walks, so the problem, in principle, is possible to solve.

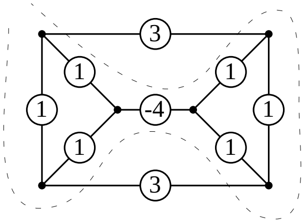


Figure 6.2 *An assignment of local costs with a positive cost sum along all cycles but a negative cost sum along the indicated irreducible closed walk.*

We are, however, more interested in theoretical results that apply to any cycle code. The following two results, which are proved in Appendix A.5, provides some insight to the structure of irreducible closed walks.

Lemma 6.3 No site occurs more than twice in an irreducible closed walk.

Lemma 6.4 Any non-cycle irreducible closed walk contains at least two cycles (which may be partially or completely overlapping, or disjoint).

From these two results, it appears to be "relatively unlikely" to have a negative cost sum along a non-cycle irreducible closed walk when the cost sums along all cycles are positive. The reasoning behind this is that the cost sum of such a walk can be written as the sum of the costs of some (at least two) positive terms corresponding to cycles in the walk, and a remaining term; to get a negative sum, the remaining term has to outweigh the sum of the other terms.

Unfortunately, we do not have a formal (nontrivial) result regarding the influence on the error probability of these non-cycle irreducible closed walks. To illustrate the complexity of the problem, we give in Figure 6.3 two non-cycle irreducible closed walks, both rather different from the one in Figure 6.1.

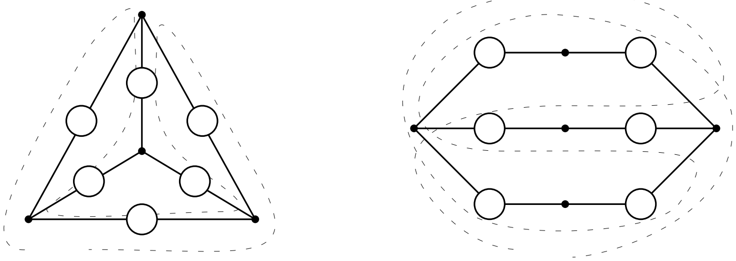


Figure 6.3 *Two examples of irreducible closed walks that are not cycles.*

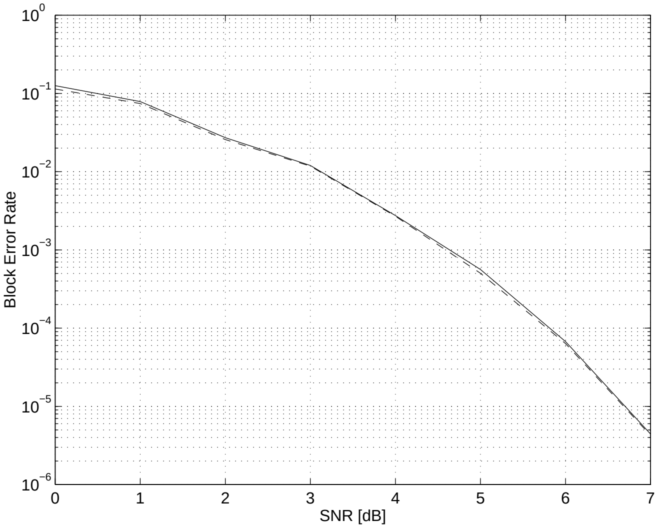


Figure 6.4 *Decoding performance of the (*15, 6, 5) *cycle code of the Peterson graph* [12, pp. 136-138] *used on a Gaussian channel. The upper curve corresponds to 30 iterations of the min-sum algorithm, and the lower curve corresponds to maximum-likelihood decoding.*

For very small codes, it is possible to use simulation to compare the performance of the min-sum algorithm with that of maximum-likelihood decoding. The result of such a simulation is illustrated in Figure 6.4, indicating that the min-sum algorithm is very close to maximum-likelihood for that code.

6.2 Tailbiting Trellises

The reason why it is possible to characterize the possible error events for cycle codes is that their deviations have a chain-like structure and may thus be partitioned into finite parts corresponding to a certain structure on the graphs. This structure is shared with the deviations of trellises (i.e., the detours). For ordinary (not tailbiting) trellises, which are cycle-free, this leads to the standard analysis of the distance structure, which can be found, e.g., in [21]. For tailbiting trellises, which are not cycle-free, we can use an analysis similar to that of cycle codes.

Recall that the walks corresponding to the deviations of cycle codes can be partitioned into irreducible closed walks. Similarly, deviations on a tailbiting trellis (i.e., detours) can be partitioned into closed subpaths whose length are multiples of the trellis length, each starting and ending in the same state; the deviation can only have negative cost if at least one such

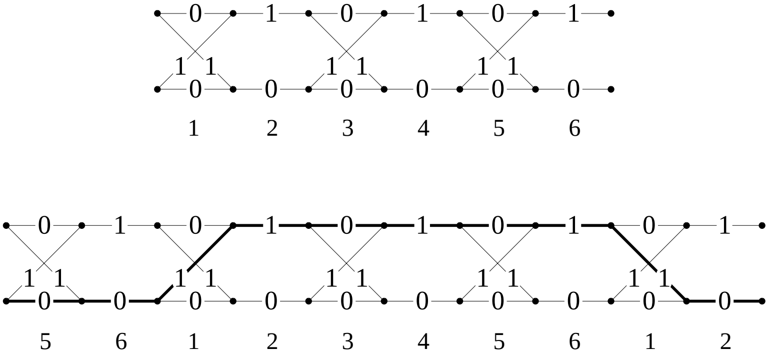


Figure 6.5 *A tailbiting trellis of length 6 (top) and the "computation trellis" corresponding to a computation tree of length 10 (bottom). The numbers below the trellis sections indicate the corresponding codeword component index. The marked path is valid on the computation trellis (in fact, it is a "detour") although it uses different branches in sections corresponding to the same original section.*

closed subpath has negative cost. The situation is more complex than for cycle codes, however, since detours may end before the leaves of the computation tree, i.e., they may have finite support even when the number of iterations tends to infinity.

So, consider the computation tree obtained with the min-sum algorithm on a tailbiting trellis. Let n be the number of trellis sections and let l be the number of sections in the computation tree. As long as $l \le n$, the computation tree is cycle-free, and the analysis of Chapter 5 applies. We are interested in the case when $l > n$. The valid configurations on the tree system are paths in a *computation trellis* which consists of multiple copies of the original tailbiting trellis, pasted together. See Figure 6.5.

The deviations on this tree system take the form of detours on the computation trellis, as discussed in Example 4.1. This means that if we follow the path from the nonzero root section towards one of the leaves (in either direction) then the path will either stay away from the zero state altogether or enter the zero state exactly once and stay in the all-zero path afterwards. The *length* of a detour is the number of nonzero branches.

As usual, we restrict ourselves to binary symbols and assume that the local costs for a "0" is zero ($\gamma_s(0) = 0$) so that the cost of a path equals the cost sum of its nonzero branches. (In fact, a branch is actually a local configuration on a section; its cost is the sum of the costs of its visible sites.)

We now define a *closed path* (in the computation trellis) as a subpath that starts and ends in the same state (of the same state spaces). Just as for cycle codes, a path is called *reducible* if it contains a closed path that may be removed, leaving a shorter path, so that the cost of the original path is the sum of the costs of the remaining shorter path and the closed path. A path is called *irreducible* if it does not contain a closed path; the lengths of irreducible paths are limited by nM, where M is the number of states in the largest trellis state space. Note that closed paths of length n correspond to valid paths on the original tailbiting trellis, i.e., to codewords.

With these definitions, it is easy to see that any path S on the computation trellis may be decomposed into a collection $S_1, \ldots, S_m$ of irreducible closed paths and a remaining irreducible path S_0, so that the cost $\mathcal{G}(S)$ is the sum $\mathcal{G}(S_0) + \mathcal{G}(S_1) + \ldots + \mathcal{G}(S_n)$. In particular, any detour may be decomposed in this way, and the remaining irreducible path will also be a detour (unless its length is zero). But since a decoding error can only occur if some detour has negative cost, we have proved

> **Theorem 6.5** Using the min-sum algorithm with a tailbiting trellis, a necessary condition for decoding error is that there is an irreducible detour (on the computation trellis) with negative cost.

As with the cycle codes, this provides a finite formulation of the problem of estimating the decoding performance, since there is a finite number of irreducible detours on the computation trellis of any given tailbiting trellis.

6.3 The General Case

While the above analysis for realizations with chain-like deviations is relatively successful, it is also quite limited since not many realizations have this structure. Ideally, we would want to have, for any realization, a similar characterization of possibly uncorrectable channel outputs. In particular, it would be interesting to obtain such a result for turbo codes. Unfortunately, the above analysis does not work in the general case, and we will now outline why. The discussion is based on the proof of Theorem 6.1 (cf. Section A.5). As we will see, the situation is not completely hopeless.

An essential idea behind the proof of Theorem 6.1 is that the cost of an infinite deviation (i.e., a deviation on an asymptotically growing tree system) is determined in the interior of the tree; any boundary effects at the root or at the leafs can be discarded. This is because the number of nonzero positions at a particular depth in a deviation is fixed (for any deviation, there are exactly two “ones” at any depth of the computation tree), so each depth of the computation tree contributes equally much to the sum (4.2) for the global cost of a tree configuration.

Since the realization is finite, an infinite deviation cannot behave completely irregularly; it must repeat itself somehow. For cycle codes, these repetitions form irreducible closed walks, which are the building blocks of any deviation. It appears natural to look for similar repeatable, or “closed” structures in the general case, with the property that any deviation could be decomposed into such structures.

The problem with this approach is that, in many cases, the cost of an infinite deviation is not determined in the interior of the tree but rather at the leafs, so any possible regularities in the deviations do not influence the global cost. This is the case if the support of the deviations are trees (and not walks), so that the number of nonzero positions at a particular depth grows exponentially with the depth.

As an example, consider a (3, 4) low-density parity-check code. The support of the deviations of the corresponding tree system (cf. Example 4.2 on page 32) are binary trees; the number of ones at depth d is 3×2^d. So, in the sum (4.2) for the global cost of a tree configuration, about half of the terms correspond to leaf sites; $^3/_4$ of the terms correspond to leaf sites or next-to-leaf sites, and so on. Therefore, the global cost is mainly determined by the sites near the leafs in the computation tree. And, as mentioned, any regularities inside the computation tree influence the global cost very little.

6.4 Turbo Codes

Turbo codes seem to fall somewhere between cycle codes, with their chain-like deviations, and more general constructions such as (3, 4) low-density parity-check codes, whose deviations have a tree structure. As illustrated in Figure 4.6 on page 36, some of the deviations of turbo codes are chain-like and some have support that are trees. Moreover, the chain-like deviations have lower weight than the others, and the weight increases with the number of branches in the tree.

This observation indicates that the chain-like deviations could be the “weak points” of the turbo decoding algorithm, i.e., they could, essentially, determine the error probability. In that case, we could use a similar method as for cycle codes to characterize the uncorrectable errors for the turbo decoding algorithm.

There are some difficulties with this approach, however, since the probability that a specific deviation e has a negative cost is not determined by its weight, but rather by its “generalized weight”, as defined in Equation 6.3, which depends on the cycle structure of the Tanner graph. There is no straightforward relation between the weight and the generalized weight of a deviation. For example, two infinite chain-like deviations of the same weight can differ significantly in their generalized weight if one has support in a large part of the Tanner graph while the other has support only in a single cycle. It would be even worse (for our purposes) with a tree-shaped deviation having most of its support concentrated on a subgraph that is *smaller* than a cycle, since such a deviation could then have lower generalized weight than any chain-like deviation. For the moment, we do not know if such deviations are possible.

Chapter 7

More on Code Realizations

The possibilities for constructing code realizations with Tanner graphs are enormous. Examples of parameters that may be varied are

- the Tanner graph,
- the number of hidden sites,
- the site alphabets, and
- the local behaviors.

Since the goal is to use the realizations for decoding, they should (ideally) provide good decoding performance while keeping the decoding complexity low. The connection between realization complexity and decoding complexity is relatively straightforward (cf. Section 3.3), except for the number of decoding iterations, which depends, of course, on how fast the decoding performance increases. Unfortunately, the results on the decoding performance have been rather meager so far. Still, it seems natural to look for realizations with large girth, since a larger girth gives a larger number of cycle-free decoding iterations. Indications on the connection between decoding performance and girth were given in [22] and [23] (graphs with large girth are discussed in, e.g., [24] and [25]). An obvious requirement for good decoding performance is, of course, that the theoretical code performance is good.

In summary, we are looking for realizations with

- low realization complexity,
- good theoretical code performance,
- no short cycles in the Tanner graph.

We begin this chapter by discussing the connection between realization complexity and the presence of cycles in the Tanner graph, as promised in Chapter 2.

7.1 Realization Complexity

It is well known that, for a given ordering of the time axis, a linear (block or convolutional) code (or, more generally, a group code) has a well-defined unique minimal trellis; any trellis for the same code can be collapsed (by state merging) to the minimal trellis (see for example [11]). An equivalent statement holds for realizations with an arbitrary cycle-free Tanner graph, which is easily seen by adapting the proofs of [11] to this case.

For Tanner graphs with cycles, however, this is no longer true. For instance, for tail-biting convolutional codes, it is easy to construct examples of inequivalent realizations (in this case: tail-biting trellises) for the same linear code that are minimal in the sense that no states can be merged [13, pp. 34-36]. Nevertheless, the theory of [11] is easily adapted to give a lower bound on the size of certain site alphabets (state spaces), which we will now develop.

Let (N, W, B) be a linear system with the check structure Q. We will use the notion of a cut: A *cut* on a check structure Q is a partition (I, K, J) of the sites in Q such that no check set in Q contains sites from both I and J. Informally, there is no walk from I to J that does not go through K. We will use the following obvious result about cuts:

> **Lemma 7.1** Let $(K ; I, J)$ be a cut on Q. Then any $x \in W$ is valid (i.e., in B) if and only if $x_{K \cup I} \in B_{K \cup I}$ and $x_{K \cup J} \in B_{K \cup J}$.

For a site subset $R \subseteq N$, we define $\tilde{B}_R \triangleq \{x \in B : x_{N \setminus R} = 0\}$, i.e., $\tilde{B}_R$ consists of those valid configurations that are zero outside R. We then have the following definition, which is adapted directly from [11]:

> **Definition 7.1** Let (I, J) be a partition of N. The *abstract state space between I and J* is the quotient space $S_{I,J}(B) \triangleq B/(\tilde{B}_I + \tilde{B}_J)$. The *abstract state between I and J* of a valid configuration $x \in B$ is the coset $\sigma_{I,J}(x) \triangleq x + (\tilde{B}_I + \tilde{B}_J)$.

For $N = \{1 \dots n\}$, the time-*j* state space of the minimal trellis for B (with the given order of the time axis) is in one-to-one correspondence with the abstract state space $S_{\{1\dots j\},\{j+1\dots n\}}(B)$. For general check structures, we have the following bound, which is the main result of this section:

> **Theorem 7.2** Let (N, W, B) be a linear system with the check structure Q. Let $(K ; I, J)$ be a cut on Q. Then, for any $I' \subseteq I$ and any $J' \subseteq J$, there exists a linear mapping from B_K onto $S_{I',J'}(B_{I' \cup J'})$. In particular, $\dim B_K \geq \dim S_{I',J'}(B_{I' \cup J'})$.

(The proof is given in Appendix A.6.) Of special interest is the case when K contains only hidden sites:

> **Corollary 7.3** Let (L, V, W, B) be a linear system with hidden sites and with a check structure Q. Let $(K ; I, J)$ be a cut on Q such that K contains no visible sites, i.e., $I' \triangleq I \cap V$ and $J' \triangleq J \cap V$ form a partition of V. Then $\sum_{s \in K} \dim A_s \geq \dim S_{I',J'}(B_V)$.

Note that this bound can simultaneously be applied to all cuts of the Tanner graph, which results in a linear programming bound on the dimensions of the site state spaces. Note further that bounds on the dimensions of the subcodes $\tilde{B}_R$ may be obtained from the dimension/length profile (generalized Hamming weights) of the code (cf. e.g. [26]), which in turn can be bounded by known bounds on the minimum distance of block codes. The full development of this approach is a challenging research problem; for us, Theorem 7.2 and Corollary 7.3 mainly serve to indicate why it is useful to consider realizations with cycles.

7.2 Cycle-Free Realizations

Despite the inherent complexity of cycle-free realizations (as shown in the previous section), they prove very useful for applications. The reason, of course, is the optimality of the min-sum and sum-product algorithms when used on such realizations. Therefore, it is interesting to search for cycle-free realizations with as low complexity as possible for a given output code. Much recent work has been devoted to finding the symbol orderings and "sectionalization" that give rise the smallest minimal trellises, see for example [27] and [28].

It appears natural to extend that work to include cycle-free realizations that are not trellises. As it turns out, some trellis realizations that are already proposed may be more naturally described with a different, non-trellis (but still cycle-free) Tanner graph. This is the case for some trellis realizations with large sections containing many parallel branches (cf. e.g. [27]). Moreover, the variation of the Viterbi decoding algorithm that is usually proposed for such realizations (where the parallel branches are processed in a first computation step before the trellis is processed in the usual way) is a direct application of the min-sum algorithm applied to the other cycle-free Tanner graph.

As an example, consider the "Cubing Construction" by Forney [29], where a code of length $3n$ is obtained from codes V, C, and D of length n such that a codeword x may be written as $x = (x_1, x_2, x_3) = (v_1 + c_1 + d, v_2 + c_1 + c_2 + d, v_3 + c_2 + d)$, where $v_1, v_2, v_3 \in V$, $c_1, c_2 \in C$, and $d \in D$. (The codes V, C, and D are obtained in a special way, which is not important for our discussion.) Forney [29] described these codes with a trellis consisting of three sections of length n, corresponding to x_1, x_2, and x_3; the intermediate states would then be $c_1 + d$ and $c_2 + d$, i.e., the information that is "shared" between the sections. See Figure 7.1.

However, the trellis description hides some of the symmetry of the construction since the middle section is seemingly different from the others. With a different Tanner graph (for the same system), the symmetry is transparent, see Figure 7.2. Moreover, the "variation" of the

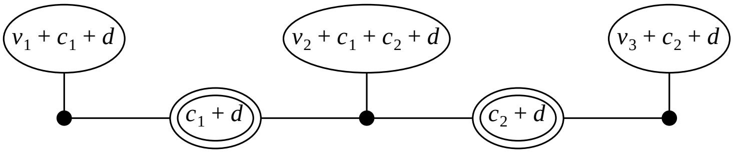


Figure 7.1 *Tanner graph for the trellis description of the "Cubing Construction". The expressions indicate the site values of a general valid configuration.*

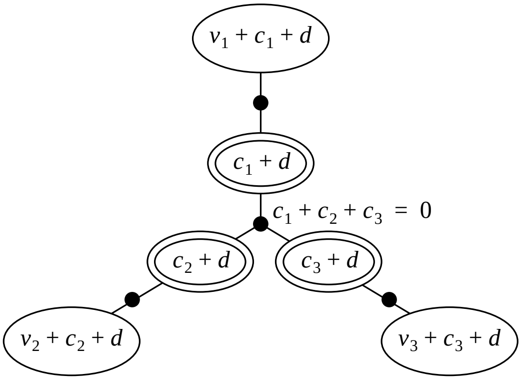


Figure 7.2 *A different, perhaps more natural Tanner graph for the "Cubing Construction". The expressions indicate the site values of a general valid configuration. The equation in the middle indicates the local behavior of the middle check.*

Viterbi algorithm proposed by Forney in [29], which in many cases uses fewer operations than what is possible with a normal trellis, is a direct application of the min-sum algorithm to the Tanner graph of Figure 7.2.

From this example and the recent work on trellis sectionalization [27], it is clear that it is often advantageous to allow large trellis sections with many parallel branches. From our perspective, this indicates that non-trellis cycle-free realizations may sometimes be less complex than trellis realizations for the same codes. Arguably, any such realization could be seen as a trellis with (perhaps extremely) large sections; with the use of Tanner graphs, on the other hand, the underlying structure may be more transparent.

7.3 Realizations with Cycles

The benefits (in terms of complexity) of allowing cycles in the Tanner graphs are illustrated nicely by tailbiting trellises [4], which are the perhaps "simplest" extension from cycle-free Tanner graphs. As mentioned in the introduction (Section 1.1), a tailbiting trellis contains several starting states, and equally many ending states; a path is required to start and end in the same state. Figure 7.3 illustrates the Tanner graph for the tailbiting trellis of Figure 1.3.

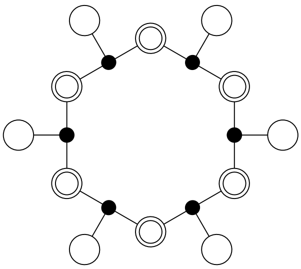


Figure 7.3 *The Tanner graph of a small tailbiting trellis.*

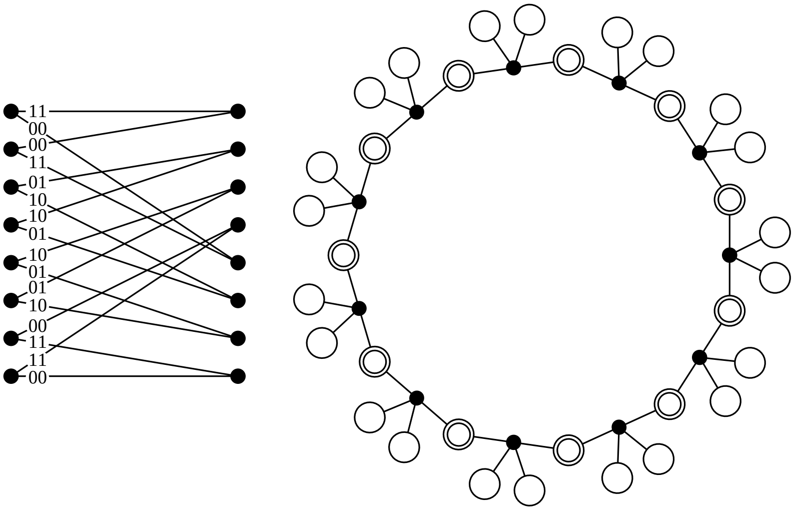


Figure 7.4 *Tailbiting trellis realization of a* (22, 11, 7) *subcode of the binary Golay code. The eight-state trellis section to the left is repeated 11 times and "tied together" at the ends, forming the Tanner graph shown to the right.*

It is well-known (cf. [4]) that for some codes, the number of states in a tailbiting trellis may be much lower than the number of states in any ordinary trellis. For example, it is demonstrated in [4] that a (22, 11, 7) subcode of the Golay code has a tailbiting trellis with as few as eight states, as shown in Figure 7.4. In view of Corollary 7.3, this means that the abstract state space of that code, with respect to a cut that divides the Tanner graph in two halves, has dimension at most six (since the abstract state space must be spanned by the two hidden site alphabets in the cut, each having dimension three). In fact, the dimension of the abstract state space is exactly six for many such cuts, implying that a trellis (not tailbiting) must have 64 states in some state spaces.

Even further complexity reductions are possible with more complicated Tanner graphs with more connections. Figure 7.5 illustrates a "wheel-shaped" Tanner graph for a four-state realization of a (22, 11, 7) subcode of the binary Golay code. The realization consists of a tailbiting trellis with some of the output bits shared across the graph, thereby creating a structure which is locally identical to the turbo code realization (cf. Figures 2.4 and 2.5); in particular, the computation trees are exactly the same (cf. Section 4.2.1). The underlying trellis section (which is repeated 11 times) is shown in Figure 7.6. As with the turbo codes, every other redundancy bit is punctured to achieve the rate 1/2.

To see that this realization actually yields the desired (22, 11, 7) code, consider the generator matrix for that code, as given in [30, pp. 509]. It is a double circulant matrix of the form $G = (I|P)$ where the top row of P is $(1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0)$.

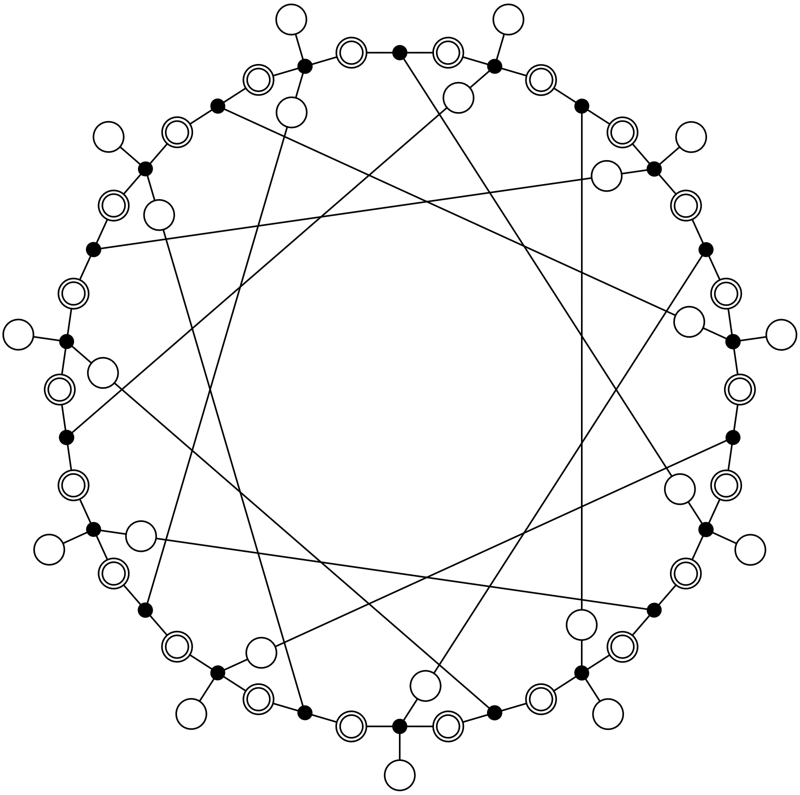


Figure 7.5 *The Tanner graph of a "turbo-style" realization of the* (22, 11, 7) *subcode of the Golay code. The trellis section is shown in Figure 7.6. The rightmost bit of the trellis section is punctured in every other occurrence of the section.*

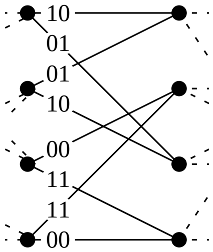


Figure 7.6 *Trellis diagram for the realization in Figure 7.5. The leftmost bit corresponds to the connected sites in the interior of Figure 7.5, while the rightmost bit corresponds to the leaf sites on the outside of the Tanner graph (some of which are punctured away).*

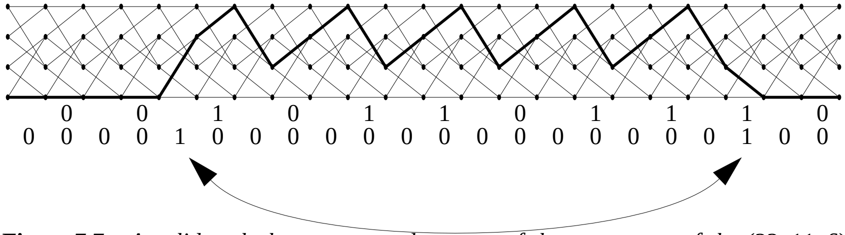


Figure 7.7 *A valid path that corresponds to one of the generators of the* (22, 11, 6) *code. The 22 lower bits correspond to the interior sites in Figure 7.5; note that the single "one" occurs at two places in the trellis. The 11 upper bits correspond to the redundancy sites on the outside of the wheel.*

We will show that the leftmost 11 columns of G (the information bits) correspond to the connected sites of the realization whereas the rightmost 11 columns (the redundancy bits) correspond to the remaining sites. To do this, we will show that the 11 connected sites determine the remaining sites uniquely, and that a single "one" placed in any of the interior sites corresponds to a row of G, i.e., it is a generator of the (22, 11, 7) code.

To see that the interior sites determine the entire configuration, consider the tailbiting trellis (Figure 7.7) and a path that is zero in all the information bits. From Figure 7.6, it is clear that such a path must be either the all-zero path, or a path that circulates among the nonzero states with a period of three sections. But a path cannot have period three, because there are 22 sections in the tailbiting trellis. Thus, there is only one path with zeros in the information bits; it follows by linearity that any valid configuration on the interior sites (the information bits) determines the rest of the configuration uniquely.

Next, assume that a single interior site is "one" (the remaining sites are zero). A corresponding valid configuration, i.e. a path with "ones" in two connected positions, is shown in Figure 7.7. This valid configuration corresponds to a row of G (the upper bits are found to the right in the matrix). The remaining rows of G correspond to cyclic shifts of this valid configuration.

As mentioned, the realization of Figure 7.5 is rather similar to the turbo code realization (the difference is that the turbo codes consist of two terminated trellises which are interconnected). In particular, the computation tree has the same structure as that of a turbo code. We now consider the "wheel construction" more generally, allowing different connection patterns and other trellis sections. A potential advantage of this variation of turbo codes is that it permits constructions with high symmetry, which may simplify both analysis and computer search for good realizations.

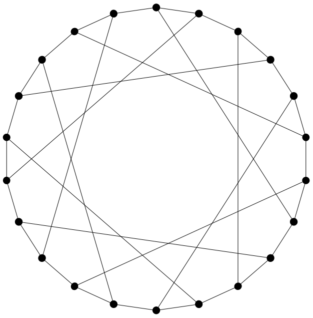


Figure 7.8 *The 3-regular graph underlying the realization of Figure 7.5.*

7.3.1 The Wheel Construction

The following ingredients are needed to define the wheel construction in general.

- A regular graph of degree 3 with a Hamiltonian cycle (a cycle containing all vertices exactly once), and
- a trellis section with one bit considered as information bit (i.e., the left state and the information bit determine the redundancy bits and the right state uniquely).

The Tanner graph is constructed from these ingredients by associating a hidden site with each edge in the Hamiltonian cycle and visible sites with the remaining edges. As in Figure 7.5, the Hamiltonian cycle is laid out in a circle, forming the "time axis" for a trellis, with the remaining edges extending across the circle. Each vertex of the original graph is a section of the trellis (i.e., a check set of the Tanner graph), and the redundancy bits of the trellis sections are appended as leaf sites; often, some of them will be punctured (i.e., not transmitted) and thus need not be included in the realization at all. The realization of Figure 7.5 was obtained in this way from the graph of Figure 7.8.

As a side note, cycle codes appear as a special case of the wheel construction, with the two-state trellis section shown in Figure 7.9. Following the general wheel construction, the leftmost bit is considered as the information bit and the corresponding visible site is shared with some other section, while the rightmost bit is considered as a redundancy bit and its visible site is a leaf site. The point is that the right state of the section is identical to the redundancy bit, so instead of a visible leaf site at each section we can simply make the state sites visible. But then the local behavior is just a parity check on three visible sites (the information site and the left and right state sites) and we have obtained a cycle code.

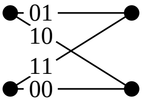


Figure 7.9 *A simple two-state trellis section.*

Naturally, more powerful codes can be obtained by using larger state spaces. We have made some investigations of the wheel realizations obtained with the four-state trellis section of Figure 7.6. To limit the possibilities further, we have required the graphs to have a periodic structure, as follows. Let p be a positive even integer, the *period*, and consider a graph of order n with a Hamiltonian cycle, where $p|n$. Number the vertices consecutively along the Hamiltonian cycle with elements from $\mathcal{Z}_n$, i.e., from 0 to $n-1$ modulo n. (Thus, there is an edge between the vertices i and $i+1$, for any integer i.) The remaining edges (those that correspond to the connected "information" sites) are determined by p integers $k_0, \ldots, k_{p-1}$ such that for any i, there is an edge between the vertices i and $i + k_{\mathrm{rem}(i,\, p)}$ where $\mathrm{rem}(i, p)$ is the remainder of i mod p. For example, in the graph of Figure 7.8, we have $n = 22$, $p = 2$, $k_0 = 7$, and $k_1 = 15$.

(Not all value combinations of $k_0, \ldots, k_{p-1}$ are valid. For example, for $p = 2$, k_0 must be odd (unless $k_0 = n/2$) and k_1 is given by $k_0 + k_1 \equiv 0 \pmod{n}$, to make the graph 3-regular. For general p, since the edge between the vertices i and $i + k_{\mathrm{rem}(i,\, p)}$ must be the same edge as the one between vertices j and $j + k_{\mathrm{rem}(j,\, p)}$ for $j = i + k_{\mathrm{rem}(i,\, p)}$, we have the requirement $k_{\mathrm{rem}(i,\, p)} + k_{\mathrm{rem}(j,\, p)} \equiv 0 \ (\mathrm{mod}\, n)$.)

Table 7.1 *Some codes obtained from wheel realizations with period 2 and the trellis section of Figure 7.6.*

				Code parameters	
n	k_0	k_1	girth	unpunctured	punctured
12	5	7	4	(18, 6, 6)	(12, 6, 4)
14	5	12	6	(21, 7, 7)	(14, 7, 3)
14	7	7	4	(21, 7, 6)	(14, 7, 4)
16	5	11	6	(24, 8, 8)	(16, 8, 5)
16	7	9	4	(24, 8, 8)	(16, 8, 5)
18	5	13	6	(27, 9, 8)	(18, 9, 5)
20	5	15	6	(30, 10, 9)	(20, 10, 5)
20	7	13	6	(30, 10, 9)	(20, 10, 5)
20	9	11	4	(30, 10, 9)	(20, 10, 5)
22	5	17	6	(33, 11, 10)	(22, 11, 5)
22	7	15	6	(33, 11, 11)	(22, 11, 7)
22	9	13	6	(33, 11, 9)	(22, 11, 5)

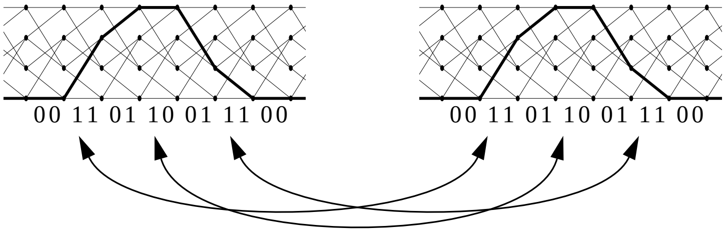


Figure 7.10 *A low-weight configuration. The arrows indicate the connected "ones".*

Table 7.1 lists some of the realizations obtained with the period $p = 2$ and the four-state trellis section of Figure 7.6, including the realization of Figure 7.8. For each realization, we have considered both the output code obtained by including all bits of the trellis sections as visible sites, and the output code obtained when every second redundancy site is punctured.

In Table 7.1, the realization obtained with $k_0 = 7$ and $k_1 = 15$ (cf. Figure 7.8) gives the largest minimum distances, both in the punctured ($d = 7$) and unpunctured ($d = 11$) versions. In fact, these are the largest possible minimum distances for realizations with period 2 and the trellis section of Figure 7.6. The reason for this is that, with period 2, there is an information pattern that gives a codeword of weight at most 7 (in the punctured version, 11 in the unpunctured version) regardless of n, k_0, and k_1. As illustrated in Figure 7.10, the nonzero information bits are placed such that two identical low-weight detours are obtained. For unpunctured codes, this gives a weight of at most 11 (remember that the information bits are shared); for punctured codes, the weight is at most 7. The minimum weight may often be smaller than 7 or 11, as seen in Table 7.1. (For small graphs, for instance, the two detours obtained in this way may in fact overlap, giving an even lower weight. And, of course, there may be other configurations with low weight, too.)

Similar types of information patterns exist for any period p, but in general, a larger value of p gives a larger upper bound to the minimum distance. For example, for $p = 4$, the largest possible minimum distance is 15 for the unpunctured version and 9 for the punctured version. Table 7.2 lists some codes obtained with $p = 4$.

Table 7.2 *Some codes obtained from wheel realizations with period 4 and the trellis section of Figure 7.6.*

						Code parameters	
n	k_0	k_1	k_2	k_3	girth	unpunctured	punctured
40	15	33	7	25	8	(60, 20, 11)	(40, 20, 7)
52	19	41	11	33	8	(78, 26, 15)	(52, 26, 8)
56	23	13	43	33	8	(84, 28, 15)	(56, 28, 9)
64	23	13	51	41	8	(96, 32, 15)	(64, 32, 9)

7.4 Modeling Complicated Channels

Although our main focus in the thesis is on single-user communication on memoryless channels, we also wish to point out that the framework of realizations based on Tanner graphs may be useful to model more complicated situations, in particular channels with memory. The idea is to model the random behavior of the channel with local check cost functions.

As an example, consider the Gilbert-Elliot channel model [31], which is a binary symmetric channel with two different crossover probabilities, depending on whether the channel is in the good or in the bad state, and some probability of going from the good to the bad state and vice versa. The Gilbert-Elliot channel model is a simple model of channels generating error bursts. One way of handling the error bursts is to reorder the encoded symbols with an interleaver before they are sent on the channel (and reorder the channel output back again before decoding). This way, the decoder only “sees” the mean crossover probability (weighted by the marginal probabilities of the two states) provided that the interleaving is long enough. In the simplest case, the decoder just operates as if the channel was truly memoryless, i.e., it searches for the codeword which corresponds to the fewest number of errors.

The disadvantage of this simple procedure is that some information about the channel is discarded, leading to an inefficient system. Assume, for a moment, that the decoder knows the true state of the channel at each symbol interval. Then this information could be used to improve the performance by relying more on the good symbols and less on the bad ones. Of course, the true states of the channel are inaccessible; it is possible, however, to estimate the channel states given the channel output and the code structure. One way of doing this is to use the interleaving scheme described above and an iterative decoding method, as follows. First, the deinterleaved channel output is decoded as if the channel was memoryless. Then the result of this decoding is fed back to a model of the channel, and used in a channel estimation procedure based on this model. The channel estimates can then be used to obtain a better decoding result, which may be used to get an even better channel estimate, and so on.

The sum-product algorithm and realizations with Tanner graphs seem to be an ideal tool for modeling such a decoding procedure, as follows. We first consider the channel model only. Let $Z = (Z_1, \ldots, Z_n)$ be the channel error vector and let $S = (S_1, \ldots, S_n)$ be the sequence of channel states. Let ε_1 and ε_2 be the crossover probabilities in the two channel states, let p_{12} and p_{21} be the transition probabilities (p_{12} is the probability of going from state 1 to state 2), and let p_{s_1} be the probability of starting in the state s_1. Then the joint probability distribution of the pair (S, Z) may be written as

$$p_{S,Z}(s, z) = p_{s_1} \prod_{i=1}^{n-1} p_{s_i, s_{i+1}} \prod_{i=1}^{n} \varepsilon_{s_i}^{z_i} (1 - \varepsilon_{s_i}^{1-z_i}) . \qquad (7.1)$$

This probability may actually be seen as the global cost of a “configuration” (s, z), using the multiplicative global cost function as defined for the sum-product algorithm in Section 3.2. The Tanner graph used for that global cost function is shown in Figure 7.11; the factors of (7.1) appear as local check (and site) cost functions, which are also shown in the figure.

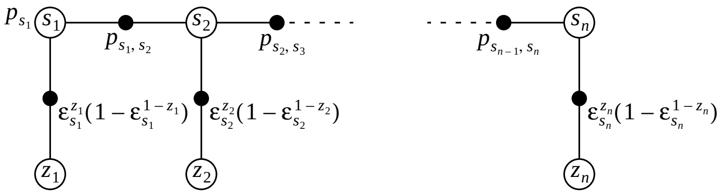

p_{s_1}
s_1
p_{s_1,s_2}
s_2
p_{s_2,s_3}
p_{s_{n-1},s_n}
s_n
$\varepsilon_{s_1}^{z_1}(1-\varepsilon_{s_1}^{1-z_1})$
$\varepsilon_{s_2}^{z_2}(1-\varepsilon_{s_2}^{1-z_2})$
$\varepsilon_{s_n}^{z_n}(1-\varepsilon_{s_n}^{1-z_n})$
z_1
z_2
z_n

Figure 7.11 *A Tanner graph for the Gilbert-Elliot channel model, with local check (and site) cost functions.*

We now include the code in the model. Let $X = (X_1, \ldots, X_n)$ be an interleaved version of a random codeword (uniformly distributed over the code), and let $Y = (Y_1, \ldots, Y_n) = X + Z$ be the received word (S and Z are defined as before). Then the a posteriori distribution of X conditioned on Y is

$$p_{X|Y}(x|y) = \frac{p_{X,Y}(x,y)}{p_Y(y)} = \frac{1}{p_Y(y)}\sum_s p_{X,Y,S}(x,y,s), \tag{7.2}$$

where the sum is over all channel state sequences s. The terms of the sum can be written as

$$p_{X,Y,S}(x,y,s) = p_X(x)p_S(s)p_{Y|X,S}(y|x,s) = p_X(x)p_{s_1}\prod_{i=1}^{n-1}p_{s_i,s_{i+1}}\prod_{i=1}^{n}\varepsilon_{s_i}^{y_i-x_i}(1-\varepsilon_{s_i}^{1-y_i+x_i}), \tag{7.3}$$

using the fact that X and S are statistically independent. This probability distribution can be modeled by a global cost function $G(x, s)$, which depends on the channel output y. The corresponding Tanner graph is illustrated in Figure 7.12. The code structure is represented by a trellis and local check cost functions of the “indicator” type, as discussed in Section 3.2, so that the trellis part of G is $p_X(x)$.

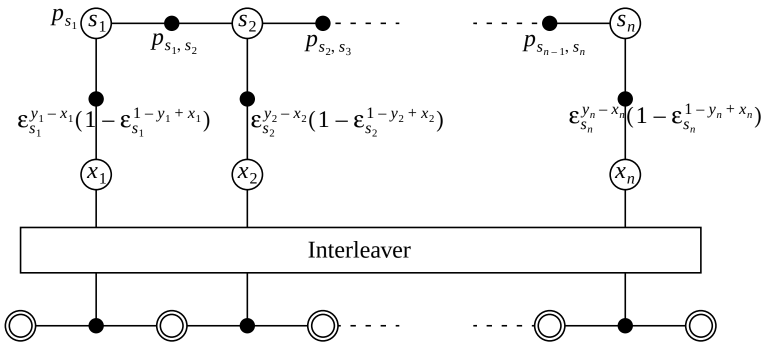

p_{s_1}
s_1
p_{s_1,s_2}
s_2
p_{s_2,s_3}
p_{s_{n-1},s_n}
s_n
$\varepsilon_{s_1}^{y_1-x_1}(1-\varepsilon_{s_1}^{1-y_1+x_1})$
$\varepsilon_{s_2}^{y_2-x_2}(1-\varepsilon_{s_2}^{1-y_2+x_2})$
$\varepsilon_{s_n}^{y_n-x_n}(1-\varepsilon_{s_n}^{1-y_n+x_n})$
x_1
x_2
x_n
Interleaver

Figure 7.12 *Model for the system with trellis code (bottom), interleaver, and channel (top). Note that the channel output $y_1, \ldots, y_n$ influences the local check cost functions.*

In particular, the projection of G onto a codeword component X_i is proportional to the a posteriori distribution of X_i conditioned on Y, i.e.,

$$p_{X_i|Y}(a|y) = \sum_{x\,:\,x_i = a} p_{X|Y}(x|y) = \frac{1}{p_Y(y)} \sum_{x\,:\,x_i = a} \sum_{s} G(x, s). \tag{7.4}$$

So, if the realization had been cycle-free, the sum-product algorithm would have computed, as its final cost functions, the a posteriori distributions (7.4) (Theorem 3.2), thereby allowing optimal (MAP) decoding of the codeword components X_i conditioned on the channel output Y. Obviously, the realization contains cycles, so we cannot use Theorem 3.2 directly; still, by using a suitable interleaving scheme, it is possible to avoid short cycles in the Tanner graph, thereby allowing at least a few cycle-free decoding iterations, as discussed in Chapter 5.

For the moment, this seems like a promising direction for future research.

Chapter 8

Conclusions

The min-sum and sum-product algorithms are two fundamental algorithms that appear in many versions, in many fields, and under different names. In particular, many well-known decoding algorithms, such as Gallager's decoding algorithms for low-density parity-check codes, the Viterbi algorithm, the forward-backward algorithm, and turbo decoding are special cases of either the min-sum or the sum-product algorithm. This observation is the main contribution of this thesis, along with a general formulation of the two algorithms, within the framework of "realizations" based on "Tanner graphs".

The fundamental theoretical property of the min-sum and sum-product algorithms, their optimality on cycle-free Tanner graphs, has been stated and proved in a general setting. While cycle-free Tanner graphs have the advantage of yielding optimal decoders, such as the Viterbi algorithm, they have a severe disadvantage regarding decoding complexity, which increases exponentially with the minimum distance of the code. Tanner graphs with cycles, on the other hand, offer significantly lower complexity, but the resulting "iterative" decoders are generally suboptimal.

A major part of the thesis has been devoted to performance analysis of iterative decoding. Unfortunately, the results are far from conclusive; some new results have been obtained, though. In particular, the union bound commonly used with the Viterbi algorithm is shown to be applicable to the min-sum algorithm in general, at least for the first few decoding iterations; the bound seems rather weak, however. For further decoding iterations, the cycles of the Tanner graph begin to influence the decoding process, which makes the analysis much more difficult. Still, for a limited class of codes, we have characterized the decoding errors remaining after infinitely many decoding iterations; the result suggests that the performance is relatively close to maximum-likelihood, which is also verified by simulations. This result is not strictly applicable to turbo codes; on the other hand, some similarity between turbo codes and the codes in this class exists, indicating that the success of turbo decoding might be partly "explained" by this result. In summary, our new results on decoding performance might provide additional understanding of the underlying mechanisms, although they fail to predict the amazing decoding performance of turbo coding, which is still only demonstrated by simulations.

Since the performance of turbo coding (as indicated by theoretical code performance [8] and simulation of decoding [7]) is so close to the Shannon limit, it is an adequate question whether further research on coding methods is interesting. Indeed, in situations involving single user communication on memoryless channels, and where very long block lengths are tolerable, it seems unlikely that further improvements could have any practical significance. (Of course it would still be desirable to obtain a better theoretical understanding, if for no other reason than to provide guidance when designing turbo coding systems for particular applications.)

On the other hand, many applications (such as telephony) actually demand relatively short block lengths (under 1000 bits). In this area, it appears promising to look at explicit code constructions (rather than the random construction used in turbo codes, which works well for very long blocks). A particular example is the work of Kötter and Nilsson [22][23], which indicates that interleaving schemes obtained from graph theory may provide better performance than random interleavers for moderate block lengths. In Section 7.3.1, we pointed out a slightly different way of constructing short turbo codes.

Another research area that appears promising is coding in more complicated situations, for example multiuser communication and channels with memory. As pointed out in Section 7.4, the framework of realizations on Tanner graphs is a powerful and straightforward tool for constructing iterative decoding schemes in such situations.

Appendix A

Proofs and Derivations

A.1 Proof of Theorems 3.1 and 3.2

We will actually only carry out the proof for Theorem 3.1, i.e., for the min-sum algorithm. The proof is readily adapted to the sum-product algorithm by replacing all sums with products and all minimizations with sums.

Let Q be a cycle-free check structure for the system (N, W, B). For any site subset $R \subseteq N$, we will use the notation

$$G_R(x_R) \triangleq \sum_{E \in Q : E \subseteq R} \gamma_E(x_E) + \sum_{s \in R} \gamma_s(x_s) \tag{A.1}$$

for the part of the global cost that corresponds to R. (We then have $G_N(x) = G(x)$ as a special case.)

To prove the theorem, we will start with the claimed expression (3.6) for the final site cost functions. (The final check cost functions will be discussed afterwards.) This expression will be broken down recursively, working backwards through the algorithm, until only the local cost functions remain. In order to keep the notation transparent, the proof is carried out only for the specific check structure of Figure A.1; the generalization to arbitrary check structures will be obvious.

The final site cost functions μ_s

Consider the final cost $\mu_1(a)$ for some value $a \in A_1$ at site 1 in Figure A.1. Let $\nu_1(a)$ denote the claimed value for the final cost, i.e., $\nu_1(a) \triangleq \min_{x \in B : x_1 = a} G(x)$. We write this expression as a sum of three terms, patterned after the updating rule:

$$\nu_1(a) = \min_{x \in B : x_1 = a} G(x) \tag{A.2}$$

$$= \min_{x \in B : x_1 = a} \{\gamma_1(a) + G_{R_1}(x_{R_1}) + \gamma_{E_1}(x_{E_1}) + G_{R_2}(x_{R_2}) + \gamma_{E_2}(x_{E_2})\} \tag{A.3}$$

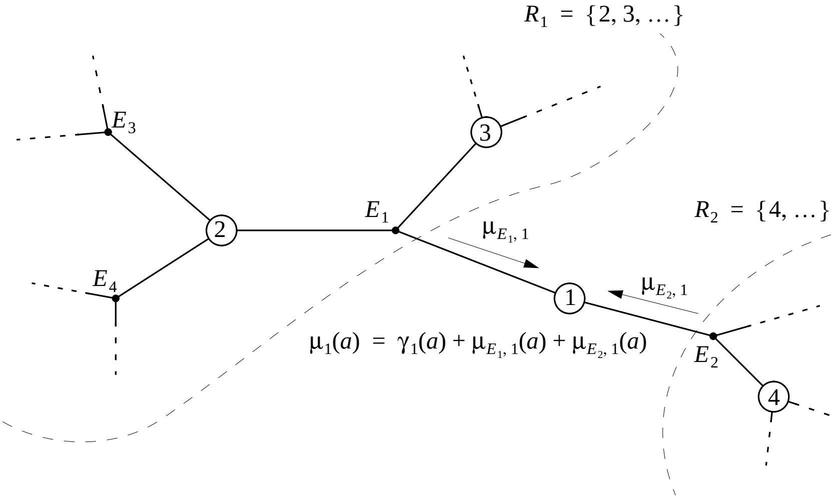


Figure A.1 *Computation of the final site cost* $\mu_1(a)$.

$$= \gamma_1(a) + \underbrace{\min_{x_{R_1 \cup \{1\}} \in B_{R_1 \cup \{1\}} : x_1 = a} [G_{R_1}(x_{R_1}) + \gamma_{E_1}(x_{E_1})]}_{\nu_{E_1, 1}(a)} + \underbrace{\min_{x_{R_2 \cup \{1\}} \in B_{R_2 \cup \{1\}} : x_1 = a} [G_{R_2}(x_{R_2}) + \gamma_{E_2}(x_{E_2})]}_{\nu_{E_2, 1}(a)} \tag{A.4}$$

This expression has the same structure as the updating rule for the final costs (cf. Figure A.1), and it follows that the final cost functions indeed get the claimed value ν_1, provided that the cost functions coming into site 1 have the right value, i.e., provided that $\mu_{E_1, 1} = \nu_{E_1, 1}$ and $\mu_{E_2, 1} = \nu_{E_2, 1}$. Due to the symmetry of the situation, it suffices to consider $\mu_{E_1, 1}$.

The check-to-site cost functions $\mu_{E, s}$

Figure A.2 illustrates the computation of $\mu_{E_1, 1}(a)$, which we claim equals $\nu_{E_1, 1}(a)$. The latter can be broken up into three independent parts: the local check-set cost $\gamma_{E_1}(x_{E_1})$ and two costs associated with the site subsets R_2 and R_3, respectively:

$$\nu_{E_1, 1}(a) = \min_{x_{R_1 \cup \{1\}} \in B_{R_1 \cup \{1\}} : x_1 = a} [G_{R_1}(x_{R_1}) + \gamma_{E_1}(x_{E_1})] \tag{A.5}$$

$$= \min_{x_{R_1 \cup \{1\}} \in B_{R_1 \cup \{1\}} : x_1 = a} [\gamma_{E_1}(x_{E_1}) + G_{R_2}(x_{R_2}) + G_{R_3}(x_{R_3})] \tag{A.6}$$

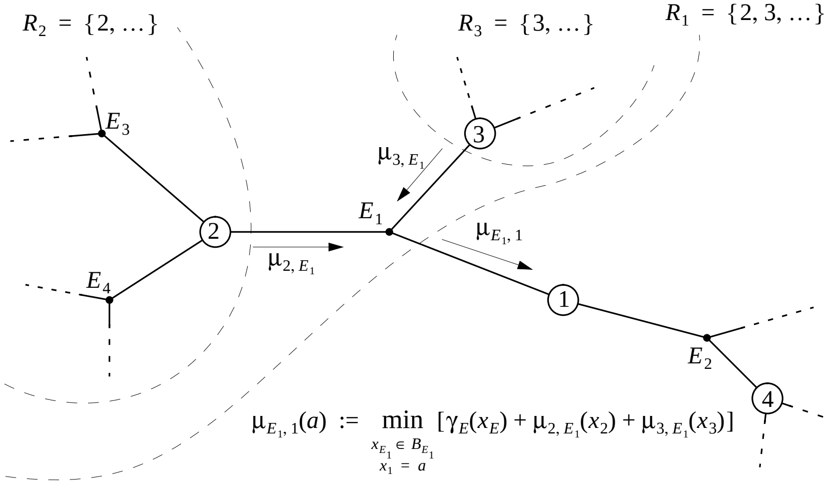


Figure A.2 *Computation of the check-to-site cost* $\mu_{E_1,1}(a)$.

$$\min_{{}_1 \in B_{E_1} : x_1 = a} \left| \gamma_{E_1}(x_{E_1}) + \underbrace{\min_{x'_{R_2} \in B_{R_2} : x'_2 = x_2} G_{R_2}(x'_{R_2})}_{\nu_{2,E_1}(x_2)} + \underbrace{\min_{x'_{R_3} \in B_{R_3} : x'_3 = x_3} G_{R_3}(x'_{R_3} \cdot}_{\nu_{3,E_1}(x_3)} \right. \tag{A.7}$$

Again, this expression has the same structure as the updating formula for $\mu_{E_1,1}$, cf. Figure A.2. To prove that the updating indeed computes $\nu_{E_1,1}$, we will show that the input costs to the updating rule are the same as the terms of (A.7), i.e., that $\mu_{2,E_1} = \nu_{2,E_1}$ and $\mu_{3,E_1} = \nu_{3,E_1}$. Due to the symmetry of the situation, it suffices to consider μ_{2,E_1}.

The site-to-check cost functions $\mu_{s,E}$

Figure A.3 illustrates the computation of $\mu_{2,E_1}(a)$, which we claim equals $\nu_{2,E_1}(a)$. The latter can be broken up into three independent parts: the local site cost $\gamma_2(a)$ and two costs associated with the site subsets R_3 and R_4, respectively:

$$\nu_{2,E_1}(a) = \min_{x_{R_2} \in B_{R_2} : x_2 = a} G_{R_2}(a) \tag{A.8}$$

$$= \min_{x_{R_2} \in B_{R_2} : x_2 = a} \{\gamma_2(a) + G_{R_3}(x_{R_3}) + \gamma_{E_3}(x_{E_3}) + G_{R_4}(x_{R_4}) + \gamma_{E_3}(x_{E_3})\} \tag{A.9}$$

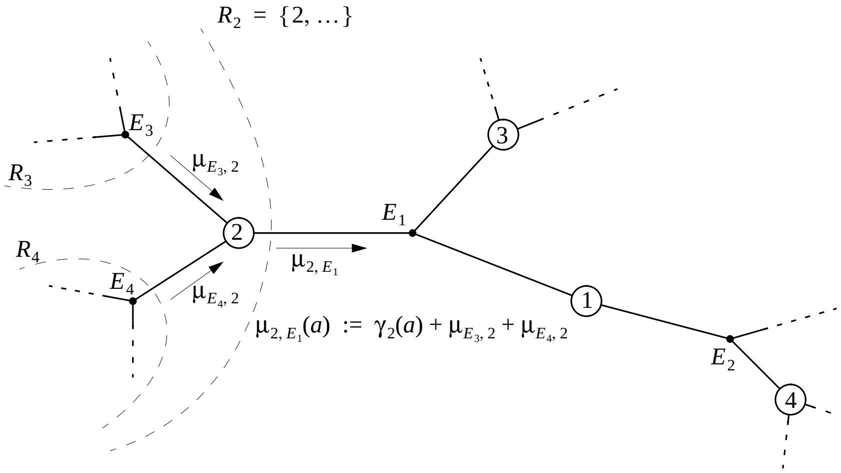


Figure A.3 *Computation of the site-to-check cost* $\mu_{2,E_1}(a)$.

$$= \gamma_2(a) + \underbrace{\min_{x_{R_3 \cup \{2\}} \in B_{R_3 \cup \{2\}} : x_2 = a} [G_{R_3}(x_{R_3}) + \gamma_{E_3}(x_{E_3})]}_{\nu_{E_3,2}(a)} + \underbrace{\min_{x_{R_4 \cup \{2\}} \in B_{R_4 \cup \{2\}} : x_1 = a} [G_{R_4}(x_{R_4}) + \gamma_{E_3}(x_{E_3})]}_{\nu_{E_4,2}(a)} \tag{A.10}$$

Again, this expression has the same form as the updating rule for μ_{2,E_1}, and we only need to show that the input to the updating rule has the value of the last terms of (A.10). This can be done by going back to the previous step, "*the check-to-site cost functions* $\mu_{E,s}$". This process is repeated recursively until the leaf sites are reached (those that belong to a single check set), where we observe that the very first updating step computes the desired expression $\nu_{s,E}$:

$$\nu_{s,E}(x_s) = \min_{x_s \in B_s} G_{\{s\}}(x_s) = \gamma_s(x_s) . \tag{A.11}$$

This completes the proof for the final site costs. For the final check-set costs $\mu_E(a)$, only the first step of the proof (the last step of the algorithm) needs to be changed, i.e., we need to start with the claimed expression (3.7) for the final check-set cost. Thus, let E be any check set (in a cycle-free check structure), and let R_s, $s \in E$, be the set of sites that are reachable from E through s. Then the claimed expression can be written as

$$\min_{x \in B : x_E = a} G(x) = \min_{x \in B : x_E = a} \gamma_E(a) + \sum_{s \in E} G_{R_s}(x_{R_s}) = \gamma_E(a) + \sum_{s \in E} \min_{x \in B : x_E = a} G_{R_s}(x_{R_s}), \tag{A.12}$$

which matches the formula for $\mu_E(a)$, and the recursive process of above can be applied to the terms of the last sum in (A.12).

A.2 Derivations for Section 3.3

We will consider the complexity of the algorithms presented in terms of the number of binary operations, such as additions, multiplications, and taking the minimum of two values. It is easy to see that adding n numbers requires $n - 1$ binary additions; similar results hold for multiplication and taking minimum. By examining the updating rules (3.1)–(3.4) and (3.10)–(3.13), it is also easy to verify that the min-sum and sum-product algorithms require exactly the same number of binary operations (although the complexity of each operation may be higher for the sum-product algorithm). Therefore, in the following we will only mention the min-sum algorithm. Further, we will not consider the computation of the local costs, and we will (for simplicity) assume that there are no local check costs.

Let (N, W, B) be a system with the check structure Q. Let $\mathcal{O}(\mu)$ denote the number of binary operations required to compute the cost function μ. For a site s, let $|s|$ be the *degree* of s, i.e., the number of check sets that s appears in. By simply counting the number of operations specified by the update rules, we get the complexity for updating the intermediate site-to-check cost functions $\mu_{s,E}$, $E \in Q$ and $s \in E$, as

$$\mathcal{O}(\mu_{s,E}) = |A_s|(|s| - 1) \text{ if } s \text{ is visible,} \tag{A.13}$$

$$\mathcal{O}(\mu_{s,E}) = |A_s|(|s| - 2) \text{ if } s \text{ is hidden.} \tag{A.14}$$

The difference between (A.13) and (A.14) arises because we have to add the local costs $\gamma_s(a)$ (i.e., the channel information) when s is visible. Note that $\mathcal{O}(\mu_{s,E})$ is independent of the check set E. Note also, that for a hidden site s that only belongs to two check sets (such as trellis state spaces), we have $\mathcal{O}(\mu_{s,E}) = 0$, i.e., there is nothing to compute since the outgoing cost functions from such sites are just copies of the incoming cost functions. Also, for a “leaf” site s (i.e., a site that belongs to exactly one check set), there is nothing to compute and we have $\mu_{s,E} = \gamma_s$. (In fact, if a leaf site was *hidden*, we would get a negative number in (A.14). This may be attributed to the fact that $\mu_{s,E}$ is the all-zero function in that case, and the computations regarding the adjacent check can be simplified by simply discarding $\mu_{s,E}$. However, the “negative complexity” obtained from (A.14) need not necessarily “balance” this reduction in complexity. We will assume that there are no hidden leaf sites.)

The complexity of computing a final site cost function μ_s, $s \in N$, is

$$\mathcal{O}(\mu_s) = |A_s||s| \text{ if } s \text{ is visible.} \tag{A.15}$$

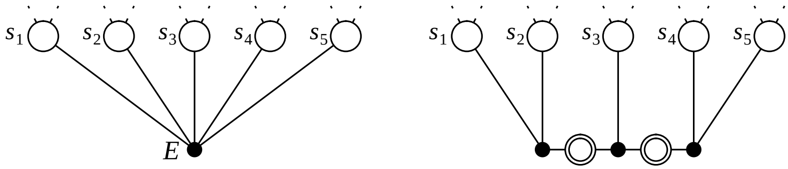


Figure A.4 *A refinement of a check E (left) into a small cycle-free check structure (right).*

We now turn to the complexity of updating the check-to-site cost function $\mu_{E,s}$. We may divide the computation into two steps. At the first step we loop through all local configurations and compute their cost sum; this takes $|B_E|(|E|-2)$ operations (if there is no local check cost function γ_E, otherwise $|B_E|(|E|-1)$ are needed). At the second step, we consider, for each site value $a \in A_s$, all local configurations $x_E \in B_E$ that match with a, i.e., with $x_s = a$, and take the minimum of their costs. So, let $M_a \triangleq |\{x_E \in B_E : x_s = a\}|$ be the number of local configurations that match with a. Then, the number of operations in this second step to compute $\mu_{E,s}(a)$ is $M_a - 1$, and the total number of operations in the second step is $\sum_{a \in A_s}(M_a - 1) = |B_E| - |A_s|$. By adding the complexity of the two steps, we get

$$\mathcal{C}(\mu_{E,s}) = |B_E|(|E|-2) + |B_E| - |A_s| = |B_E|(|E|-1) - |A_s| . \tag{A.16}$$

For large $|B_E|$ and $|E|$, there are often more efficient ways of updating the check E than just looping through the local behavior. Such "optimizations" can often be described as replacing the check with a small cycle-free realization of B_E. For example, a parity check on k sites has a local behavior of size 2^{k-1}; if the corresponding check-to-site cost functions $\mu_{E,s}$ are computed in the "naive" way, by applying the updating formula (3.2) k times (one for each site of E), the total complexity of that check is (using (A.16)) $k(2^{k-1}(k-1)-2)$. By implementing the parity check as a small cycle-free system, a much lower complexity is obtained, at least for large values of k. A parity check of size k can be replaced by $k-2$ parity checks of size three, and $k-3$ intermediate (hidden) binary sites, as in Figure A.4. Using (A.16) again, the number of operations for each such small check is 18, so the total complexity of the refined check structure is $18(k-2)$ (the new hidden sites require no computation, and the other sites are excluded from the comparison).

Since the savings that can be obtained in this way depend heavily on B_E, we will not consider this issue further, but instead assume that the local behaviors are small, so that (A.16) is useful.

A.2.1 Updating All Intermediate Cost Functions

When using the min-sum (or sum-product) algorithm on a realization with cycles, all intermediate cost functions typically have to be computed for each iteration. Here we consider the number of operations needed for each such iteration.

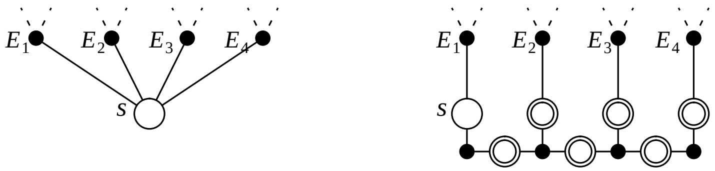


***Figure A.5** A site, s, which is connected to three checks (left), is **refined** into several sites, each connected to only two checks (right).*

Consider first the computation of the final cost function and all the site-to-check cost functions from a site s, i.e., μ_s and $\mu_{s,E}$ for all check sets E with $s \in E$. In a naive implementation, we might compute these cost functions independently, with the total number of operations $\mathcal{C}(\mu_s) + \sum_{E \in Q : s \in E} \mathcal{C}(\mu_{s,E}) = |s|^2|A_s|$. A more economical way is to save and reuse the temporary sums that occur in the computations, as follows. Let $E_1, \ldots, E_{|s|}$ be the check sets that s belongs to. Let $L_0 = \gamma_s$ and $L_i = L_{i-1} + \mu_{E_i,s}$ for $1 \le i < |s|$. Similarly, let $R_{|s|+1} = 0$ and $R_i = R_{i+1} + \mu_{E_i,s}$ for $1 < i \le |s|$. Then $\mu_{s,E_i} = L_{i-1} + R_{i+1}$. Computing L_i for all i requires $|A_s|(|s|-1)$ operations, while R_i (for all i) requires $|A_s|(|s|-2)$ operations. The final additions to get the cost functions μ_{s,E_i} require $|A_s|(|s|-2)$ operations (not counting addition with a zero constant) The total number of operations $\mathcal{C}_s$ involved in computing the site-to-check cost functions from the site s in this way is thus

$$\mathcal{C}_s = |A_s|(3|s|-5) \text{ if } s \text{ is visible.} \tag{A.17}$$

For hidden sites, the number of additions is one less, as usual. The computation of the final cost function μ_s requires only a single addition (for each element of the site alphabet), since $\mu_s = L_{|s|-1} + \mu_{E_{|s|},s}$. Also, it is not necessary that all outgoing cost functions of the site s are computed at the same time for this updating scheme to work.

In fact, this "optimized" site updating scheme can be obtained by replacing the site with a small cycle-free realization of it (just as with checks), consisting of several sites that are copies of the original one. Each site with degree larger than two is replaced by a small check structure, as illustrated in Figure A.5. The refined check structure consists of several new sites with the same alphabet as the original site, only one of which is visible, and each is connected to only two check sets. The new checks that are introduced are "repetition codes", i.e., they force their sites to have the same value. In other words, the new hidden sites are "copies" of the original site s.

Consider a realization (N, W, B) of the output code B_V (where $V \subseteq N$ are the visible sites) and a corresponding check structure Q. The number of binary operations of a complete update of all cost functions can be expressed using (A.17) and (A.16). Updating all site-to-check cost functions thus amounts to

$$\sum_{s \in V} |A_s|(3|s|-5) + \sum_{s \in N \setminus V} |A_s|(3|s|-6) \tag{A.18}$$

binary operations, while updating all check-to-site functions requires

$$\sum_{E \in Q} \sum_{s \in E} [|B_E|(|E|-1) - |A_s|] = \sum_{E \in Q} |B_E|(|E|^2 - |E|) - \sum_{s \in N} |A_s||s| \tag{A.19}$$

binary operations. The total number of binary operations of a complete update (excluding the final cost functions) is thus

$$\sum_{E \in Q} |B_E|(|E|^2 - |E|) + \sum_{s \in V} |A_s|(2|s|-5) + \sum_{s \in N \setminus V} |A_s|(2|s|-6) \ . \tag{A.20}$$

Computing the final cost functions requires only $|A_s|$ additions for each site s for which the final cost function is desired.

A.2.2 The Min-Sum Algorithm on Cycle-Free Realizations

In a cycle-free code realization, it is natural to use a smart updating order as discussed in Section 3.3. Such an updating order consists of two updating phases. In the first phase, all cost functions pointing “towards” a given site *r*, called the “root site”, are computed, as well as the final cost function at *r*. In the second phase, the remaining intermediate cost functions, which point “outwards” from *r* are computed, together with the final cost functions. With the min-sum algorithm, the second updating phase may be replaced by a simple “backtracking” procedure (as in the Viterbi algorithm), if the goal is just to find the lowest-cost valid configuration. We now consider only the first phase.

For each site *s* except *r*, we need to compute $\mu_{s,\,E(s)}$ where $E(s)$ is the check set containing *s* which is “closest” to the root site *r*. For each check set *E*, we need to compute $\mu_{E,\,s(E)}$, where $s(E)$ is the site of *E* which is “closest” to *r*. Finally, we need to compute μ_r. Thus, the complexity $\mathcal{C}_1$ of the first phase may be expressed as

$$\mathcal{C}_1 = \sum_{s \in N : s \neq r} \mathcal{C}(\mu_{s,\,E(s)}) + \sum_{E \in Q} \mathcal{C}(\mu_{E,\,s(E)}) + \mathcal{C}(\mu_r). \tag{A.21}$$

Now, let $(L, I, H, \{r\})$ be a partition of the site set *N* such that *L* are the leaf sites, *I* are the visible interior sites except for the root site *r*, and *H* are the hidden sites except for *r*. Then we can expand (A.21) as

$$\mathcal{C}_1 = \sum_{s \in I} |A_s|(|s|-1) + \sum_{s \in H} |A_s|(|s|-2) + \sum_{E \in Q} [|B_E|(|E|-1) - |A_{s(E)}|] + |r||A_r| \tag{A.22}$$

$$= \sum_{s \in I} |A_s||s| + \sum_{s \in H} |A_s|(|s|-1) + \sum_{E \in Q} |B_E|(|E|-1) \ , \tag{A.23}$$

where the last equality follows from $\sum_{E \in Q} |A_{s(E)}| = \sum_{s \in V \cup H} |A_s| + |r||A_r|$, which holds because $\{s(E) : E \in Q\}$ contains all interior sites $s \in I \cup H$ exactly once, except for the root site r which occurs $|r|$ times.

As mentioned, with the min-sum algorithm it often suffices to find the best configuration, and it only remains to find the optimum value at r and then trace stored pointers outwards through the Tanner graph. The complexity of finding the smallest of $|A_r|$ values is $|A_r| - 1$, so the overall complexity in this case is $\mathcal{C}_1 + |A_r| - 1$ plus the complexity of following the pointers.

A.3 Derivation for Section 3.4

For the updating of $\mu_{E,s}$, consider the site-to-check costs $\mu_{s',E}$, $s' \neq s$, and let $S_{s'}$ and $M_{s'}$ denote their sign and magnitude, so that $\mu_{s',E} = S_{s'} M_{s'}$. With these definitions, the updating rule can be written in the simple form

$$\mu_{E,s} = \left(\prod_{s'} S_{s'} \right) \min_{s'} M_{s'} , \tag{A.24}$$

where s' runs through all sites in E except that $s' \neq s$. To see that this is in fact the case, we first define the function

$$c : B_E \rightarrow \mathcal{R} : c(x_E) = \sum_{s' \in \text{support}(x_E) : s' \neq s} \mu_{s',E}, \tag{A.25}$$

i.e., $c(x_E)$ is the sum of the site-to-check costs $\mu_{s',E}$ over the “ones” in x_E (except for the site s). We also define the sets $B_E^{(0)} \triangleq \{x_E \in B_E : x_s = 0\}$ and $B_E^{(1)} \triangleq \{x_E \in B_E : x_s = 1\}$. We can then write the cost of “0” as

$$\mu_{E,s}(0) = \min_{x_E \in B_E^{(0)}} \left\lfloor \sum_{s' \in E : s' \neq s} \mu_{s',E}(x_{s'}) \right\rfloor \tag{A.26}$$

$$= \sum_{s' \in E : s' \neq s} \mu_{s',E}(0) + \min_{x_E \in B_E^{(0)}} c(x_E) , \tag{A.27}$$

and, similarly, the cost of “1” as

$$\mu_{E,s}(1) = \sum_{s' \in E : s' \neq s} \mu_{s',E}(0) + \min_{x_E \in B_E^{(1)}} c(x_E) , \tag{A.28}$$

so that the cost difference $\mu_{E,s} \triangleq \mu_{E,s}(1) - \mu_{E,s}(0)$ is

$$\mu_{E,s} = \min_{x_E \in B_E^{(1)}} c(x_E) - \min_{x_E \in B_E^{(0)}} c(x_E) . \tag{A.29}$$

Consider a local configuration $\hat{x}_E \in B_E$ that minimizes $c(\cdot)$ among all local configurations in B_E. Let $a = (\hat{x}_E)_s$ be the value at the site s in this minimizing configuration; clearly, $\hat{x}_E \in B_E^{(a)}$. We will now try to find a local configuration $\tilde{x}_E \in B_E^{(b)}$, for $b = 1 - a$, which minimizes $c(\cdot)$ in $B_E^{(b)}$. Then we will have, from (A.29), that $|\mu_{E,s}| = c(\tilde{x}_E) - c(\hat{x}_E)$.

Assume, for a moment, that $\mu_{s',E} = 0$ for some $s' \in E$, $s' \neq s$. Then there is a $\tilde{x}_E \in B_E^{(b)}$ with $c(\tilde{x}_E) = c(\hat{x}_E)$ (the one that differs from $\hat{x}_E$ only at s and s'), and we can conclude that $\mu_{E,s} = 0$. So, in the following, we assume that $\mu_{s',E} \neq 0$ for all s'.

Since the bits of $\hat{x}_E$ are chosen freely (except at s) to minimize $c(\cdot)$, it is clear that $\hat{x}_{s'} = 1$ exactly for $\mu_{s',E} < 0$. This means that for any local configuration x_E we can write

$$c(x_E) = c(\hat{x}_E) + \sum_{s' \in \operatorname{support}(x_E - \hat{x}_E) : s' \neq s} M_{s'}, \tag{A.30}$$

and in particular, for $\tilde{x}_E \in B_E^{(b)}$ where $\tilde{x}_E$ and $\hat{x}_E$ differ in at least one bit (except s), it is best to choose s' to minimize $M_{s'}$. Thus, $|\mu_{E,s}| = \min_{s'} M_{s'}$. It remains to determine the sign of $\mu_{E,s}$. From (A.29), we see that $\mu_{E,s} = c(\tilde{x}_E) - c(\hat{x}_E) > 0$ if $\hat{x}_E \in B_E^{(0)}$, otherwise $\mu_{E,s} < 0$. But $\hat{x}_E \in B_E^{(0)}$ precisely if there is an even number of ones in $\hat{x}_E$, i.e., if $\mu_{s',E} < 0$ for an even number of sites s'; it follows that $\operatorname{sign}(\mu_{E,s}) = \prod_{s'} S_{s'}$.

A.4 Derivation used in Section 5.4

Let X and Y be i.i.d. random variables with density $f_X(x) = f_Y(x)$ and distribution functions $F_X(x) = F_Y(x) = P(X \leq x)$. Let $\bar{F}_X(x) = \bar{F}_Y(x) = 1 - F_X(x)$. Let Z be a random variable defined as a function of X and Y as

$$Z = \operatorname{sign} XY \min(|X|, |Y|). \tag{A.31}$$

Then we have the density of Z as

$$f_Z(z) = \begin{cases} 2 f_X(z)\bar{F}_X(z) + 2 f_X(-z)F_X(-z), & \text{if } z \geq 0 \\ 2 f_X(z)\bar{F}_X(-z) + 2 f_X(-z)F_X(z), & \text{if } z < 0 \end{cases} \tag{A.32}$$

$$= 2 f_X(z)\bar{F}_X(|z|) + 2 f_X(-z)F_X(-|z|). \tag{A.33}$$

The expectation of Z is then

$$E[Z] = \int_{-\infty}^{\infty} z f_Z(z)dz = 2\int_{-\infty}^{\infty} z f_X(z)\bar{F}_X(|z|)dz + 2\int_{-\infty}^{\infty} z f_X(-z)\bar{F}_X(-|z|)dz \qquad \text{(A.34)}$$

(substituting z for -z in second the term of the integral)

$$= 2\int_{-\infty}^{\infty} z f_X(z)(1 - F_X(|z|))dz - 2\int_{-\infty}^{\infty} z f_X(z)F_X(-|z|)dz \qquad \text{(A.35)}$$

$$= 2\int_{-\infty}^{\infty} z f_X(z)dz - 2\int_{-\infty}^{\infty} z f_X(z)(F_X(|z|) + F_X(-|z|))dz \qquad \text{(A.36)}$$

(utilizing $F_X(|z|) + F_X(-|z|) = F_X(z) + F_X(-z)$)

$$2E[X] - 2\underbrace{\int_{-\infty}^{\infty} z f_X(z)F_X(z)dz}_{A} - 2\underbrace{\int_{-\infty}^{\infty} z f_X(z)F_X(-z)dz}_{B}. \qquad \text{(A.37)}$$

We now assume that X and Y are gaussian with $E[X] = \mu$ and $V[X] = \sigma^2$, i.e.

$$f_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(x-\mu)^2}{2\sigma^2}} \qquad \text{(A.38)}$$

$$F_X(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(t-\mu)^2}{2\sigma^2}}dt \triangleq \Phi\left(\frac{t-\mu}{\sigma}\right). \qquad \text{(A.39)}$$

Inserting this into (A.37), and performing variable substitutions of the form $z = u + v$, $t = u - v$, we get (with the aid of Maple)

$$A = \frac{\mu}{2} + \frac{\sigma}{2\sqrt{\pi}}, \qquad \text{(A.40)}$$

and

$$B = \mu Q(\sqrt{2}\frac{\mu}{\sigma}) - \frac{e^{-\frac{\mu^2}{\sigma^2}}\sigma}{2\sqrt{\pi}}. \qquad \text{(A.41)}$$

where $Q(\cdot)$ is the standard tail function of the normal distribution. Inserting A and B in (A.37) gives

$$E[Z] = 2\mu - 2\left(\frac{\mu}{2} + \frac{\sigma}{2\sqrt{\pi}}\right) - 2\left(\mu Q\left(\sqrt{2}\frac{\mu}{\sigma}\right) - \frac{e^{-\frac{\mu^2}{\sigma^2}}\sigma}{2\sqrt{\pi}}\right) \tag{A.42}$$

$$= \mu\,\mathrm{erf}\left(\frac{\mu}{\sigma}\right) - \frac{\sigma}{\sqrt{\pi}} + \frac{e^{-\frac{\mu^2}{\sigma^2}}}{\sqrt{\pi}}. \tag{A.43}$$

We now go on to compute the second moment of Z.

$$E[Z^2] = \int_{-\infty}^{\infty} z^2 f_Z(z)dz = 2\int_{-\infty}^{\infty} z^2 f_X(z)\bar{F}_X(|z|)dz + 2\int_{-\infty}^{\infty} z^2 f_X(-z)F_X(-|z|)dz \tag{A.44}$$

(again, we substitute $-z$ for z in the second term)

$$= 2\int_{-\infty}^{\infty} z^2 f_X(z)(1 - F_X(|z|))dz + 2\int_{-\infty}^{\infty} z^2 f_X(z)F_X(-|z|)dz \tag{A.45}$$

$$= 2E[X^2] + 2\int_{-\infty}^{\infty} z^2 f_X(z)F_X(-|z|)dz - 2\int_{-\infty}^{\infty} z^2 f_X(z)F_X(|z|)dz \tag{A.46}$$

$$= 2E[X^2] + 2\int_{-\infty}^{\infty} z^2 f_X(z)\int_{-\infty}^{-|z|} f_X(t)dtdz - 2\int_{-\infty}^{\infty} z^2 f_X(z)\int_{-\infty}^{|z|} f_X(t)dtdz \tag{A.47}$$

$$= 2E[X^2] - 2\left\{\underbrace{\int_0^{\infty}\int_{-z}^{z} z^2 f_X(z) f_X(t)dtdz}_{A} + \underbrace{\int_{-\infty}^{0}\int_{z}^{-z} z^2 f_X(z) f_X(t)dtdz}_{B}\right\} \tag{A.48}$$

With similar substitutions as in the case of the first moment, we get

$$A = 2\int_0^{\infty}\int_0^{\infty}(u + v)^2 f_X(u + v) f_X(u - v)dvdu \tag{A.49}$$

$$= \frac{1}{4\pi}(\mu\sigma\sqrt{\pi} + 2\sigma^2)e^{-\frac{\mu^2}{\sigma^2}} + \frac{1}{2}\left(\mu^2 + \frac{2\mu\sigma}{\sqrt{\pi}} + \sigma^2\right)\left(1 - Q(\frac{\mu\sqrt{2}}{\sigma})\right) , \tag{A.50}$$

and

$$B = 2\int_0^\infty\int_0^\infty (-(u+v))^2 f_X(-(u+v)) f_X(u-v) dvdu \tag{A.51}$$

$$= \frac{1}{2}\left(\mu^2 + \sigma^2 - \frac{2\mu\sigma}{\sqrt{\pi}}\right)Q(\frac{\mu\sqrt{2}}{\sigma}) + \left(\frac{\sigma^2}{2\pi} - \frac{\mu\sigma}{4\sqrt{\pi}}\right)e^{-\frac{\mu^2}{\sigma^2}} . \tag{A.52}$$

Inserting in (A.48), we obtain

$$E[Z^2] = 2(\mu^2 + \sigma^2) - 2A - 2B \tag{A.53}$$

$$= \mu^2 + \sigma^2 - \frac{2\sigma^2}{\pi}e^{-\frac{\mu^2}{\sigma^2}} - \frac{2\mu\sigma}{\sqrt{\pi}}\operatorname{erf}(\frac{\mu}{\sigma}) . \tag{A.54}$$

A.5 Proofs for Section 6.1

A.5.1 Proof of Theorem 6.1

To prove the theorem, we will need the following lemma:

> **Lemma A.1** On a finite Tanner graph, the length of any irreducible walk is upper bounded by the number of edges in the graph.
>
> *Proof*. Let $s_1 \ldots s_n$ be a walk where n is larger than the number of edges in the Tanner graph. Then there is a corresponding sequence $t_1, F_1, t_2, F_2, \ldots, t_n, F_n$ of alternating sites and check sets on the tree. Any combination (t, F) of a site and a check set (with $t \in F$) corresponds to an edge on the Tanner graph; since n is larger than the number of edges, it follows that there are integers i and $j > i$ such that (t_i, F_i) and (t_j, F_j) correspond to the same edge. But then $t_i, F_i, \ldots, t_{j-1}, F_{j-1}$ corresponds to a closed walk, and the remaining sequence $t_1, F_1, \ldots, t_{i-1}, F_{i-1}, t_j, F_j, \ldots, t_n, F_n$ corresponds to a walk, thus the walk $s_1 \ldots s_n$ is reducible. ❑

Proof of Theorem 6.1. Let $S = s_1, s_2, \ldots, s_k$ be the walk corresponding to a deviation e (the root site is in the middle of the walk). We now factor this walk into a collection $S_1, \ldots, S_m$ of irreducible closed walks and a remaining irreducible walk S_0. The cost of e may be expressed as

$$\mathcal{G}(e) = \sum_{s=1}^{k} \gamma_{s_i} = \sum_{s \in S_0} \gamma_s + \sum_{i=1}^{m} \sum_{s \in S_i} \gamma_s = G(S_0) + \sum_{i=1}^{m} G(S_i), \tag{A.55}$$

where the cost $G(S)$ of any of the “factor” irreducible walks is finite (since their length is finite). Let $\hat{G}$ be the smallest cost of any irreducible closed walk. Then

$$\mathcal{G}(e) \geq G(S_0) + m\hat{G} \tag{A.56}$$

which tends to infinity if $\hat{G} > 0$. Thus, $\mathcal{G}(e) \leq 0$ when the number of iterations tends to infinity only if some irreducible closed walks have negative (or zero) cost. ❑

A.5.2 Proof of Theorem 6.2

Let $\mu_{E,s}^{(i)}$ and $\mu_{s,E}^{(i)}$ be the intermediate cost functions after iteration i, and assume that $\mu_{E,s}^{(i)}(1) - \mu_{E,s}^{(i)}(0) \to \infty$ as $i \to \infty$, for all check sets E and sites $s \in E$. Then, for any site $s \in N$, there exists an integer l_s such that both

$$\mu_{E,s}^{(i)}(0) < \mu_{E,s}^{(i)}(1) \tag{A.57}$$

and

$$\mu_{s,E}^{(i)}(0) \leq \mu_{s,E}^{(i)}(1) \tag{A.58}$$

for all $i \geq l_s$. Let l_0 be the maximum over these l_s, $s \in N$.

We now fix a check set E, a site $s \in E$, and some $l > l_0$. Recall that $\mu_{E,s}^{(i)}(0)$ is the lowest cost on the subtree emanating from s through E of all valid configurations x with $x_s = 0$. Let $\hat{x}$ be such a lowest-cost configuration. But (A.57) and (A.58) imply that $\hat{x}$ is all-zero from the root site on to depth $l - l_0$. We record this as a lemma:

For any fixed check set E, site $s \in E$, and any $l > l_0$, the tree configuration $\hat{x}$ with lowest cost on the subtree through E is zero from the root on to depth $l - l_0$. ❑

Continuing with the proof of Theorem 6.2 we now assume, contrary to the claim of the theorem, that there exists an irreducible closed walk $S \subseteq N$ on Q such that $G(S) \leq 0$. We will derive a contradiction, which proves the claim. First, note that there exists a real number τ such that

$$\sum_{i=1}^{k} \gamma_{s_i} < \tau \tag{A.59}$$

for any walk $s_1, \ldots, s_k$ of length at most $l_0 + |S|$. We now choose a site s in S. Let E be the check set "between" s and its successor in S. For some fixed $l > l_0$, let $\hat{x}$ be the lowest-cost configuration with $\hat{x}_s = 0$, as discussed above. We now travel from the root to one of the leaves, along the walk $\hat{S}$ that corresponds to circulating in S; along this path, we invert all bits of $\hat{x}$, which results in a valid tree configuration $x^{(1)}$ with root value one. This new configuration cannot have a cost lower than $\mu_{E,s}^{(l)}(1)$, and thus

$$\mu_{E,s}^{(l)}(1) - \mu_{E,s}^{(l)}(0) \leq \sum_{s' \in \hat{S}} \gamma_{s'}(x_{s'}^{(1)}) - \gamma_{s'}(\hat{x}_{s'}) \tag{A.60}$$

But the above lemma implies that $\hat{x}$ is zero from the root on to depth $l - l_0$. By using (A.59) and the fact that $G(S) \leq 0$, we get

$$\mu_{E,s}^{(l)}(1) - \mu_{E,s}^{(l)}(0) \leq \tau . \tag{A.61}$$

Since τ is independent of l, this contradicts our assumption that $\mu_{E,s}^{(l)}(1) - \mu_{E,s}^{(l)}(0) \to \infty$, and the theorem is proved. ❑

A.5.3 Proof of Lemma 6.3

Let $s_1, \ldots, s_n$ be an irreducible closed walk, and let the corresponding alternating site/check set sequence be $s_1, E_1, \ldots, s_n, E_n$. Note that, for irreducible walks, $s_i = s_j$ implies $E_i = E_{j-1}$, since otherwise $s_i, \ldots, s_{j-1}$ would be a closed walk and the remaining sites $s_1, \ldots, s_{i-1}, s_j, \ldots, s_n$ would also be a walk (the last fact follows since a site belongs to exactly two check sets, and thus $E_i \neq E_{j-1}$ and $s_i = s_j$ implies $E_{i-1} \neq E_j$).

Assume, contrary to the claim, that $s_i = s_j = s_k$ with $i < j < k$. Then we have $E_i = E_{j-1}$ and $E_j = E_{k-1}$, and thus we have $E_i \neq E_{k-1}$ (since $E_{j-1} \neq E_j$). But then $s_i, \ldots, s_{k-1}$ is a closed walk that can be removed from S, leaving the walk $s_1, \ldots, s_{i-1}, s_k, \ldots, s_n$, which is impossible since we assumed S to be irreducible. ❑

A.5.4 Proof of Lemma 6.4

Let $S = s_1, \ldots, s_n$ be an irreducible closed walk with $s_i = s_j$ for $i < j$ and assume that no site occurs twice between i and j, i.e., there are no p, q with $s_p = s_q$ and $i < p < q < j$ (we can do this with full generality because, if there were such p and q, then we would simply take $i = p$ and $j = q$ and start over). Then $s_{i+1}, s_{i+2}, \ldots, s_{j-2}, s_{j-1}$ is a cycle, because s_i $(= s_j)$, s_{i+1} and s_{j-1} are all in the same check set (see the proof for Lemma 6.3).

By applying the same reasoning to the site sequence $s_j, \ldots, s_n, s_1, \ldots, s_i$, we see that there is a cycle there too. ❑

A.6 Proof of Theorem 7.2

For disjoint site subsets I and J, the notation $[x_I, y_J]$ will be used for the *concatenation* of x_I and y_J, i.e., $z = [x_I, y_J] \in W_{I \cup J}$ with $z_I = x_I$ and $z_J = y_J$. To prove Theorem 7.2, we need the following lemma:

Lemma A.2 Let (N, W, B) be a linear system and let (I, J) be a partition of the site set N. Then, for arbitrary valid configurations $x \in B$ and $y \in B$, the “mix” configuration $[x_I, y_J]$ is valid (i.e., in B) if and only if $\sigma_{I,J}(x) = \sigma_{I,J}(y)$, i.e., if and only if $y - x \in \tilde{B}_I + \tilde{B}_J$.

Proof. $[x_I, y_J] \in B \Leftrightarrow [0_I, (y-x)_J] \in B \Leftrightarrow (y-x)_J \in \tilde{B}_J$. Similarly, $[x_I, y_J] \in B \Leftrightarrow (y-x)_I \in \tilde{B}_I$. Together, this gives $[x_I, y_J] \in B \Leftrightarrow y - x \in \tilde{B}_I + \tilde{B}_J$. ❑

Proof of Theorem 7.2. We first claim that, for any valid configurations x and y such that $x_K = y_K$, we have $\sigma_{I',J'}(x') = \sigma_{I',J'}(y')$, where $x' \triangleq x_{I' \cup J'}$ and $y' \triangleq y_{I' \cup J'}$. Indeed, from Lemma A.2, we have $[x_K, x_I, y_J] \in B$ and thus $[x'_{I'}, y'_{J'}] \in B_{I' \cup J'}$, which implies $\sigma_{I',J'}(x') = \sigma_{I',J'}(y')$ by Lemma A.2 (applied to $B_{I' \cup J'}$).

The mapping $\varphi : B_K \to S_{I',J'}(B_{I' \cup J'}) : x_K \mapsto \sigma_{I',J'}(x_{I' \cup J'})$ (for arbitrary $x \in B$) is thus well defined. It remains to verify that φ is linear, i.e., that $\varphi(\alpha x_K + \beta y_K) = \alpha\varphi(x_K) + \beta\varphi(y_K)$ for any $x, y \in B$ and any scalars α, β. This follows from the following derivation.

$$\varphi(\alpha x_K + \beta y_K) = \sigma_{I',J'}((\alpha x_K + \beta y_K)_{I' \cup J'}) \tag{A.62}$$

$$= (\alpha\sigma_{I',J'}(x_{I' \cup J'}) + \beta\sigma_{I',J'}(y_{I' \cup J'})) \tag{A.63}$$

$$= (\alpha\varphi(x_K) + \beta\varphi(y_K)) \ . \tag{A.64}$$

References

[1] T. M. Cover and J. A. Thomas, *Elements of Information Theory,* Wiley & Sons, 1991.

[2] G. D. Forney, Jr., “The Viterbi algorithm”, *Proc. IEEE,* vol. 61, pp. 268–278, March 1973.

[3] D. J. Muder, “Minimal trellises for block codes”, *IEEE Trans. Inform. Theory,* vol. IT–34, pp. 1049–1053, Sept. 1988.

[4] G. Solomon and H. C. A. van Tilborg, “A connection between block and convolutional codes”, *SIAM J. of Appl. Math,* vol. 37, no. 2, Oct. 1979.

[5] R. G. Gallager, “Low-density parity-check codes”, *IRE Trans. Inform. Theory,* vol. 8, pp. 21–28, Jan. 1962.

[6] R. M. Tanner, “A recursive approach to low complexity codes”, *IEEE Trans. Inform. Theory,* vol. IT-27, pp. 533–547, Sept. 1981.

[7] C. Berrou, A. Glavieux, and P. Thitimajshima, “Near Shannon limit error-correcting coding and decoding: Turbo codes (1)”, *Proc. ICC’93,* Geneva, Switzerland, 1993, pp. 1064–1070.

[8] S. Benedetto and G. Montorsi, “Unveiling turbo codes: some results on parallel concatenated coding schemes”, to appear in *IEEE Trans. Inform. Theory.*

[9] N. Wiberg, H.-A. Loeliger, R. Kötter, “Codes and Iterative Decoding on General Graphs”, *European Trans. on Telecomm.,* vol. 6, no. 5, pp. 513–526, Sept. 1995.

[10] J. C. Willems, “Models for dynamics”, in *Dynamics Reported*, vol. 2, U. Kirchgraber and H. O. Walther, eds., Wiley and Teubner, 1989, pp. 171–269.

[11] H.-A. Loeliger, G. D. Forney, T. Mittelholzer and M. D. Trott, “Minimality and observability of group systems”, *Linear Algebra & Appl.*, vol. 205–206, pp. 937–963, July 1994.

[12] W. W. Peterson and E. J. Weldon, Jr., *Error-Correcting Codes*. 2nd ed., Cambridge, MA: MIT Press, 1972.

[13] N. Wiberg, *Approaches to Neural-Network Decoding of Error-Correcting Codes*, Linköping Studies in Science and Technology, Thesis No. 425, 1994.

[14] J. Hagenauer and L. Papke, “Decoding ‘turbo’-codes with the soft output Viterbi algorithm (SOVA)”, *Proc. 1994 IEEE Int. Symp. Inform. Th.*, Trondheim, Norway, June 27–July 1, 1994, p. 164.

[15] V. A. Zinoviev and V. V. Zyablov, “Decoding of non-linear generalized concatenated codes”, *Probl. Peredach. Inform.*, vol. 14, pp. 46–52, 1978.

[16] J. L. Massey, *Threshold decoding*, MIT Press, Cambridge, MA, 1963.

[17] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, “Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate”, *IEEE Trans. Inform. Theory*, vol. IT-20, pp. 284–287, March 1974.

[18] R. Kindermann and J. L. Snell, *Markov Random Fields and their Applications*. Providence: American Mathematical Society, 1980.

[19] J. Hagenauer, P. Hoeher, “A Viterbi algorithm with soft-decision outputs and its applications”, *Proc. GLOBECOM’89*, Dallas, Texas, pp. 47.1.1–47.1.7, Nov. 1989.

[20] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[21] R. Johannesson, K. S. Zigangirov, *Fundamentals of Convolutional Codes*, Unfinished working manuscript, Aug. 1992.

[22] J. E. M. Nilsson and R. Kötter, “Iterative decoding of product code constructions”, *Proc. ISITA94*, pp. 1059–1064, Sydney, November 1994.

[23] R. Kötter and J. E. M. Nilsson, “Interleaving strategies for product codes”, *Proc. EIDMA Winter Meeting on Coding Th., Inform. Th. and Cryptol.*, Veldhoven, Netherlands, Dec. 19–21, 1994, p. 34.

[24] A. E. Brouwer and A. M. Cohen, A. Neumaier, *Distance-Regular Graphs*, Springer, Berlin, 1989.

[25] P. K. Wong, “Cages—a survey”, *J. Graph. Th.*, vol. 6, pp. 1–22, 1982.

[26] G. D. Forney, Jr., “Density/length profiles and trellis complexity of linear block codes and lattices”, *Proc. IEEE Int. Symp. Inform. Th.*, Trondheim, Norway, June 27–July 1, 1994, p. 339.

[27] A. Lafourcade-Jumenbo and A. Vardy, “On Trellis Complexity of Block Codes: Optimal Sectionalizations”, *Proc. IEEE Int. Symp. Inform. Th.*, Whistler, B.C., Canada, Sept. 17–22, 1995, p. 123.

[28] A. Lafourcade-Jumenbo and A. Vardy, “On Trellis Complexity of Block Codes: Lower Bounds”, *Proc. IEEE Int. Symp. Inform. Th.*, Whistler, B.C., Canada, Sept. 17–22, 1995, p. 124.

[29] G. D. Forney, Jr., “Coset Codes—Part II: Binary Lattices and Related Codes”, *IEEE Trans. Inform. Theory*, vol. IT-34, pp. 1152–1187, Sept. 1974.

[30] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*, Elsevier Science Publishers B.V., 1977.

[31] L. N. Kanal and A. R. K. Sastry, “Models for Channels with Memory and Their Applications to Error Control”, *Proc. IEEE*, vol. 66, no. 7, July 1978.

Linköping Studies in Science and Technology
Dissertations, Information Theory

Viiveke Fåk: *Data Security by Application of Cryptographic Methods*.
Dissertation No. 25, 1978.

Robert Forchheimer: *Vectorial Delta Modulation for Picture Coding*.
Dissertation No. 40, 1979.

Rolf Blom: *Information Theoretic Analysis of Ciphers*.
Dissertation No. 41, 1979.

Hans Knutsson: *Filtering and Reconstruction in Image Processing*.
Dissertation No. 88, 1982.

Jan-Olof Brüer: *Information Security in a Computerized Office*.
Dissertation No. 100, 1983.

Torbjörn Kronander: *Some Aspects of Perception Based Image Coding*.
Dissertation No. 203, 1989.

Michael Bertilsson: *Linear Codes and Secret Sharing*.
Dissertation No. 299, 1993.

Jonas Wallberg: *Permutation Codes and Digital Magnetic Recording*.
Dissertation No. 306, 1993.

Haibo Li: *Low Bitrate Image Sequence Coding*.
Dissertation No. 318, 1993.

Magnus Nilsson: *Linear Block Codes over Rings for Phase Shift Keying*.
Dissertation No. 321. 1993.

Ragnar Nohre: *Some Topics in Descriptive Complexity.*
Dissertation No. 330, 1994.