Chapter 3
Channel Coding Basics

This chapter gives a short overview of the basic principle of linear block codes and possible decoding methods. As this treatise only deals with a subset of existing channel codes, only a short primer with small examples are given. For an in depth study many good books are available, like [1] and [2].

When speaking about channel coding we have to distinguish between the code definition itself and the associated decoding algorithm which has to calculate an estimation of the transmitted information based on the sequence of received symbols. The goal of a practical system is to establish a channel coding scheme which has the theoretical capability to approach the Shannon limit. However, just as important is an associated decoding method that can be implemented in hardware.

All described codes here in this chapter and in this manuscript are linear block codes. The basic terms and overview of channel codes are introduced in Sect. 3.1, linear block codes are presented in Sect. 3.2. The general decoding problem -trying to solve the maximum likelihood (ML) criterion—is addressed in Sect. 3.3. For the decoding algorithm we have to distinguish between a soft-input and hard-input information as explained in Sect. 2.2. Soft-input information refers to the LLR values for each received sample, provided by the demodulator. All decoding algorithms presented in this chapter require these LLR values as input, furthermore the decoding algorithms are derived in the logarithm domain which enables an efficient decoder hardware realizations.

Solving the ML criterion is an NP complete problem, thus in practical applications we can often only approximate the ML result by utilizing heuristics. These iterative heuristics are mandatory in the case of turbo and LDPC code decoding. When applying iterative decoding we have to solve the symbol-by-symbol maximum a posteriori (MAP) criterion which is explained in Sect. 3.3.3. One important class of channel codes is the class of convolutional codes. These are introduced in Sect. 3.4, an example of a decoding algorithm which approximates the MAP criterion is derived in Sect. 3.4.2.

F. Kienle, *Architectures for Baseband Signal Processing*,
DOI: 10.1007/978-1-4614-8030-3_3, © Springer Science+Business Media New York 2014

3.1 Overview Channel Coding

The goal of a channel code is to transmit information reliably via an unrealiable channel. To achieve this we map an information word $\boldsymbol{u}$ into a codeword $\boldsymbol{x}$. Throughout this manuscript we assume binary information for the elements $u_i, x_i \in [0, 1]$. Table 3.1 shows an exemplary binary code with $K = 3$ information bits which are mapped onto $N = 8$ distinct codewords. The code C is given by K basis vectors of length N. The cardinality of this code is $|C| = 2^3$. The process of mapping an information word to a code word is called encoding. A channel code defines only the mapping from $\boldsymbol{u}$ to $\boldsymbol{x}$, it does not define a possible decoding procedure.

One important measure for the quality of the code is its minimum (Hamming) distance. The Hamming distance is defined as the distance between two binary vectors, i.e.

$$d(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=0}^{N-1} |x_i - y_i|. \tag{3.1}$$

The minimum distance is an important measure for the quality of a code. The presented channel code in this example is a linear code, i.e., a channel code is linear if the addition of two codewords yields again a valid codeword. The minimum distance of a linear code is the minimum weight of a codeword:

$$d_{min} = \min_{\forall \boldsymbol{x} \in C} \sum_{i=0}^{N-1} x_i. \tag{3.2}$$

The code of Table 3.1 has a minimum Hamming distance of $d_{min} = 3$.

During transmitting of the codeword $\boldsymbol{x}$ errors may occur. Suppose we have transmitted one codeword of Table 3.1 and we receive a (hard) bit vector $\boldsymbol{y} = [111100]$. By using nearest neighbor decoding we can measure the distance of the received (corrupted) vector to the possibly sent codewords. The distances for all 8 possible codeword are: $d(\boldsymbol{x_0}, \boldsymbol{y}) = 4$, $d(\boldsymbol{x_1}, \boldsymbol{y}) = 1$, $d(\boldsymbol{x_2}, \boldsymbol{y}) = 3$, $d(\boldsymbol{x_3}, \boldsymbol{y}) = 2$, $d(\boldsymbol{x_4}, \boldsymbol{y}) = 2$, $d(\boldsymbol{x_5}, \boldsymbol{y}) = 3$, $d(\boldsymbol{x_6}, \boldsymbol{y}) = 5$, $d(\boldsymbol{x_7}, \boldsymbol{y}) = 4$. The codeword $\boldsymbol{x_1}$ has the smallest distance

Table 3.1 A binary code with $K = 3$ information bits and $N = 6$ codeword bits

Information word	Codeword
$\boldsymbol{u_0}$=[0 0 0]	$\boldsymbol{x_0}$=[0 0 0 0 0 0]
$\boldsymbol{u_1}$=[0 0 1]	$\boldsymbol{x_1}$=[1 1 0 1 0 0]
$\boldsymbol{u_2}$=[0 1 0]	$\boldsymbol{x_2}$=[0 1 1 0 1 0]
$\boldsymbol{u_3}$=[0 1 1]	$\boldsymbol{x_3}$=[1 0 1 1 1 0]
$\boldsymbol{u_4}$=[1 0 0]	$\boldsymbol{x_4}$=[1 1 1 0 0 1]
$\boldsymbol{u_5}$=[1 0 1]	$\boldsymbol{x_5}$=[0 0 1 1 0 1]
$\boldsymbol{u_6}$=[1 1 0]	$\boldsymbol{x_6}$=[1 0 0 0 1 1]
$\boldsymbol{u_7}$=[1 1 1]	$\boldsymbol{x_7}$=[0 1 0 1 1 1]

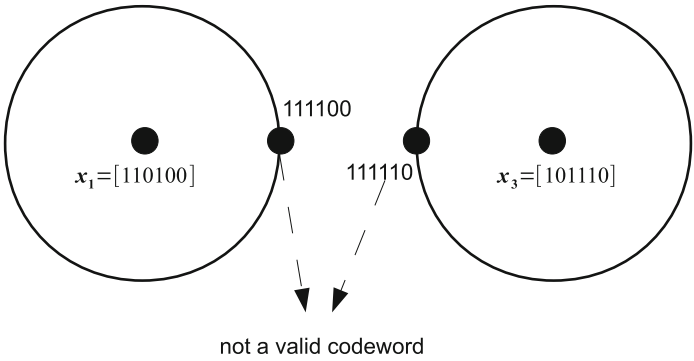


Fig. 3.1 Hamming distance between two valid codewords

to the received information $\boldsymbol{y}$ and is thus the codeword that was most likely sent by the transmitter.

The importance of a large minimum distance is shown in Fig. 3.1. It shows two codewords $\boldsymbol{x_1}$ and $\boldsymbol{x_3}$ with the Hamming distance of $d(\boldsymbol{x_1}, \boldsymbol{x_3}) = 3$. The non-overlapping spheres around the codewords indicate the decision region for a particular codeword, i.e. vectors inside this region can be clearly associated to a codeword. For a code with a minimum distance d_{min} we have a guaranteed error correction capability of:

$$t = \left\lfloor \frac{d_{min}}{2} \right\rfloor, \tag{3.3}$$

with t the number of errors occurred in the received vector. $\lfloor a \rfloor$ defines the next integer which is lower than or equivalent to a. The error detection capability of the code is $d_{min} - 1$, i.e., it can detect any invalid sequence as long as this sequence is not a codeword. Thus, the worst thing that can happen during transmission is an error sequence which results in another valid codeword, i.e. $\boldsymbol{y} = \boldsymbol{x_i} + \boldsymbol{e} = \boldsymbol{x_j}$, with $\boldsymbol{x_j} \in C$. The error sequence is represented here by an error vector $\boldsymbol{e}$ with $e_i \in \{0, 1\}$. An one entry indicates that the corresponding bit position is flipped. One goal when designing channel codes is to ensure a minimum distance as large as possible which will enable a high error correction capability.

3.2 Linear Block Codes

A binary linear block code with cardinality $|C| = 2^K$ and block length N is a K dimensional subspace of the vector space $\{0, 1\}^N$ defined over the binary field ($GF(2)$). It is possible to define a linear block code as well over larger field, e.g. extension fields $GF(2^m)$. However, here we only address binary codes. The operations with respect

Table 3.2 GF(2) addition (XOR)

+	0	1
0	0	1
1	1	1

Table 3.3 GF(2) multiplication (AND)

*	0	1
0	0	0
1	0	1

to $GF(2)$ are shown in Tables 3.2 and 3.3. Additions and multiplications result in simple *XOR* and *AND* operations.

The linear code C is given by K basis vectors of length N which are represented by a $K \times N$ matrix $\mathbf{G}$ (generator matrix). The encoding process can be described by a multiplication with the generator matrix G. Thus the encoder evaluates:

$$\mathbf{x} = \mathbf{uG} \tag{3.4}$$

The output of the encoder is the *codeword* $\boldsymbol{x}$. Most of the practically codes utilized in communication standards are linear codes. Wireless communications systems utilize packet based transmission techniques, thus linear block codes that are defined on vectors fits well to theses systems.

Equivalently, a code C can be described by a parity check matrix $\boldsymbol{H} \in \{0, 1\}^{M \times N}$ where $M = N - K$. We thus have $\boldsymbol{x} \in C$, i.e., $\boldsymbol{x}$ is a codeword, if

$$\boldsymbol{x}\boldsymbol{H}^T = \mathbf{0}. \tag{3.5}$$

The scalar product of each row $\boldsymbol{x} \cdot \boldsymbol{h_i}^T$ has to be zero. All operations are performed in the binary domain. We denote the i^{th} row and j^{th} column of $\boldsymbol{H}$ by $H_{i,.}, H_{.,j}$ respectively. $\boldsymbol{x}H_{i,.}^T = 0$ in $GF(2)$ is defined as the i^{th} parity check constraint.

The parity check constraints are used within decoding algorithms to check for a valid codeword. Assuming a vector $\boldsymbol{z} = \boldsymbol{x} + \boldsymbol{e}$ with remaining errors, the resulting codeword check will evaluated to

$$\boldsymbol{y}\boldsymbol{H}^T = \boldsymbol{x}\boldsymbol{H}^T + \boldsymbol{e}\boldsymbol{H}^T = \boldsymbol{s}. \tag{3.6}$$

$\boldsymbol{s}$ is denoted as syndrome and contains information about the current error vector. This property can be as well utilized within a decoding algorithm [2].

Since the codeword $\boldsymbol{x}$ may be derived via $\boldsymbol{x} = \boldsymbol{uG}$ one can see the important relation between generator matrix and parity check matrix which is:

$$\boldsymbol{G} \cdot \boldsymbol{H}^T = 0 \tag{3.7}$$

If the information word $\boldsymbol{u}$ is part of the codeword the code is called systematic, e.g.:

$$\boldsymbol{x} = \boldsymbol{uG} = [x_0 x_1 \ldots x_{N-1}] = [\boldsymbol{up}] = [\underbrace{u_0\, u_1 \ldots u_{K-1}}_{\text{systematic bits}}\ \underbrace{p_0 \ldots p_{M-1}}_{\text{parity bits}}] \tag{3.8}$$

The code is called non-systematic if the information vector is not part of the codeword. A linear systematic generator matrix is specified by $\boldsymbol{G} = [\boldsymbol{IP}]$, with $\boldsymbol{I}$ an identity matrix of size $K \times K$.

$$\boldsymbol{G} = \left[\begin{array}{cccc|ccc} 1 & 0 & \ldots & 0 & p_{0,0} & p_{0,1} & \cdots\ p_{0,M-1} \\ 0 & 1 & \ldots & 0 & p_{1,0} & p_{1,1} & \cdots\ p_{1,M-1} \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & \ldots & 1 & p_{K-1,0} & p_{K-1,1} & \cdots\ p_{K-1,M-1} \end{array}\right] \tag{3.9}$$

The number of parity bits M is defined by $N - K = M$. The matrix part which defines the parity constraints $\boldsymbol{P}$ is transpose within in the corresponding parity check matrix.

$$\boldsymbol{H} = [\boldsymbol{P^T I}] = \left[\begin{array}{ccc|ccc} p_{0,0} & p_{1,0} & \cdots\ p_{K-1,0} & 1 & 0 & \ldots\ 0 \\ p_{0,1} & p_{1,1} & \cdots\ p_{K-1,1} & 0 & 1 & \ldots\ 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ p_{0,M-1} & p_{1,M-1} & \cdots\ p_{K-1,M-1} & 0 & 0 & \ldots\ 1 \end{array}\right] \tag{3.10}$$

One of the first and best known linear block codes is the Hamming code.

$$\boldsymbol{G} = \begin{bmatrix} 1\,0\,0\,0\,1\,1\,0 \\ 0\,1\,0\,0\,0\,1\,1 \\ 0\,0\,1\,0\,1\,0\,1 \\ 0\,0\,0\,1\,1\,1\,1 \end{bmatrix} \quad \boldsymbol{H} = \begin{bmatrix} 1\,0\,1\,1\,1\,0\,0 \\ 1\,1\,0\,1\,0\,1\,0 \\ 0\,1\,1\,1\,0\,0\,1 \end{bmatrix}$$

The generator matrix enables a systematic encoding which can be seen by the identity matrix I at the left part of the matrix $G = [IP]$. With P the parity check equations for generating the corresponding parity bits. As mentioned mentioned the scalar product of each row $\boldsymbol{x} \cdot \boldsymbol{h_i}$ has to be zero, thus each code word has to fulfill the following parity check equations.

$$\begin{aligned} x_0 + x_2 + x_3 + x_4 &= 0 \\ x_0 + x_1 + x_3 + x_5 &= 0 \\ x_1 + x_2 + x_3 + x_6 &= 0 \end{aligned}$$

When ever all these parity check equations are fulfilled a valid codeword is found. This is utilized during the decoding process. This (7,4) Hamming code has a minimum distance of $d_{min} = 3$.

3.3 General Decoding Problem

The general decoding task is to detect or even correct errors which may have occurred during transmission. For deriving corresponding decoding criteria we initially ignore the result of the demodulator of Sect. 2.2 and derive the decoding task directly on the received vector $\boldsymbol{y}$. Later we will comment on the important separation of demodulator and decoding algorithm. We can distinguish two different decoding principles, which either optimizes results on the entire codeword or on each individual bit.

- Maximum likelihood criterion: solving the maximum likelihood criterion optimizes the so called codeword probability which means ML decoding picks a codeword $\hat{\boldsymbol{x}}$ which maximizes the condition probability:

$$\hat{\boldsymbol{x}} = \arg\max_{\boldsymbol{x} \in C} P\,(\boldsymbol{x} \quad \text{sent} | \boldsymbol{y} \quad \text{received}) \tag{3.11}$$

  The receiver has no knowledge about the true sent codeword $\boldsymbol{x}$. Thus the receiver algorithm for decoding may check all possible sent codewords $\boldsymbol{x} \in C$ and decides for the codeword $\hat{\boldsymbol{x}}$ which was most likely sent.
- Symbol-by-symbol maximum a posteriori criterion: solving the symbol-by-symbol MAP criterion optimizes the bit probability which means MAP decoding decides for a single bit x_i

$$\hat{x}_i = \arg\max P\,(x_i \quad \text{sent} | \boldsymbol{y} \quad \text{received}) \tag{3.12}$$

  The resulting bit estimations for the entire codeword $\hat{x}_i$ for $i \in \{1, \ldots, N\}$ will result in the optimum bit error rate. The result will not necessarily be a valid codeword. The MAP probability will be $\arg\max P\,(x_i | \boldsymbol{y})$ with $x_i \in \{0, 1\}$.

3.3.1 Maximum Likelihood (ML) Decoding

For many block based transmissions the frame error rate is of importance, thus when ever possible we should try to solve th ML criterion. In the following we will derive the ML criterion in a more evident form. Applying Bayes's rule to entire vectors results for the ML criterion in:

$$\hat{\boldsymbol{x}} = \arg\max_{\boldsymbol{x} \in C} \left(P\,(\boldsymbol{y} | \boldsymbol{x}) \cdot \frac{P(\boldsymbol{x})}{P(\boldsymbol{y})} \right) \tag{3.13}$$

$P(\boldsymbol{y})$ is the vector of channel related probability. The vector elements are constant for each received information, it will thus have no influence on the final decision. Furthermore we assume that each codeword was equally likely sent. With this assumption, the term $P(\boldsymbol{x})$ can be omitted as well. Under the assumption that each received sample is independent and with the transformation to the logarithm domain

the maximum likelihood (ML) criterion turns into

$$\hat{\boldsymbol{x}} = \arg\max_{\boldsymbol{x}\in C} P(\boldsymbol{y}|\boldsymbol{x}) \tag{3.14}$$

$$= \arg\max_{\boldsymbol{x}\in C} \left(\prod_{i=0}^{N-1} P(y_i|x_i)\right) \tag{3.15}$$

$$= \underset{\boldsymbol{x}}{\in} C \arg\min \left(-ln\prod_{i=0}^{N-1} P(y_i|x_i)\right) \tag{3.16}$$

$$= \arg\min_{\boldsymbol{x}\in C} \left(-\sum_{i=0}^{N-1} lnP(y_i|x_i)\right) \tag{3.17}$$

The vector problem can be decomposed in the product form. This is only possible if the received values are independent of each other. The next two steps are the transformation in the logarithm domain and the transformation in a minimization problem which can be simply done by -1 multiplication. Note, that $P(y_j|x_i)$ is exactly the result of our demodulator in the probability domain, attention here we assumed that for each received symbol one bit exists.

It is more convenient to express the ML criterion with log likelihood values. If we add the constant $\sum_{i=0}^{N-1} lnP(y_i|0)$, i.e. the weight of the zero codeword, the problem results in

$$\hat{\boldsymbol{x}} = \arg\min_{\boldsymbol{x}\in C} \left(\sum_{i=0}^{N-1} (lnP(y_i|x_i = 0) - lnP(y_i|x_i))\right) \tag{3.18}$$

When ever the codeword has a zero at the corresponding position $x_i = 0$ no weight will be added to the entire cost function. If the corresponding position equals $x_i = 1$ the already introduced LLR term results with $\lambda_i = ln\frac{P(y_i|x_i=0)}{P(y_i|x_i=1)}$. Now, the ML criterion can be expressed in a more elegant form with the scalar product of the LLR vector obtained of the demodulator and a possible codeword $\boldsymbol{x} \in C$. The maximum likelihood criterion is thus a minimum search via a linear cost function of the demodulator output and a possible code word

$$\hat{\boldsymbol{x}} = \arg\min_{\boldsymbol{x}\in C} \left(\sum_{i=0}^{N-1} \lambda_i x_i\right). \tag{3.19}$$

The maximum likelihood criterion derived here has the cost function of $\sum_{i=0}^{N-1} \lambda_i x_i$. Each possible codeword $\boldsymbol{x} \in C$, is associated with its cost or weight w_c, which can be expressed as the scalar product of $w_c =< \boldsymbol{\lambda}, \boldsymbol{x}_c >$, with c the index of one particular codeword. We decide in favor of the codeword with the minimum resulting weight. Each λ_i value represents the derived LLR result of Sect. 2.2 on bit level, defined in a general form $\lambda_i = ln\frac{P(y_j|x_i=0)}{P(y_j|x_i=1)}$. Depending on the channel statistics this calculation may differ, however the derived ML criterion is still valid if λ_i is represented on

bit level. A general framework to solve the ML criterion for linear block codes is presented in the next section.

3.3.2 ML Decoding by Solving IP Problems

The ML decoding problem can be formulated as an integer program (IP). In literature mainly two different possibilities can be found for the IP modeling [3, 4]. One formulation relies on the parity check equation, the other one relies on the syndrome and the error vector $\boldsymbol{e}$. Both formulations model the ML decoding problem exactly and are thus equivalent.

$$\min \sum_{i=0}^{N-1} \lambda_i x_i \tag{3.20}$$

$$\text{s.t.} \quad \boldsymbol{H}\boldsymbol{x} - 2\boldsymbol{z} = \boldsymbol{0}$$

$$\boldsymbol{x} \in \{0, 1\}^N$$

$$\boldsymbol{z} \in \mathbb{Z}^{N-K}, \; z_i \geq 0 \tag{3.21}$$

The variables $\boldsymbol{x}$ model bits of the codeword, while $\boldsymbol{z}$ is a vector of variables used by the IP formulation to account for the binary modulo 2 arithmetic. The cost function to be minimized is identical to that derived in previous section.

For the syndrome based formulation the cost function changes and is based on the error vector $\boldsymbol{e}$.

$$\min \sum_{i=0}^{N-1} |\lambda_i| e_i \tag{3.22}$$

$$\text{s.t.} \quad \boldsymbol{H}\boldsymbol{e} - 2\boldsymbol{z} = \boldsymbol{s}$$

$$\boldsymbol{e} \in \{0, 1\}^N$$

$$\boldsymbol{z} \in \mathbb{Z}^{N-K}, \; z_i \geq 0 \tag{3.23}$$

The constraints are based on the syndrome vector $\boldsymbol{s}$ while again the $\boldsymbol{z}$ variables ensure the $GF(2)$ operations.

Interestingly both formulations are efficient enough such that a general purpose IP solver can tackle the problem for codes of practical interest. At least for smaller block sizes we can perform Monte-Carlo simulations to obtain the FER performance. The IP formulation has to be solved many times, once for every simulated frame.

Since the formulation are only based on the parity check matrix all types of linear block codes can be simulated with this IP formulation. Furthermore, the IP can be enhanced to comprise as well higher oder modulation types. This was shown in [5].

The general framework of integer programming models is shortly introduced together with some solution strategies. Let $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{b} \in \mathbb{R}^m$, and $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ be given.

Let $\boldsymbol{x} \in \mathbb{R}^n$ denote the vector of variables. Integer programming is the mathematical discipline which deals with the optimization of a linear *objective function* $c^T x$ over the feasible set, i.e. the space of all candidate solutions. A general linear integer programming problem can be written as

$$\begin{aligned} \min \text{ or } \max \quad & \boldsymbol{c}^T \boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \in \mathbb{Z}^n. \end{aligned}$$

Without loss of generality, we may consider minimization problems only. In contrast to linear programming problems, solutions are required to be integral. General IP problems—as well as many special cases—are NP-hard. However, due to extensive studies of theoretical properties, the development of sophisticated methods, as well as increasing computational capabilities, IP proved to be very useful to model and solve many real-world optimization problems (e.g. production planning, scheduling, routing problems,…).

IP problems can be solved in a brute force way by explicitly enumerating all possible values of the variable vector $\boldsymbol{x}$ and choosing the one yielding the minimal objective function value. Though correct, this procedure is guaranteed to be exponential in the number of components of $\boldsymbol{x}$. To avoid excessive computational effort, a theory inducing more efficient algorithms was developed which relies on techniques like implicit enumeration, relaxation and bounds, and the usage of problem-specific structural properties. In the following, we will highlight some basic concepts. An in-depth exposition to this field can be found in [6].

Branch and bound is a wide-spread technique realizing the concept of divide-and-conquer in the context of IP: For each $i \in \{0, \ldots, n-1\}$ and $v \in \mathbb{Z}$, an optimal solution $\boldsymbol{x}^*$ either satisfies $x_i^* \leq v$ or $x_i^* \geq v + 1$.

Using these two constraints two sub problems can be created from the original problem formulation by adding either one or the other constraint to the original set of constraints. This can be seen as branches of a tree, while each branch posses a smaller feasible set. At least one branch results in the optimal solution $\boldsymbol{x}^*$. Iterative application of this branching step yields IP problems of manageable size. For each (sub)problem, primal and dual bounds are obtained by relaxation techniques and heuristics. They allow to prune branches of the search tree, thus reducing the search area (implicit enumeration). Branch and bound techniques are often used within algorithms for communications systems, e.g., the sphere decoding which is presented in Chap. 8.

For any IP problem a linear programming (LP) problem can be derived, called the LP relaxation. This can be done by taking the same objective function and same constraints but with the requirement that the integer variables are replaced by appropriate continuous constraints. Cutting plane algorithms rely on the idea of solving the IP problem as the LP problem

$$\min\{\boldsymbol{c}^T \boldsymbol{x} : \boldsymbol{x} \in \operatorname{conv}(P_I)\}.$$

However, the convex hull of the feasible set (P_I) of the IP problem is in general hard to compute explicitly. Therefore, approaches are developed which iteratively solve the LP relaxation of the IP and compute a cutting plane, i.e., a valid inequality separating the feasible set P_I from the optimal solution of the LP relaxation. These cuts are added to the formulation in all subsequent iteration. Important questions address the convergence of this procedure, as well as the generation of strong valid inequalities. In case of channel code decoding this technique is applied in [4].

A mixture of strategies such as branch-and-cut and the utilization of problem intrinsic structures might lead to enhanced solution strategies. Implementing an efficient IP problem solver is certainly a demanding task. For a special purpose solver, the problem has to be thoroughly understood, a suitable algorithm has to be chosen and efficiently realized. But there also exists a bandwidth of all-purpose solvers, both open source (like GLPK [7]) and commercial (e.g. CPLEX [8]), which may be sufficient to solve various different IP problems such as the ones mentioned for ML decoding. Several ML decoding results are presented within this treatise, all are based on solving the corresponding IP formulations as described in this section. In [9, 10] further IP formulations are presented, that cover different aspects of code analysis. All formulations are general and can therefore be applied to arbitrary linear block codes.

ML Decoding Examples

In the following two simple channel codes are introduced together with their ML decoding procedure. Since the code examples are very small it is possible to explicitly enumerate all solutions. For both examples we assume an input LLR vector $\boldsymbol{\lambda}$ of length $N = 4$. Of course, for decoding the receiver needs of course the knowledge of utilized encoding scheme. One example uses a so called repetition code for encoding, the second example a so called single parity check code.

Example: Repetition Code

A repetition code is a very simple code that simply repeats a single bit of information (K = 1) N times. The generator matrix and the parity check matrix of a repetition code for $N = 4$ is the following:

$$\boldsymbol{G} = [1111] \; \boldsymbol{H} = \begin{bmatrix} 1\,0\,0\,1 \\ 0\,1\,0\,1 \\ 0\,0\,1\,1 \end{bmatrix}$$

In this example one bit is repeated four times which results in a code rate of $R = 1/4$. For this type of a code it is obvious that only two possible codewords exist, since $K = 1$: either the all zero codeword or the all one codeword.

Imagine we receive the LLRs $\boldsymbol{\lambda} = \begin{bmatrix} 0.4 & 1.1 & -0.1 & -0.3 \end{bmatrix}$. The ML decoder determines, which codeword $\hat{\boldsymbol{x}}$ was the one that most likely led to the observation of $\boldsymbol{\lambda}$ solving:

$$\hat{\boldsymbol{x}} = \underset{\boldsymbol{x} \in C}{\arg\min} \left(\sum_{i=0}^{N-1} \lambda_i x_i \right).$$

As already mentioned the code space C for a repetition code has just two possible codewords. Evaluating the cost function for these two codewords results in:

$$C := \begin{cases} [0\ 0\ 0\ 0] \\ [1\ 1\ 1\ 1] \end{cases} \quad \Rightarrow \quad \sum_{i=0}^{N-1} \lambda_i x_i = \begin{cases} 0 \\ 1.1 \end{cases}$$

When looking at the result of the cost function it can be seen that $\sum_{i=0}^{3} \lambda_i x_i = 0$ is the result for the all zero codeword while for the all one codeword the weight is $\sum_{i=0}^{3} \lambda_i x_i = 1.1$. The ML decoding solution is thus $\hat{\boldsymbol{x}} = [0\ 0\ 0\ 0]$. The minimum distance of this code is $d_{min} = 4$, since all four bits are different. When looking at the input values of $\boldsymbol{\lambda}$ one can see that a decoding with just the sign information $sign(\boldsymbol{\lambda}) = \begin{bmatrix} 1.0 & 1.0 & -1.0 & -1.0 \end{bmatrix}$ would result in cost values which can not be distinguished. Thus we could only detect an error and not correct it. This is one intuitive example which shows that soft information input provides a better error correction capability than hard-input values.

Example: Single Parity Check Code

A second very simple code is the so called single parity check code with $K = 3$ information bits. The corresponding generator and parity check matrices for the resulting block length of N=4 are:

$$\boldsymbol{G} = \begin{bmatrix} 1\ 0\ 0\ 1 \\ 0\ 1\ 0\ 1 \\ 0\ 0\ 1\ 1 \end{bmatrix} \boldsymbol{H} = [1111]$$

Again the codeword is generated by the multiplication of the information vector with the generator matrix $\boldsymbol{x} = \boldsymbol{u}G$. The code space has now $2^K = 8$ possible codewords, since the information vector $\boldsymbol{u}$ has now three bits. The resulting eight codewords and thus the code space C is:

$$\boldsymbol{u} = \begin{cases} [0\,0\,0] \\ [0\,0\,1] \\ [0\,1\,0] \\ [0\,1\,1] \\ [1\,0\,0] \\ [1\,0\,1] \\ [1\,1\,0] \\ [1\,1\,1] \end{cases} \Rightarrow [u_0 u_1 u_2] \begin{bmatrix} 1\,0\,0\,1 \\ 0\,1\,0\,1 \\ 0\,0\,1\,1 \end{bmatrix} = \boldsymbol{x} \qquad C := \begin{cases} [0\,0\,0\,0] \\ [0\,0\,1\,1] \\ [0\,1\,0\,1] \\ [0\,1\,1\,0] \\ [1\,0\,0\,1] \\ [1\,0\,1\,0] \\ [1\,1\,0\,0] \\ [1\,1\,1\,1] \end{cases}$$

In the following we will calculate the ML estimation $\hat{\boldsymbol{x}}$ when receiving $\boldsymbol{\lambda} = \begin{bmatrix} 0.4 & -1.1 & 0.1 & 0.3 \end{bmatrix}$. The cost function for all possible codewords can be evaluated to:

$$C := \begin{cases} [0\,0\,0\,0] \\ [0\,0\,1\,1] \\ [0\,1\,0\,1] \\ [0\,1\,1\,0] \\ [1\,0\,0\,1] \\ [1\,0\,1\,0] \\ [1\,1\,0\,0] \\ [1\,1\,1\,1] \end{cases} \Rightarrow \sum_{i=0}^{N-1} \lambda_i x_i = \begin{cases} 0 \\ 0.4 \\ -0.8 \\ -1.0 \\ 0.7 \\ 0.5 \\ -0.7 \\ -0.3 \end{cases}$$

The smallest value is -1.0 and thus the codeword that most likely equals the originally transmitted codeword is $\hat{\boldsymbol{x}} = [0\,1\,1\,0]$.

Solving the maximum likelihood criterion is the goal of a channel decoder. However, maximum likelihood decoding is an NP-hard problem [11]. The presented examples solve the ML decoding problem by evaluating all possible codes in a brute force manner. In practical examples this is not a feasible solution since the cardinality of the code space is $|C| = 2^K$, with $K > 1000$ for many applications. This is why some channel codes, e.g. convolutional codes, have special properties which enable an efficient realization of an ML decoding algorithm. For example the well known Viterbi algorithm is one algorithm which utilizes a special structure of the code to solve the ML criterion—in this case the so called trellis structure. The trellis structure is an elegant way to represent all possible codewords generated by a convolutional code (CC), see Sect. 3.4. The Viterbi algorithm checks all possible realizations of the codeword $\boldsymbol{x}$ and decides in favor of that codeword with the minimum cost function $\sum_{i=0}^{N-1} \lambda_i x_i$.

3.3.3 Symbol-by-Symbol MAP Decoding

For modern codes (turbo codes, LDPC codes) the maximum likelihood decoding is too complex for practical applications. Thus, heuristics are used which tries to approximate the ML solution. The typical procedure is to divide the code C in smaller component codes

$$C = \{C_0 \cap C_1 \cap \ldots \cap C_j\} \tag{3.24}$$

Typically, the cardinality $|C_k|$ is identically for each sub-code, while a sub-code has a smaller cardinality as the original code $|C_k| \leq |C|$. Each sub-code is a linear code itself which can be defined by a parity check matrix $\boldsymbol{H}_{C_k}$. $\boldsymbol{H}_{C_k}$ is in turn a part of the original parity check matrix. As an example the Hamming code is decomposed in three sub-codes.

$$\boldsymbol{H} = \begin{bmatrix} 1\,0\,1\,1\,1\,0\,0 \\ 1\,1\,0\,1\,0\,1\,0 \\ 0\,1\,1\,1\,0\,0\,1 \end{bmatrix} \quad \begin{aligned} \boldsymbol{H}_{C_0} &= [1\,0\,1\,1\,1\,0\,0] \\ \boldsymbol{H}_{C_1} &= [1\,1\,0\,1\,0\,1\,0] \\ \boldsymbol{H}_{C_2} &= [0\,1\,1\,1\,0\,0\,1] \end{aligned}$$

For decoding, each component code can be solved independently. The result of each component decoder is then passed to the others component code decoders and serves as an additional support information (a priori information). For 3GPP turbo codes the utilized decoding algorithm exchanges information between two component codes $C = \{C_1 \cap C_2\}$. LDPC codes are typically divided in M component codes $C = \{C_0 \cap C_1 \cap \ldots \cap C_{M-1}\}$, which means we decompose the original code, row-by-row, in M single parity check codes.

Each sub-code is decoded by utilizing a so called symbol-by-symbol MAP criterion or in short MAP algorithm. The MAP algorithm provides a posteriori probabilities for each symbol or bit. For each decoded bit x_i, the probability is calculated that this bit was either 0 or 1, given the received sequence $\boldsymbol{\lambda}$.

The MAP probability is derived by an example using a single parity check code. The same received sequence is the starting point, i.e. $\boldsymbol{\lambda} = \begin{bmatrix} 0.4 & -1.1 & 0.1 & 0.3 \end{bmatrix}$.

One possible first step is to evaluate the already explained cost function for all possible codewords.

$$C := \begin{cases} \boldsymbol{x}_0 = [0\,0\,0\,0] \\ \boldsymbol{x}_1 = [0\,0\,1\,1] \\ \boldsymbol{x}_2 = [0\,1\,0\,1] \\ \boldsymbol{x}_3 = [0\,1\,1\,0] \\ \boldsymbol{x}_4 = [1\,0\,0\,1] \\ \boldsymbol{x}_5 = [1\,0\,1\,0] \\ \boldsymbol{x}_6 = [1\,1\,0\,0] \\ \boldsymbol{x}_7 = [1\,1\,1\,1] \end{cases} \quad \Rightarrow \quad \sum_{i=0}^{N-1} \lambda_i x_i = \begin{cases} w_0 = 0 \\ w_1 = 0.4 \\ w_2 = -0.8 \\ w_3 = -1.0 \\ w_4 = 0.7 \\ w_5 = 0.5 \\ w_6 = -0.7 \\ w_7 = -0.3 \end{cases} \tag{3.25}$$

A labeling for each weight $w_c =< \boldsymbol{\lambda}, \boldsymbol{x}_c >$ is now introduced with an associated codeword numbering. $\boldsymbol{x}_c$ is one codeword of the codeword space C with the corresponding cost value w_c. The set of all possible weights is denoted as $\boldsymbol{w}$. The weight which associated to the ML codeword will be denoted as w_{ML}. Now we would like to evaluate the MAP probability for the first bit. Each cost w_c gives us a relative measure of a certain code word $\boldsymbol{x}_c$ with respect to a received $\boldsymbol{\lambda}$.

When we derived the cost function we have subtracted the all zero codeword, see Eq. 3.18. Thus each weight w_c reflects the log likelihood ratio of a certain codeword $\boldsymbol{x}_c$ with respect to the all zero codeword $\boldsymbol{x}_0$.

$$w_c = ln\left(\frac{P(\boldsymbol{x}_c|\boldsymbol{y})}{P(\boldsymbol{x}_0|\boldsymbol{y})}\right) \tag{3.26}$$

with $\boldsymbol{x}_c$ one codeword of the codeword space C. The MAP probability for the first bit position $P(x_0 = 0|\boldsymbol{y})$ gives the probability that the first bit position was sent as a zero, i.e., that either $\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2$ or $\boldsymbol{x}_3$ have been sent. These four codewords have all a zero at the first position. When looking on Eq. 3.26 we can evaluate this question by

$$P(x_0 = 0|\boldsymbol{y}) = \frac{e^{w_0} + e^{w_1} + e^{w_2} + e^{w_3}}{\text{normalizer}}. \tag{3.27}$$

The normalizer has to ensure that the probability over all codewords is 1. Again the normalizer can be omitted when building the likelihood ratio, or here directly the log-likelihood ratio.

$$\Lambda(x_0|\boldsymbol{y}) = ln\left(\frac{P(x_0 = 0|\boldsymbol{y})}{P(x_0 = 1|\boldsymbol{y})}\right) = ln\frac{e^{w_0} + e^{w_1} + e^{w_2} + e^{w_3}}{e^{w_4} + e^{w_5} + e^{w_6} + e^{w_7}} \tag{3.28}$$

This is the symbol-by-symbol MAP result in terms of log-likelihood ratios, since $\Lambda(x_0|\boldsymbol{y})$ is conditioned on an entire receive vector it denotes with a capital lambda. For the second bit position x_1 we have to evaluate

$$\Lambda(x_1|\boldsymbol{y}) = ln\left(\frac{P(x_1 = 0|\boldsymbol{y})}{P(x_1 = 1|\boldsymbol{y})}\right) = ln\frac{e^{w_0} + e^{w_1} + e^{w_4} + e^{w_5}}{e^{w_2} + e^{w_3} + e^{w_6} + e^{w_7}}. \tag{3.29}$$

A general expression for each bit position results in

$$\Lambda(x_i|\boldsymbol{y}) = ln\frac{\sum\limits_{\boldsymbol{x}|x_i=0} e^{w_c}}{\sum\limits_{\boldsymbol{x}|x_i=1} e^{w_c}} \tag{3.30}$$

$\boldsymbol{x}|x_i = 1$ is a codeword $\boldsymbol{x} \in C$ with the is bit set to 1. Equation 3.30 is one possible description to calculate the symbol-by-symbol MAP probability for any type of linear block code. The formulation is based on the weights of LLR values and can be used as starting point for implementation. The basic procedure to solve this equation is to

exploit the code properties to enable an intelligent search with respect to the current constraint codeword $\boldsymbol{x}|x_i$.

3.3.4 Max-Log-MAP Approximation

Equation 3.30 delivers the correct MAP result in the log-likelihood domain and is typically denoted as Log-MAP solution. The logarithm over the sum of exponential functions can expressed by the Jacobian logarithm either in its positive form with

$$\ln\left(e^{\delta_1} + e^{\delta_2}\right) = \max^*(\delta_1, \delta_2) = \max(\delta_1, \delta_2) + ln(1 + e^{-|\delta_1 - \delta_2|}), \tag{3.31}$$

or in its negative form with

$$-\ln\left(e^{-\delta_1} + e^{-\delta_2}\right) = \min^*(\delta_1, \delta_2) = \min(\delta_1, \delta_2) - ln(1 + e^{-|\delta_1 - \delta_2|}). \tag{3.32}$$

By utilizing the Jacobian logarithm and ignoring the correction term $ln(1+e^{-|\delta_1-\delta_2|})$ yields the so called Max-Log-MAP approximation which will be:

$$\begin{aligned} \Lambda(x_i|\boldsymbol{y}) &= ln\left(\frac{P(x_i = 0|\boldsymbol{y})}{P(x_i = 1|\boldsymbol{y})}\right) \\ &\approx + \max_{\boldsymbol{x}|x_i=0}\{\boldsymbol{w}\} - \max_{\boldsymbol{x}|x_i=1}\{\boldsymbol{w}\} \end{aligned} \tag{3.33}$$

Or we can express this function as well utilizing the minimum search by just multiplying it with -1.0; The resulting Max-Log-MAP expression results in:

$$\begin{aligned} \Lambda(x_i|\boldsymbol{y}) &= ln\left(\frac{P(x_i = 0|\boldsymbol{y})}{P(x_i = 1|\boldsymbol{y})}\right) \\ &\approx - \min_{\boldsymbol{x}|x_i=0}\left\{\sum_{i=0}^{N-1}\lambda_i x_i\right\} + \min_{\boldsymbol{x}|x_i=1}\left\{\sum_{i=0}^{N-1}\lambda_i x_i\right\} \\ &\approx - \min_{\boldsymbol{x}|x_i=0}\{\boldsymbol{w}\} + \min_{\boldsymbol{x}|x_i=1}\{\boldsymbol{w}\} \end{aligned} \tag{3.34}$$

Equation 3.34 searches through the weights $\boldsymbol{w}$, always with respect to one position i. Note that the weight of the ML solution w_{ML} will always part of the solution.

The sub-optimal Max-Log-MAP calculation becomes clearer when evaluating again the our single parity check example with the received sequence $\boldsymbol{\lambda} = \begin{bmatrix} 0.4 & -1.1 & 0.1 & 0.3 \end{bmatrix}$. The corresponding weights and codewords are shown in Eq. 3.25, the approximated Max-Log-MAP expression evaluates to:

$$\begin{aligned}
\Lambda(x_0|\boldsymbol{y}) &= -min\{0, 0.4, -0.8, -1.0\} + min\{0.7, 0.5, -0.7, -0.3\} = \quad 0.3 \\
\Lambda(x_1|\boldsymbol{y}) &= -min\{0, 0.4, 0.7, 0.5\} + min\{-0.8, -1.0, -0.7, -0.3\} = -1.0 \\
\Lambda(x_2|\boldsymbol{y}) &= -min\{0, -0.8, 0.7, -0.7\} + min\{0.4, -1.0, 0.5, -0.3\} = -0.2 \\
\Lambda(x_3|\boldsymbol{y}) &= -min\{0, -1.0, 0.5, -0.7\} + min\{0.4, -0.8, 0.7, -0.3\} = \quad 0.2
\end{aligned}$$

In each result of $\Lambda(x_i|\boldsymbol{y})$ the weight of the ML result $w_{ML} = -1.0$ is used. The sign bit of each LLR value calculated by the Max-Log-MAP approximation correspond to the maximum likelihood estimation $\hat{\boldsymbol{x}}$.

In hardware realizations we can either use the Max-Log-MAP approximation or the Log-MAP realization which mainly depends on the type of application.

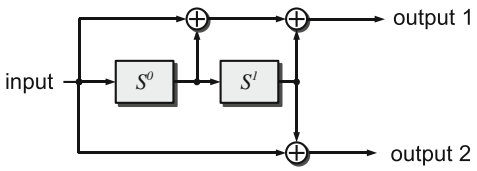
- A Max-Log-MAP implementation is chosen when the communications performance result of a Max-Log-MAP approximation is close to that of a Log-MAP realization. The difference with respect to communications performance is shown for turbo codes in Sect. 6.2.2.
- The computational complexity of the approximation term can be quite significant. However, sometime an approximation towards the Log-MAP solution becomes mandatory due to communications performance reasons. Then in hardware an approximation of the correction term is realized, one possible approximation is shown in Sect. 5.2.
- One major advantage of a MAX-Log-MAP realization is its independence of the signal-to-noise ratio, i.e. the algorithm is robust with respect to linear scaling of the input values provided by the demodulator. One example with respect to fixed-point realization and analysis of the robustness of an algorithm is shown in Sect. 6.2.3.

3.4 Convolutional Codes

Convolutional codes (CC) were already introduced in 1954 [12] and are still parts of many communication systems. This section just gives a pragmatic introduction to this type of codes. The theory for designing a convolutional codes and its mathematical description mandatory for in-depth analysis is omitted. An excellent in depth analysis is found in [1].

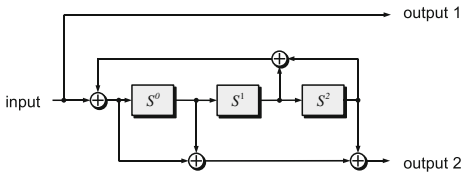
The encoder of a convolutional code is either a recursive or a non recursive filter, operating on bit level. It is composed of M memory elements, as shown for $M = 2$ in Fig. 3.2 and for $M = 3$ in Fig. 3.3 respectively.

Figure 3.2 shows a non-recursive encoder for a non-systematic convolutional (NSC) code. One input bit produces always two output bits. Thus this is a $R = \frac{1}{2}$ code. Depending on the values stored at any given moment in the storage elements S^0 and S^1 and on the input different output bits are produced. Since two storage devices are present $M = 2$, 4 different internal states can be reached (2^M) The so called transition table is shown at the right which shows the current state, input, next state, output1 and output2 values. Figure 3.3 shows a recursive systematic convolutional (RSC) code, again with transition table. Recursive means that there exist a feedback



current state	input	next state	out1-out2
00	0	00	00
00	1	10	11
01	0	00	11
01	1	10	00
10	0	01	10
10	1	11	01
11	0	01	01
11	1	11	10

Fig. 3.2 4-state NSC code with state transition table



current state	input	next state	out1-out2
000	0	000	00
000	1	100	11
001	0	100	00
001	1	000	11
010	0	101	01
010	1	001	10
011	0	001	01
011	1	101	10
100	0	010	01
100	1	110	10
101	0	110	01
101	1	010	10
110	0	111	00
110	1	011	11
111	0	011	00
111	1	111	11

Fig. 3.3 8-state RSC code with state transition table. This code is used in 3GPP turbo encoding

from the state information to the input bit. The plus boxes indicates an XOR of the participating bits.

The transition table describes a Mealy automaton, where the output depends on the state and the input information. For the 4-state NSC code the automaton is shown in Fig. 3.4. The finite state diagram does not contain any information in time, but unrolled over time steps results in the trellis diagram. For each time step k one new trellis step exists, which is shown for three time steps in Fig. 3.4. The trellis is one graphical representation for all possible output sequences.

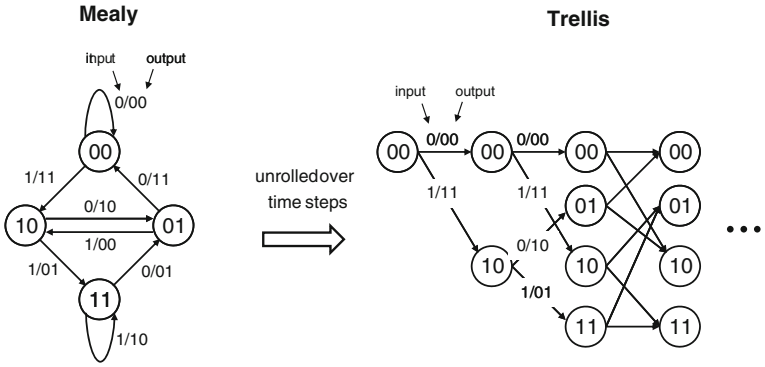


Fig. 3.4 State diagram unrolled over the time results in the trellis diagram

As described by the Mealy automaton we have in this example always two possibilities to reach a certain state. In the case of the encoding procedure of Fig. 3.4 this reflects two possible input/output sequences resulting in the same state.

In Fig. 3.5 the trellis is shown for two possible paths which converge again, e.g. there exist two paths which start from the same state and end up at the same state. Since a path is associated to an input bit/output bit combination one can see that the input sequence $[u_0\ u_1\ u_2] = [0\ 0\ 0]$ and $[u_0\ u_1\ u_2] = [1\ 0\ 0]$ have paths which merges again. The corresponding output sequence and thus part of the codeword are $[x_0\ x_1\ x_2\ x_3\ x_4\ x_5] = [0\ 0\ 0\ 0\ 0\ 0]$ and $[x_0\ x_1\ x_2\ x_3\ x_4\ x_5] = [1\ 1\ 1\ 0\ 1\ 1]$, respectively. This 6 bits are just a part of the codeword. Thus we have at least a minimum distance of the two codewords of $d_{min} = 5$.

As well convolutional codes are utilized in packet based transmission systems. At the end of the encoding of one block, typically the trellis is terminated in the zero state. This is enforced by adding tail bits to the sequence. The tail bits are chosen depending on the last encoding stage. This is shown for the 4-state NSC code of our example in Fig. 3.6. For a 2^M-state NSC code a sequence of M zeros can be used as input to enforce the termination in the zero state.

A formal method for describing a convolutional code is to give its generator matrix G in the D-transform notation, where D represents a delay operator. With respect to the encoding process of Fig. 3.2 we can say that the current input bit at time step $D^0 = 1$ is influenced by the bit of the previous time step D^1 and the bit at the input

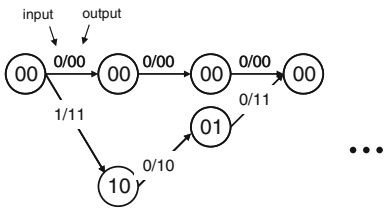


Fig. 3.5 Merging paths in a trellis

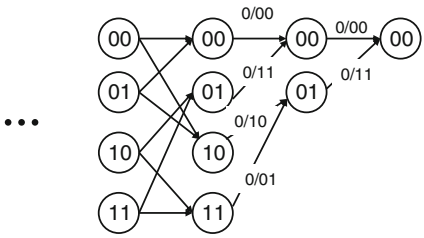


Fig. 3.6 Tail bits to enforce the final zero state

two time steps before D^2. The linear map of the input bits onto the coded bits can be written as a multiplication of the input sequence and a generator:

$$\boldsymbol{x}(D) = \boldsymbol{u}(D)G(D) \tag{3.35}$$

For the 4-state NSC encoder (Fig. 3.2) and the presented 8-state RSC encoder these generators are:

$$G(D) = [1 + D + D^2 \quad 1 + D^2] \qquad G(D) = \left[1 \quad \frac{1 + D^2 + D^3}{1 + D + D^3}\right] \tag{3.36}$$

In communication standards these generators are given typically in octal form which directly shows the structure for the encoding process. For the 4-state the so called forward polynomial for output one is $G_0 = 111_2 = 7_8$, and for output two is $G_1 = 101_2 = 5_8$. The binary form describes directly the taps of the NSC filter. Thus, the octal representation groups 3 taps positions per digit. For the recursive structure a standard typically defines a forward polynomial, here for the 8-state $G_0 = 13_8$ and the feedback polynomial as $G_{FB} = 15_8$. For different code rates more forward polynomials may exist, which are labeled by G_i.

Since the encoding scheme and as well the decoding scheme is quite simple many communication standards utilize convolution codes. Table 3.4 shows a selection of different convolutional codes utilized in various standards. The list shows the number of states, the code rate, and the defined polynomials. In a communication standards like defined by 3GPP there exist various different codes for e.g. different control channels or data channels. It quite surprising how many different encoder structures can be found when reading through the documentation of the standards.

3.4.1 ML Decoding of Convolutional Codes

For decoding we ideally would like to solve the ML criterion which is defined as:

$$\hat{\boldsymbol{x}} = \arg\min_{\boldsymbol{x} \in C} \left(\sum_{i=0}^{N-1} \lambda_i x_i \right)$$

Table 3.4 Selection of standards featuring convolutional codes, with the code type, number of states, and polynomials

Standard	Codes	States	Rate	Polynomials
	NSC	16	1/2	$G_0 = 23_8, G_1 = 33_8$
GSM/EDGE	NSC	16	1/3	$G_0 = 33_8, G_1 = 25_8, G_2 = 37_8$
	RSC	16	1/2	$G_0 = 33_8, G_{FB} = 23_8$
	RSC	16	1/3	$G_0 = 33_8, G_1 = 25_8, G_{FB} = 37_8$
GSM/EDGE	NSC	64	1/2	$G_0 = 123_8, G_1 = 171_8$
	NSC	64	1/3	$G_0 = 123_8, G_1 = 145_8, G_3 = 171_8$ or $G_3 = 175_8$
	NSC	64	1/4	$G_0 = 123_8, G_1 = 145_8, G_2 = 175_8, G_3 = 171_8$
GSM/EDGE	RSC	64	1/4	$G_0 = 123_8, G_1 = 145_8, G_2 = 175_8, G_{FB} = 171_8$
	RSC	64	1/4	$G_0 = 123_8, G_1 = 145_8, G_2 = 171_8, G_{FB} = 175_8$
UMTS	NSC	256	1/2	$G_0 = 561_8, G_1 = 753_8$
	NSC	256	1/3	$G_0 = 557, G_1 = 663_8, G_2 = 711_8$
DVB-H	CC	64	1/2	$G_0 = 171_8, G_1 = 133_8$
IEEE802.11a/g/n	CC	64	1/2	$G_0 = 133_8, G_1 = 171_8$
IEEE802.16e	CC	64	1/2	$G_0 = 133_8, G_1 = 171_8$

As seen in Sect. 3.3.2 a brute force check of all possible code words is quite cumber sum or even impossible for larger block sizes. The trellis however as introduced in the previous chapter is a graph structure which was obtained direct from the encoding process and represents all possible bit patterns. Solving the ML solution by using a trellis is equivalent to solving a shortest path problem and is denoted as Viterbi algorithm in the field of communications [13].

This is now demonstrated for a code Rate of $R = 1/2$. The sum of the cost function can be decomposed in partial sums:

$$\left(\sum_{i=0}^{N-1} \lambda_i x_i\right) = \left(\sum_{i=0}^{2k-1} \lambda_i x_i\right) + \underbrace{\left(\sum_{i=2k}^{2k+1} \lambda_i x_i\right)}_{partial \sum = branch\ metric} + \left(\sum_{i=2k+2}^{N-1} \lambda_i x_i\right) \tag{3.37}$$

The middle part, which reflects the partial sum of one encoding step, is of special interest. This partial sum is labeled with $\gamma_k^{x_i x_{i+1}}$ and are denoted as branch metrics. Four different branch metrics are possible, depending on the bit combinations of x_i and x_{i+1}.

$$\begin{aligned} \gamma_k^{00} &= 0 &&\rightarrow [x_i\ x_{i+1}] = [0\ 0] \\ \gamma_k^{01} &= \lambda_{i+1} &&\rightarrow [x_i\ x_{i+1}] = [0\ 1] \\ \gamma_k^{10} &= \lambda_i &&\rightarrow [x_i\ x_{i+1}] = [1\ 0] \\ \gamma_k^{11} &= \lambda_i + \lambda_{i+1} &&\rightarrow [x_i\ x_{i+1}] = [1\ 1] \end{aligned} \tag{3.38}$$

As already mentioned the trellis defines all possible bit combinations. A valid path from the beginning to end of the trellis defines a valid codeword. For each trellis step

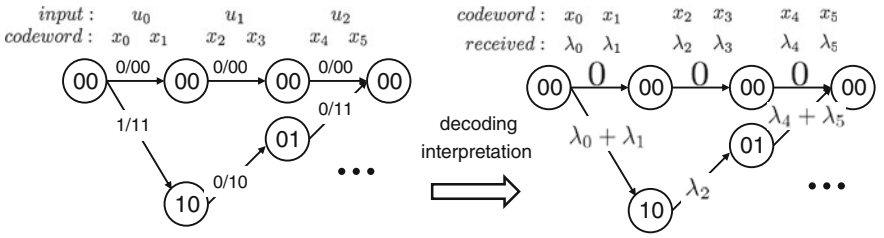


Fig. 3.7 Merging paths in a trellis

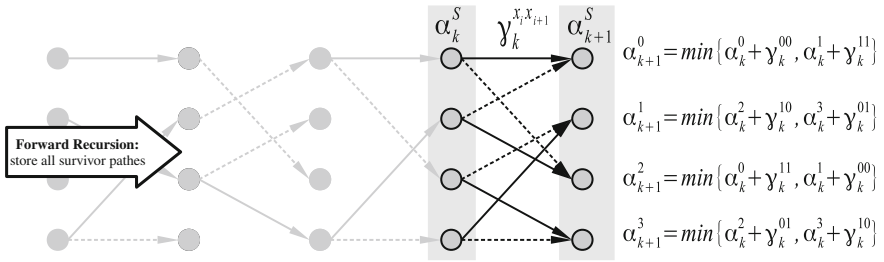


Fig. 3.8 Viterbi processing with the resulting surviver paths

k we can associate one intermediate sum value of Eq. 3.38 to one state transition, while i defines the position of the bit x_i within the codeword. The relation between trellis step k and codeword position i is $2 * k = i$, which results due to the code rate of $R = \frac{1}{2}$ within the example. One branch metric associated to an edge is denoted as:

$$\gamma_k^{x_i x_{i+1}}(S_k^m, S_{k+1}^{m'}) \tag{3.39}$$

which determines the mapping of a branch metric to a state transition or edge in the trellis, with S_k^m and $S_{k+1}^{m'}$ the connected states at trellis step k to $k+1$ with $m, m' \in \{0 \ldots 2^M - 1\}$. Fig. 3.7 shows 3 trellis steps which corresponds to 6 bits of the codeword. On the right side of Fig. 3.7 the paths are labeled with the corresponding $\sum \lambda_i x_i$ values. The labeling of the edges are exactly one branch metric of Eq. 3.38. The partial sum $(\lambda_0 + \lambda_1) + (\lambda_2) + (\lambda_4 + \lambda_5)$ comprises 3 encoding steps and is part of a valid cost function. However, the all zero path with the partial sum weight $0 + 0 + 0$ could be one part as well. Thus it makes sense only to follow the survivor of a minimum search

$$\alpha_{k=3}^0 = \min\{0 + 0 + 0\ ,\ (\lambda_0 + \lambda_1) + (\lambda_2) + (\lambda_4 + \lambda_5)\} \tag{3.40}$$

With $\alpha_{k=3}^0$ the so called state metric or current survivor sum at trellis step $k = 3$ and state S^0. The calculation has to be done for each state and each trellis step. We recursively calculate new state metrics α_{k+1}^S, with S the state number which is in this example for possible states with $S \in 0, 1, 2, 3$.

Figure 3.8 shows the basic sketch of the Viterbi processing at processing step k. Processing step and trellis step are identically in this example. The Viterbi algorithm which calculates the ML codeword can be divided in 3 major steps

- branch metric allocation: according to the state transition tables, allocate one of the corresponding branch metrics of Eq. 3.38 to an each edge in the trellis, according to Eq. 3.39.
- forward recursion: calculate at each time step and thus trellis step k the corresponding survivor path metric α_{k+1}^{m} for each state $m \in \{0 \ldots 2^M - 1\}$. We have to store these results for each state of the current processing step k. The α_{k+1}^{m} state metric hold an intermediate sum of the overall cost function of Eq. 3.37 to reach this particular state. It is the smallest sum which can be evaluated to reach this state, starting form the very first state α_0^0. In addition to the state metrics we have to store an indication about the surviver edge, i.e. where did I come from. The resulting surviver paths are indicated in Fig. 3.8 to the left of the current processing step.
- trace back: When reaching the final state at trellis step $k = N/2$ we have to extract the path which connects α_0^0 and $\alpha_{N/2}^0$. When we reached the last terminated state we obtained the final cost value of the entire code word $\alpha_{N/2}^0 = \sum_{i=0}^{N-1} \lambda_i x_i$. Obtaining the final cost value is, however, secondary to the objective of finding the ML path $\hat{\boldsymbol{x}}$. The way to obtain this path is quite elegant. For each time step we have stored the already indicated survivor at each trellis step. When reaching the final state we now use a back tracing of the path information to obtain $\hat{\boldsymbol{x}}$ which is thus obtained in a reversed order.

3.4.2 Max-Log-MAP Decoding of Convolutional Codes

The symbol-by-symbol maximum a posteriori algorithm, short MAP algorithm, provides a posteriori probabilities for each symbol or bit. The original algorithm in probability domain for trellis codes was already published in 1974 [14] and named after their inventors BCJR algorithm. However, the algorithm was not of practical importance since it is more complex than the Viterbi algorithm and has no advantage when decoding only convolutional codes. The Viterbi algorithm find the optimum sequence while the MAP algorithm provides additional reliability information for each position. First the invention of iterative decodable codes turned the focus again on the MAP algorithm. In iterative decoding the 'soft' information plays an important role.

In hardware the processing is always done in the so already mentioned log-likelihood domain. For each decoded bit x_i, the Log-Likelihood Ratio (LLR) is calculated that this bit was 0 or 1, given the received symbol sequence.

$$\Lambda(x_i|\boldsymbol{y}) = ln\left(\frac{P(x_i = 0|\boldsymbol{y})}{P(x_i = 1|\boldsymbol{y})}\right) \tag{3.41}$$

Solving the Log-MAP or Max-Log-MAP criterion in a brute force manner was already shown in Sect. 3.3.3. Here we show the procedure with respect to a trellis representation. The MAP processing on a trellis is a so called forward-backward algorithm and is shown for trellis step k in Fig. 3.9. The processing can be decomposed in 4 steps.

- branch metric allocation: according to the state transition tables, allocate one of the corresponding $\gamma_k^{x_i x_{i+1}}$ values of Eq. 3.38 to each edge in the trellis. This part is identical to the Viterbi algorithm
- forward recursion: calculate at each time step and thus trellis step k the corresponding path metric α_{k+1}^S for every state S. We have to store these results for each state S of all processing step up to trellis step k. Only the current state metrics at a given trellis step are utilized for the next time step. For the Max-Log-MAP algorithm exactly the same α or forward recursion is utilized which can be expressed in a more general form as:

$$\alpha_{k+1}^{m'} = \min_{\forall(m->m')} \left(\alpha_k^m + \gamma_k^{x_i,x_i+1}(S_k^m, S_{k+1}^{m'}) \right) \tag{3.42}$$

  We have find the minimum over all possibilities which connect states at the previous time step to $S_{k+1}^{m'}$.
- backward recursion: the backward recursion has exactly the same recursive functionality, however starting with the calculation from the last state in the trellis.

$$\beta_k^m = \min_{\forall(m'->m)} \left(\beta_{k+1}^{m'} + \gamma_k^{x_i,x_i+1}(S_k^m, S_{k+1}^{m'}) \right) \tag{3.43}$$

  Again we store all obtained results for each trellis step.
- soft-output calculation: we would like to calculate the symbol-by-symbol Max-Log MAP result. For this we need the results of the forward recursion, backward recursion and the current branch metric, which is:

$$\begin{aligned} \Lambda(x_i, \boldsymbol{y}) = & \min_{\forall\gamma|x_i=0} \left(\gamma_k^{x_i,x_i+1}(S_k^m, S_{k+1}^{m'}) + \alpha_k^m + \beta_{k+1}^{m'} \right) \\ & - \min_{\forall\gamma|x_i=1} \left(\gamma_k^{x_i,x_i+1}(S_k^m, S_{k+1}^{m'}) + \alpha_k^m + \beta_{k+1}^{m'} \right). \end{aligned} \tag{3.44}$$

  We are searching the minimum over all sum of weights which has at trellis step k under the condition that the corresponding codeword bit is either zero or one.

The individual processing steps of the Max-Log-MAP algorithm are explained now by a small example. Note, that for practical hardware implementation for turbo decoding only this algorithm is implemented, more details about this in the turbo decoder chapter. The utilized channel code in the following example is a simple 2-state RSC with $G_0 = 1$ and $G_{FB} = 3_8$, its mealy automaton ins displayed next to its trellis representation, see Fig. 3.10.

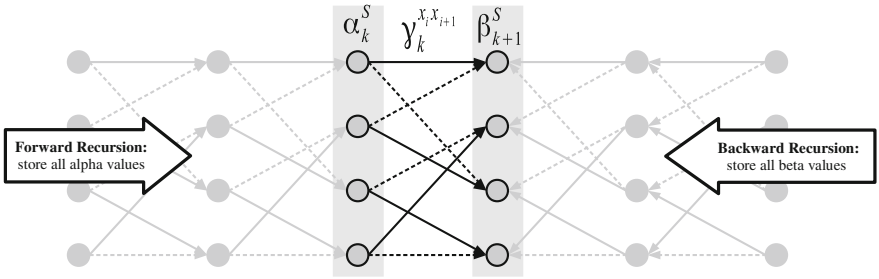


Fig. 3.9 MAP processing with the forward and backward recursion

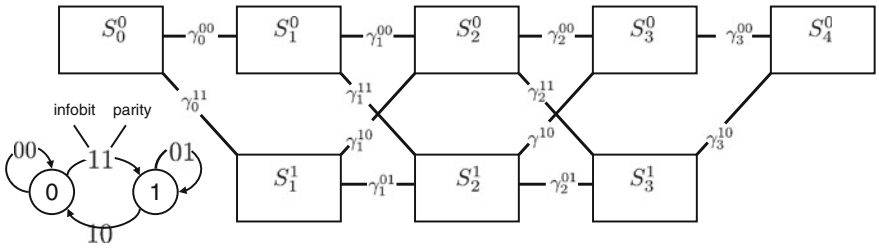


Fig. 3.10 *Step one: set up the corresponding trellis and the labeling*. The two state trellis with 4 trellis steps is shown. For each trellis step i and each edge a branch metric γ exists. Depending on the possible systematic, parity bit combination $x_i^s x_i^p$ a different branch metric has to be allocated

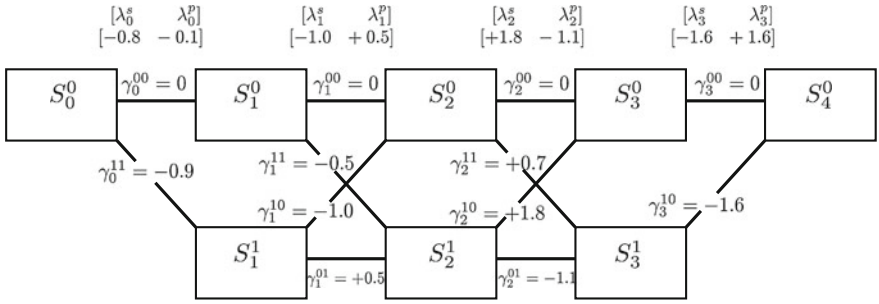


Fig. 3.11 *Step two: allocate the corresponding branch values*. The corresponding λ-values for each trellis step are shown at the top. The branch metrics with the corresponding values are now updated (the trellis step label k is omitted here). Each branch metric is calculated according to Eq. 3.38

The information word used for demonstration has just 4 bits, with $\boldsymbol{u} = [0101]$. The resulting codeword after encoding is the concatenation of the systematic part $\boldsymbol{x^s}$ and the parity part $\boldsymbol{x^p}$. The codeword is modulated via a binary phase shift keying (BPSK), which mapping $0 \rightarrow +1$ and $1 \rightarrow -1$ respectively. After demodulation we receive the LLRs. The values of the resulting codeword, the sent symbols, and the noise corrupted decoder input are the following:

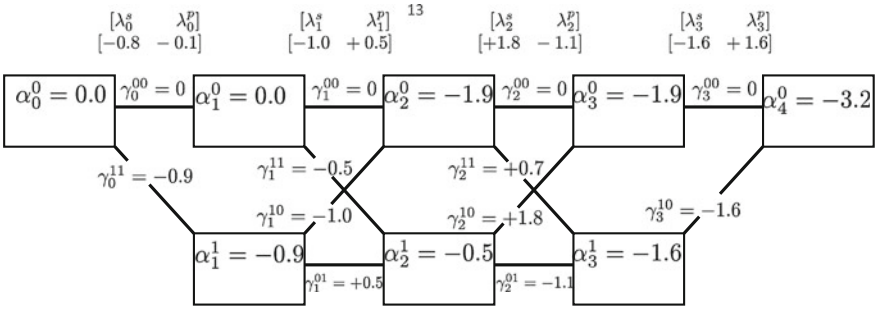


Fig. 3.12 *Step three: calculate the forward state metrics* α: In this example only two possible transitions exist between the previous states and the state we would like to calculate. Since only two states exist in this example, only two equations have to be solved for α_{k+1}^0 and α_{k+1}^1 respectively

$$\alpha_{k+1}^0 = \min\{\alpha_k^0 + \gamma_k^{00}, \alpha_k^1 + \gamma_k^{10}\}$$
$$\alpha_{k+1}^1 = \min\{\alpha_i^0 + \gamma_k^{11}, \alpha_k^1 + \gamma_k^{01}\}$$

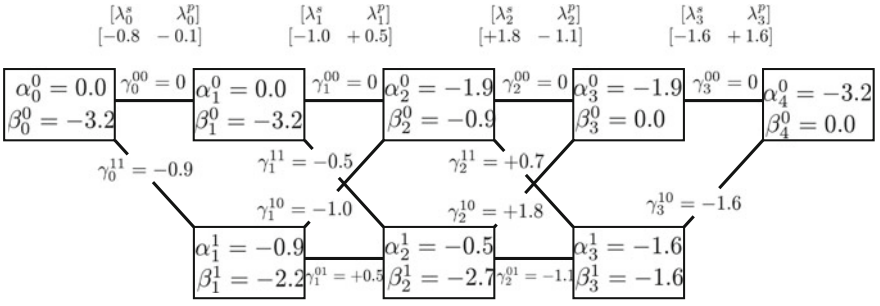


Fig. 3.13 *Step four: calculate the backward state metrics* β: The calculation is identical to that of the forward recursion, again only two equations have to be solved at each trellis step:

$$\beta_k^0 = \min\{\beta_{k+1}^0 + \gamma_k^{00}, \beta_{k+1}^1 + \gamma_k^{11}\}$$
$$\beta_k^1 = \min\{\beta_{k+1}^0 + \gamma_k^{10}, \beta_{k+1}^1 + \gamma_k^{01}\}$$

codeword:
$\boldsymbol{x} = [\boldsymbol{x^s}\ \boldsymbol{x^p}]$ = [[0 1 0 1] [0 1 1 0]]
sent symbol:
$\boldsymbol{s}$ = [+1 −1 +1 −1 +1 −1 −1 +1]
received:
$\boldsymbol{\lambda} = [\boldsymbol{\lambda^s}\ \boldsymbol{\lambda^p}]$ = [[−0.8 −1.0 +1.8 −1.6] [−0.1 +0.5 −1.1 +1.6]]

For the decoding algorithm we several steps have to be done, see Figs. 3.10, 3.11, 3.12, 3.13 and 3.14.

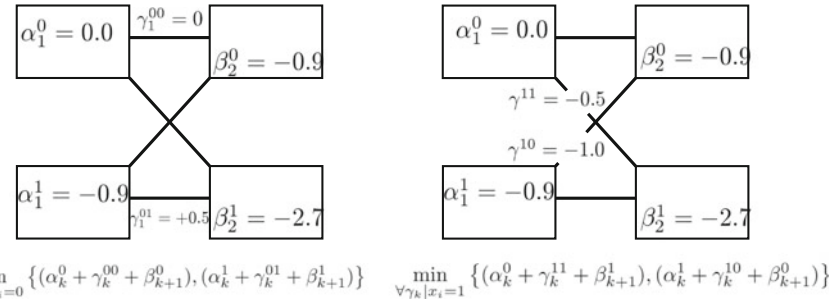


Fig. 3.14 *Step five: calculate the output Max-Log-MAP approximation:* We calculate now the output result by evaluating: $\Lambda(x_i|\boldsymbol{y}) \approx -\min_{\gamma_k|x_i=0}\{\ldots\} + \min_{\gamma_k|x_i=1}\{\ldots\}$. The figure shows the participating α, β, and γ values to calculate trellis step $k = 1$. The left side for all possible $\gamma_k|x_i = 0$ transitions and the right side of the figure shows the equation for the $\gamma_k|x_i = 0$ transition. $\alpha_0^1 = inf$ and $\beta_4^1 = inf$ are both initialized with an infinite value, since these states can not be reached. Zero values are not explicitly stated within the sum terms. For each position we have to evaluate these equations which are shown in the following:

$$
\begin{aligned}
\Lambda(x_0|\boldsymbol{y}) &\approx -\min_{\gamma_0|x_0=0}\{(-3.2), \inf\} \\
&\quad + \min_{\gamma_0|x_0=1}\{(-0.9 - 2.2), \inf\} && = 0.1 \\
\Lambda(x_1|\boldsymbol{y}) &\approx -\min_{\gamma_1|x_1=0}\{(-0.9), (-0.9 + 0.5 - 2.7)\} \\
&\quad + \min_{\gamma_1|x_1=1}\{(-0.5 - 2.7), (-0.9 - 1.0 - 0.9)\} && = -0.1 \\
\Lambda(x_2|\boldsymbol{y}) &\approx -\min_{\gamma_2|x_2=0}\{(-1.9), (-0.5 - 1.1 - 1.6)\} \\
&\quad + \min_{\gamma_2|x_2=1}\{(-1.9 + 0.7 - 1.6), (-0.5 + 1.8)\} && = 0.4 \\
\Lambda(x_3|\boldsymbol{y}) &\approx -\min_{\gamma_3|x_3=0}\{(-1.9), \inf\} \\
&\quad + \min_{\gamma_3|x_3=1}\{\inf, (-1.6 - 1.6)\} && = -1.3
\end{aligned}
\tag{3.45}
$$

3.5 Soft-Input Soft-Output (SISO) Decoder

The previous example is summarized with the following 4 lines, the sent codeword, sent symbol, received LLR, and output result respectively.

$$
\begin{aligned}
\boldsymbol{x} = [\boldsymbol{x}^s\ \boldsymbol{x}^p] &= [[\ \ 0\ \ \ 1\ \ \ 0\ \ \ 1]\ \ [0\ \ \ 1\ \ \ 1\ \ \ 0\ ]] \\
\boldsymbol{s} &= [\ +1\ \ -1\ \ +1\ \ -1\ \ +1\ \ -1\ \ -1\ \ +1\ ] \\
\boldsymbol{\lambda} = [\boldsymbol{\lambda}^s\ \boldsymbol{\lambda}^p] &= [[-0.8\ -1.0\ +1.8\ -1.6]\ [-0.1\ +0.5\ -1.1\ +1.6\ ]] \\
\boldsymbol{\Lambda}(x_i|\boldsymbol{y}) &= [[\ 0.1\ -0.1\ \ 0.4\ -1.3]\ \ [x\ \ \ x\ \ \ x\ \ \ x\ ]]
\end{aligned}
$$

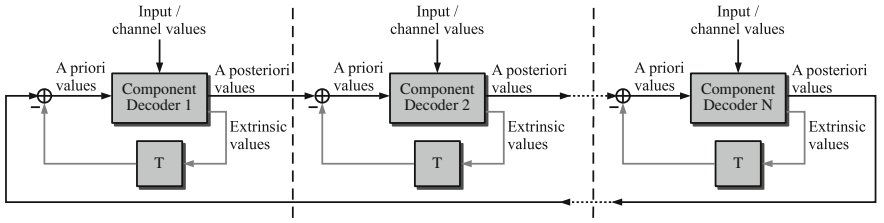


Fig. 3.15 Concatonation of component decoders for soft-in soft-out decoding

As mentioned the result of the symbol-by-symbol MAP algorithm is denoted as well as soft-output information. Typically the decoder utilizes all soft-input information to calculate this soft-output information. The gain we obtained by the soft-input soft-output algorithm is denoted as extrinsic information. It is calculated by taking the difference between the obtained information of the MAP algorithm and the input information.

$$\text{Extrinsic LLR } \boldsymbol{L}^e = \boldsymbol{\Lambda} - \boldsymbol{\lambda} = [\ 0.9\ 0.9\ -1.4\ 0.3\ ]$$

The **extrinsic information** of a symbol or bit is the additional information a (MAP) decoder calculates. The decoder calculates the a posteriori information taking all input information into account. The extrinsic information is thus the a posteriori information excluding the entire input information.

Typically, for decoding we have two types of soft-input information. The channel information with here denoted as $\boldsymbol{\lambda}$ and an additional information which is called **a priori information** ($\boldsymbol{L}^a$) which is in this example zero. The a priori information of symbol or bit is an additional information known before decoding. This information may come from a source independent of the received sequence.

The efficient usage of a priori and extrinsic information turned in to focus when Berrou and Glavieux presented the turbo codes (TC) [15]. These channel codes are decoded in an iterative manner and are already applied in many communication standards [16, 17]. Note that already low-density parity-check (LDPC)codes, introduce in 1963 utilized the iterative decoding.

The success of the iterative decoding process is the efficient usage of extrinsic and a priori information. The major principle of **all** iterative channel code decoders is the exchange of reliability information, in terms of LLRs, between 2 or more component decoders. The basic code structure of TCs is the random concatenation of 2 component codes. In the case of LDPC codes many simple component codes are concatenated. Both code types and the implementation of the decoding algorithm are explained in detail in the next chapters.

The concatenation of different component codes for decoding is shown in Fig. 3.15. For each component code a corresponding component decoder exist. A component decoder calculates a local MAP information of the bits to be decoded. These information is re-sorted according to the concatenation of the component

codes and then passed to the connected component codes. The re-sorting process is denoted as interleaving in the following, see Sect. 4.3. One iteration is finished if each component code has updated the information ones.

The outstanding communications performance of concatenated codes can only be obtained when each component decoder is utilizing soft information at the input and calculates new soft information at the output which is denoted as SISO decoder and does not really tell something about the utilized decoding algorithm like Log-MAP, Max-Log-MAP, or other algorithms.

The output a posteriori probabilities ($\boldsymbol{\Lambda}$) can be decomposed in three parts: the channel input information, the additional gain (extrinsic information), and the a priori information:

$$\boldsymbol{\Lambda} = \boldsymbol{\lambda} + \boldsymbol{L}^a + \boldsymbol{L}^e \tag{3.46}$$

Only the additional gain ($\boldsymbol{L}^e$) is passed to the other component decoders with respect to the connectivity structure. Which means, we subtract the input LLRs as well as the input a priori information:

$$\boldsymbol{L}^e = \boldsymbol{\Lambda} - \boldsymbol{\lambda} - \boldsymbol{L}^a \tag{3.47}$$

This additional information serves now as a priori information for an other component code decoder and is besides the channel information one part of the soft-input information.

Without subtracting the old input information a SISO decoder (Eq. 3.47) would just confirm its decision of prior iterations.

References

1. Lin, S., Costello, D.J. Jr.: Error Control Coding, 2nd edn. Prentice Hall PTR, Upper Saddle River (2004)
2. Bossert, M.: Kanalcodierung, 2nd edn. B.G. Teubner, Stuttgart (1998)
3. Breitbach, M., Bossert, M., Lucas, R., Kempter, C.: Soft-decision decoding of linear block codes as optimization problem. Eur. Trans. Telecommun. **9**(3), 289–293. doi:10.1002/ett.4460090308. http://dx.doi.org/10.1002/ett.4460090308 (1998)
4. Tanatmis, A., Ruzika, S., Hamacher, H.W., Punekar, M., Kienle, F., Wehn, N.: A separation algorithm for improved LP-decoding of linear block codes. IEEE Trans. Inf. Theor. **56**(7), 3277–3289 (2010). doi:10.1109/TIT.2010.2048489
5. Scholl, S., Kienle, F., Helmling, M., Ruzika, S.: ML vs. MP decoding of binary and non-binary LDPC codes. In: Proceedings of the 7th International Symposium on Turbo Codes and Iterative Information Processing (2012)
6. Wolsey, L.A., Nemhauser, G.L.: Integer and Combinatorial Optimization. Wiley-Interscience, New York (1999)
7. Free Software Foundation Inc.: GLPK—GNU Linear Programming Kit. http://www.gnu.org/software/glpk/. Accessed Apr 2012
8. IBM: IBM ILOG CPLEX Optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/. Accessed Apr 2012

9. Punekar, M., Kienle, F., Wehn, N., Tanatmis, A., Ruzika, S., Hamacher, H.W.: Calculating the minimum distance of linear block codes via integer programming. In: Proceedings of the 6th International Turbo Codes and Iterative Information Processing (ISTC) Symposium, pp. 329–333 (2010). doi:10.1109/ISTC.2010.5613894
10. Scholl, S., Kienle, F., Helmling, M., Ruzika, S.: Integer programming as a tool for code analysis. In: 9th International ITG Conference on Systems, Communications and Coding (2013) (Accepted)
11. Berlekamp, E., McEliece, R., van Tilborg, H.: On the inherent intractability of certain coding problems. IEEE Trans. Inf. Theor. **24**(3), 384–386 (1978). doi:10.1109/TIT.1978.1055873
12. Elias, P.: Error-free coding. IEEE Trans. Inf. Theor. **4**(4), 29–39 (1954)
13. Viterbi, A.J.: Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. IEEE Trans. Inf. Theor. **13**(2), 260–269 (1967)
14. Bahl, L., Cocke, J., Jelinek, F., Raviv, J.: Optimal decoding of linear codes for minimizing symbol error rate. IEEE Trans. Inf. Theor. IT-20, 284–287 (1974)
15. Berrou, C., Glavieux, A., Thitimajshima, P.: Near shannon limit error-correcting coding and decoding: turbo-codes. In: Proceedings of the 1993 International Conference on Communications (ICC '93), pp. 1064–1070. Geneva, Switzerland (1993)
16. Third Generation Partnership Project: 3GPP TS 25.212 V1.0.0; 3rd Generation Partnership Project (3GPP);Technical Specification Group (TSG) Radio Access Network (RAN); Working Group 1 (WG1); Multiplexing and channel coding (FDD). http://www.3gpp.org (1999)
17. Third Generation Partnership Project 2: 3GPP2 C.S0002-A. http://www.3gpp2.org (2000)