Corso di Information Theory and Coding

Prof. Francesco A. N. Palmieri

Dipartimento di Ingegneria, Università della Campania Luigi Vanvitelli

Corso di Laurea Magistrale in Ingegneria Informatica

AA 2020-21

CODICI CONVOLUZIONALI

Estratto dal libro:

S. Benedetto, E Biglieri, *Priciples of Digital Transmission with* Wireless Applications, Kluwer Academic Press, 1999

11

Convolutional and concatenated codes

With block codes, the information sequence is segmented into blocks that are encoded independently to form the coded sequence as a succession of fixed-length independent code words. *Convolutional* codes behave differently. The n_0 bits that the convolutional encoder generates in correspondence of the k_0 information bits depend on the k_0 data bits and also on some previous data frames (see Section 10.1): the encoder has *memory*.

Convolutional codes differ deeply from block codes, in terms of their structure, analysis and design tools. Algebraic properties are of great importance in constructing good block codes and in developing efficient decoding algorithms. Good convolutional codes, instead, have been almost invariably found by exhaustive computer search, and the most efficient decoding algorithms (like the Viterbi maximum-likelihood algorithm and the sequential algorithm) stem directly from the sequential-state machine nature of convolutional encoders, rather than from the algebraic properties of the code.

In this chapter, we will start by establishing the connection of binary convolutional codes with linear block codes, and then widen the horizon by assuming a completely different point of view that looks at a convolutional encoder as a finite-state machine and introduces the code *trellis* as the graphic tool describing all possible code sequences.

We will show how to evaluate the distance properties of the code and the error probability performance, and describe in details the application of the Viterbi algorithm to its decoding. A brief introduction to sequential and threshold decoding will also be given.

The second part of the chapter is devoted to *concatenated* codes, a concept

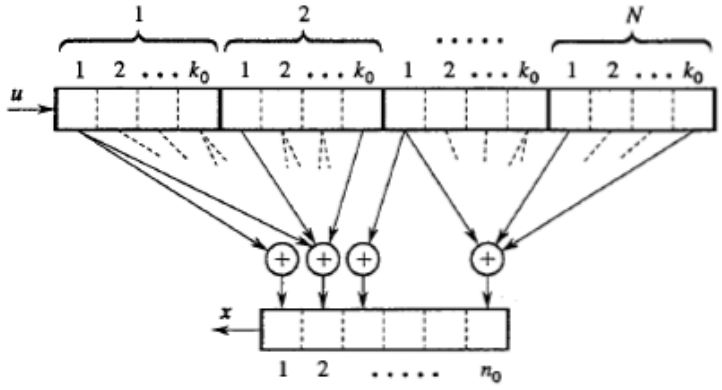


Figure 11.1: *General block diagram of a convolutional encoder in serial form for a* (n_0, k_0) *code with constraint length* N.

first introduced by Forney (1966) that has since then found a wide range of applications. After describing the classical concatenation schemes, we will also devote some space to the recently introduced "turbo" codes, a very promising new class of concatenated codes that approach the capacity coding gains at medium-to-low bit error probabilities.

11.1. Convolutional codes

A binary convolutional encoder is a finite-memory system that outputs n_0 binary digits for every k_0 information digits presented at its input. Again, the code rate is defined as $R_c = k_0/n_0$. In contrast with block codes, k_0 and n_0 are usually small numbers. A scheme that serially implements a *linear, feedforward* binary convolutional encoder is shown in Fig. 11.1. The message digits are introduced k_0 at a time into the input shift register, which has Nk_0 positions. As a block of k_0 digits enters the register, the n_0 modulo-2 adders feed the output register with the n_0 digits and these are shifted out. Then the input register is fed with a new block of k_0 digits, and the old blocks are shifted to the right, the oldest one being lost. And so on. We can conclude that in a convolutional code the n_0 digits generated by the encoder depend not only on the corresponding k_0 message digits, but also on the previous $(N-1)k_0$ ones, whose number constitutes the *memory* $\nu \triangleq (N-1)k_0$ of the encoder. Such a code is called an (n_0, k_0, N) convolutional code. The parameter N, the number of data frames contained in

the input register, is called the *constraint length* of the code.[1] With reference to the encoder of Fig. 11.1, a block code can be considered to be the limiting case of a convolutional code, with constraint length $N = 1$.

If we define $\mathbf{u}$ to be the semi-infinite message vector and $\mathbf{x}$ the corresponding encoded vector, we want now to describe how to get $\mathbf{x}$ from $\mathbf{u}$. As for block codes, to describe the encoder we only need to know the connections between the input and output registers of Fig. 11.1. This approach enables us to show both the analogies and the differences with respect to block codes. But, if pursued further, it would lead to complicated notations and tend to emphasize the algebraic structure of convolutional codes. This is less interesting for decoding purposes. Therefore, we shall only sketch this approach briefly. Later, the description of the code will be restated from a different viewpoint.

To describe the encoder of Fig. 11.1, we can use N submatrices $\mathbf{G}_1, \mathbf{G}_2, \mathbf{G}_3, \ldots, \mathbf{G}_N$ containing k_0 rows and n_0 columns. The submatrix $\mathbf{G}_i$ describes the connections of the i-th segment of k_0 cells of the input register with the n_0 cells of the output register. The n_0 entries of the first row of $\mathbf{G}_i$ describe the connections of the first cell of the i-th input register segment with the n_0 cells of the output register. A "1" in $\mathbf{G}_i$ means a connection, while a "0" means no connection. We can now define the generator matrix of the convolutional code as

$$\mathbf{G}_\infty \triangleq \begin{bmatrix} \mathbf{G}_1 & \mathbf{G}_2 & \ldots & \mathbf{G}_N & & & & \\ & \mathbf{G}_1 & \mathbf{G}_2 & \ldots & \mathbf{G}_N & & & \\ & & \mathbf{G}_1 & \mathbf{G}_2 & \ldots & \mathbf{G}_N & & \\ & & & \mathbf{G}_1 & \mathbf{G}_2 & \ldots & \mathbf{G}_N & \\ & & & & \ldots & \ldots & \ldots & \ldots \end{bmatrix} \tag{11.1}$$

All other entries in $\mathbf{G}_\infty$ are zero. This matrix has the same properties as for block codes, except that it is semi-infinite (it extends indefinitely downward and to the right). Therefore, given a semi-infinite message vector $\mathbf{u}$, the corresponding coded vector is

$$\mathbf{x} = \mathbf{u}\mathbf{G}_\infty \tag{11.2}$$

This equation is formally identical to (10.4). A convolutional encoder is said to be *systematic* if, in each segment of n_0 digits that it generates, the first k_0 are a replica of the corresponding message digits. It can be verified that this condition

[1] The reader should be warned that there is no unique definition of constraint length in the convolutional code literature.

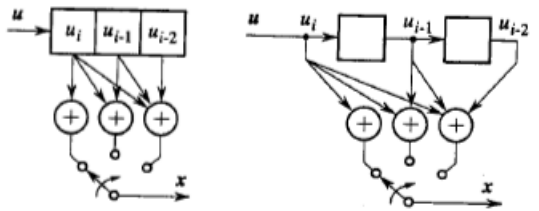


Figure 11.2: *Two equivalent schemes for the convolutional encoder of the (3,1,3) code of Example 11.1.*

is equivalent to have the following $k_0 \times n_0$ submatrices:

$$\mathbf{G}_1 = \left[\begin{array}{ccccc|c} 1 & 0 & 0 & \ldots & 0 & \\ 0 & 1 & 0 & \ldots & 0 & \\ 0 & 0 & 1 & \ldots & 0 & \mathbf{P}_1 \\ \ldots & \ldots & \ldots & \ldots & & \\ 0 & 0 & 0 & \ldots & 1 & \end{array}\right] \tag{11.3}$$

and

$$\mathbf{G}_i = \left[\begin{array}{ccccc|c} 0 & 0 & 0 & \ldots & 0 & \\ 0 & 0 & 0 & \ldots & 0 & \\ 0 & 0 & 0 & \ldots & 0 & \mathbf{P}_i \\ \ldots & \ldots & \ldots & \ldots & & \\ 0 & 0 & 0 & \ldots & 0 & \end{array}\right] \tag{11.4}$$

for $i = 2, 3, \ldots, N$. All these concepts are better clarified with two examples.

Example 11.1 Consider a (3,1,3) convolutional code. Two equivalent schemes for the encoder are shown in Fig. 11.2. The first uses a register with three cells, whereas the second uses two cells, each introducing a unitary delay. The output register is replaced by a commutator that reads sequentially the outputs of the three adders. The encoder is specified by the following three submatrices (actually, three row vectors, since $k_0 = 1$):

$$\begin{aligned} \mathbf{G}_1 &= [1\ 1\ 1] \\ \mathbf{G}_2 &= [0\ 1\ 1] \\ \mathbf{G}_3 &= [0\ 0\ 1] \end{aligned}$$

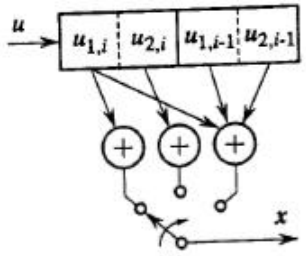


Figure 11.3: *Convolutional encoder for the (3,2,2) code of Example 11.1.*

The generator matrix, from (11.1), becomes

$$\mathbf{G}_\infty = \begin{bmatrix} 111 & 011 & 001 & 000 & \dots & \dots & \dots \\ 000 & 111 & 011 & 001 & 000 & \dots & \dots \\ 000 & 000 & 111 & 011 & 001 & 000 & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \end{bmatrix}$$

It can be verified, from (11.2), that the information sequence $\mathbf{u} = (11011\dots)$ is encoded into the sequence $\mathbf{x} = (111100010110100\dots)$. The encoder is systematic. Notice that the code sequence can be obtained by summing modulo-2 the rows of $\mathbf{G}_\infty$ corresponding to the "1" in the information sequence, as for block codes. □

Example 11.2 Consider a (3,2,2) code. The encoder is shown in Fig. 11.3. The code is now defined by the two submatrices

$$\mathbf{G}_1 = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \qquad \mathbf{G}_2 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

The encoder is systematic, since (11.3) and (11.4) are satisfied. The generator matrix is now given by

$$\mathbf{G}_\infty = \begin{bmatrix} 101 & 001 & 000 & \dots & \dots \\ 010 & 001 & 000 & \dots & \dots \\ 000 & 101 & 001 & 000 & \dots \\ 000 & 010 & 001 & 000 & \dots \\ 000 & 000 & 101 & 001 & 000 \\ 000 & 000 & 010 & 001 & 000 \\ \dots & \dots & \dots & \dots & \dots \end{bmatrix}$$

The information sequence $\mathbf{u} = (11011011\dots)$ is encoded into the code sequence $\mathbf{x} = (111010100110\dots)$. □

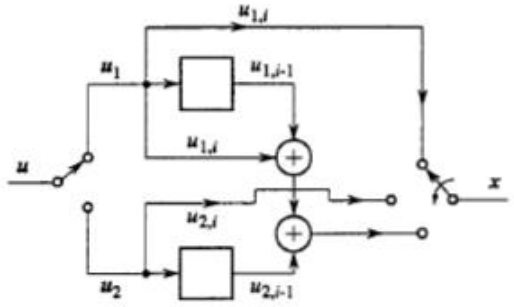


Figure 11.4: *Parallel implementation of the same convolutional encoder of Fig. 11.3.*

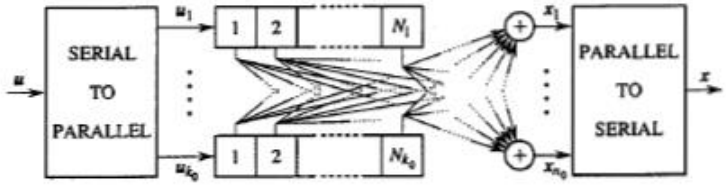


Figure 11.5: *General block diagram of a convolutional encoder in parallel form for an (n_0, k_0, N) code.*

The encoder of Fig. 11.3 requires a serial input. The $k_0 = 2$ input digits can also be presented in parallel, and the corresponding encoder is given in Fig. 11.4.

The parallel representation of the encoder, shown for a general (n_0, k_0) encoder in Fig. 11.5. is more flexible than the serial one of Fig. 11.1, as it allows allocation of a different number of register cells in each parallel section. When $N_i = N$, $\forall i$, we can define the constraint length N as in the case of the serial representation. When the N_i's are different, we define the constraint length N as the largest among the N_i's, i.e., $N \triangleq \max_i N_i$, $i = 1, \dots, k_0$. The encoder memory is in this case $\nu = \cdot \sum_{i=0}^{k_0} (N_i - 1)$.

If we look for a physical meaning of the constraint length, $N - 1$ represents the maximum number of trellis steps (see Section 11.1.1) that are required to return from any state of the encoder to the zero state. This "remerge" operation, called *trellis termination*, is required to transform K sections of the trellis of a convolutional code into a block code with parameters $k = k_0 \cdot K$, $n = n_0 \cdot (K + N - 1)$. Sometimes, system constraints impose a frame (or "burst") structure on the information stream. For short bursts, terminated convolutional codes are used, and each burst is decoded using the Viterbi algorithm without truncation

(see 11.1.3).

Notice that, in the more general case of different N_i's, the structure of the encoder is not identified by the three parameters (n_0, k_0, N); instead, beyond k_0 and n_0, the whole set $\{N_i\}_{i=1}^{k_0}$ is needed.

From Example 11.1, it can be verified that the operation of the encoder for an $(n_0, 1)$ code is to generate n_0 digits of the sequence $\mathbf{x}$ for each digit u_i according to the following expression:

$$\begin{aligned}(x_{i1}, x_{i2}, x_{i3}, \ldots, x_{in_0}) &= u_i\mathbf{G}_1 + u_{i-1}\mathbf{G}_2 + \ldots + u_{i-N+1}\mathbf{G}_N \\ &= \sum_{k=1}^{N} u_{i-k+1}\mathbf{G}_k \end{aligned} \tag{11.5}$$

This is the discrete convolution of the vectors $\mathbf{G}_1, \mathbf{G}_2, \ldots, \mathbf{G}_N$ and the N-digit input sequence $(u_i, u_{i-1}, \ldots, u_{i-N+1})$. The term *convolutional* code stems from this observation.

Quite often, the number of modulo-2 adders in the encoder is smaller than the constraint length of the code. In fact, code rates of 1/2 or 1/3 are widely used, and in these cases we have only two or three adders, respectively. For this reason, instead of describing the code with the N submatrices $\mathbf{G}_i$ previously introduced, it is more convenient to describe the encoder connections by using the *transfer function* matrix $\mathbf{G}$

$$\mathbf{G} = \begin{bmatrix} \mathbf{g}_{1,1} & \cdots & \mathbf{g}_{1,n_0} \\ \vdots & & \\ \mathbf{g}_{k_0,1} & \cdots & \mathbf{g}_{k_0,n_0} \end{bmatrix} \tag{11.6}$$

where $\mathbf{g}_{i,j}$ is a binary row vector with N entries describing the connections from the ith input, $i = 1, \ldots, k_0$, to the jth output, $j = 1, \ldots, n_0$. Vectors $\mathbf{g}_{i,j}$ are often called *generators* of the encoder.

Example 11.3 Let us reconsider the code of Example 11.1. This code has $k_0 = 1$ and $n_0 = 3$. Therefore, it can be described with the following three generators:

$$\begin{aligned}\mathbf{g}_{1,1} &= (100) \\ \mathbf{g}_{1,2} &= (110) \\ \mathbf{g}_{1,3} &= (111)\end{aligned}$$

In the literature, the binary vectors $\mathbf{g}_{i,j}$ are also represented as octal numbers ($110 \rightarrow 6$), or polynomials in the indeterminate Z, as was done for cyclic codes. As an example, the previous vector $\mathbf{g}_{1,3} = (110)$ would be represented as $g_{1,3}(Z) = Z^2 + Z$. The tables describing the "best" convolutional codes (see Section 11.1.2) shall characterize the codes using the transfer function matrix, in which each generator will be represented as an octal number.

The advantage of this representation is not immediately apparent. As in Example 11.1, we have three vectors. But, for example, in the case of a (3,1,10) code, this second representation always requires three generators of length 10, whereas in the other representation we would need 10 vectors (the submatrices $\mathbf{G}_i$) of length 3. No doubt the first description is more practical. □

11.1.1. State diagram representation of convolutional codes

As already noted, there is a powerful and practical alternative to the algebraic description of convolutional codes. This alternative is based on the observation that the convolutional encoder is a finite-memory system, and hence its output sequence depends on the input sequence and on the state of the device. The description we are looking for is called the *state diagram* of the convolutional encoder.

We shall illustrate the concepts involved in this description by taking as an example the encoder of Fig. 11.2. This encoder refers to the (3,1,3) code described in Example 11.1. Notice that each output triplet of digits depends on the input digit and on the content of the shift register that stores the oldest two input digits. The encoder has memory $\nu = N - 1 = 2$. Let us define the state σ_ℓ of the encoder at discrete time ℓ as the content of its memory at the same time. That is,

$$\sigma_\ell \triangleq (u_{\ell-1}, u_{\ell-2}) \tag{11.7}$$

There are $N_\sigma = 2^\nu = 4$ possible states. That is, 00, 01, 10, and 11. Looking at Fig. 11.2, assume, for example, that the encoder is in state 10. When the input digit is 1, the encoder produces the output digits 100 and moves to the state 11.

This type of behavior is completely described by the state diagram of Fig. 11.6. Each of the four states is represented in a circle. A solid edge represents a transition between two states forced by the input digit "0," whereas a dashed edge represents a transition forced by the input digit "1." The label on each edge represents the output digits corresponding to that transition. Using the state diagram of Fig. 11.6, the computation of the encoded sequence is quite straightforward. Starting from the initial state 00, we jump from one state to the next following a solid edge when the input is 0 or a dashed edge when the input is 1.

If we define the states to be $S_1 = (00), S_2 = (01), S_3 = (10), S_4 = (11)$, we can easily check that the input sequence $\mathbf{u} = (11011\ldots)$, already considered in Example 11.1, assuming S_1 as the initial state, corresponds to the *path* $S_1S_3S_4S_2S_3S_4\ldots$ through the state diagram, and the output sequence is $\mathbf{x} = (111\ 100\ 010\ 110\ 100\ \ldots)$, as found by writing down the sequence of edge labels.

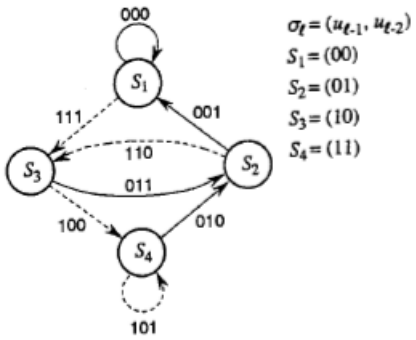


Figure 11.6: *State diagram for the (3,1,3) convolutional code of Example 11.1.*

The concept of state diagram can be applied to any (n_0, k_0, N) code with memory ν. The number of states is $N_\sigma = 2^\nu$. There are 2^{k_0} edges entering each state and 2^{k_0} edges leaving each state. The labels on each edge are sequences of length n_0. As ν increases, the size of the state diagram grows exponentially and becomes very hard to handle. As we are "walking inside" the state diagram following the guidance of the input sequence, it soon becomes difficult to keep track of the past path, because we travel along the same edges many times. Therefore, it is desirable to modify the concept of state diagram by introducing time explicitly. This result is achieved if we replicate the states at each time step, as shown in the diagram of Fig. 11.7. This is called a *trellis diagram*. It refers to the state diagram of Fig. 11.6. In this trellis, the four nodes on the same vertical represent the four states at the same discrete time ℓ, which is called the *depth* into the trellis. Dashed and solid edges have the same meaning as in the state diagram. The input sequence is now represented by the path $\sigma_0 = S_1, \sigma_1 = S_3, \sigma_2 = S_4, \ldots$, and so on. Any encoder output sequence can be found by walking through the appropriate path into the trellis.

Finally, a different representation of the code can be given by expanding the trellis diagram of Fig. 11.7 into the *tree diagram* of Fig. 11.8. In this diagram, the encoding process can be conceived as a walk through a *binary tree*. Each encoded sequence is represented by one particular path into the tree. The encoding process is guided by binary decisions (the input digit) at each *node* of the tree. This tree has an exponential growth. At depth ℓ, there will be 2^ℓ possible paths representing all the possible encoded sequences of that length. The path corresponding to the input sequence 11011 is shown as an example in Fig. 11.8. The nodes of the tree are labeled with reference to the states of the state diagram

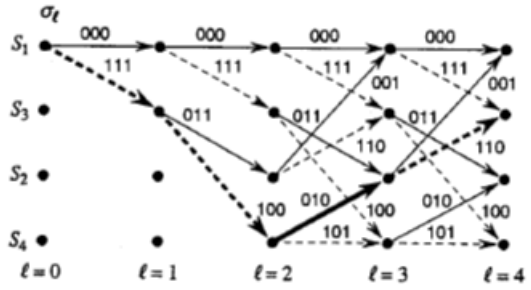


Figure 11.7: *Trellis diagram for the (3,1,3) convolutional code of Example 11.1. The boldface path corresponds to the input sequence 1101.*

shown in Fig. 11.6.

Distance properties and transfer functions of convolutional codes

As for block codes, the error-detection and error-correction capabilities of a convolutional code are directly related to the distance properties of the encoded sequences. Due to the uniform error property of linear codes, we assume that the all-zero sequence is transmitted in order to determine the performance of the convolutional code.

Let us start with some definitions. Consider a pair of encoded sequences up to the depth ℓ into the code trellis and assume that they disagree at the first branch. We define the ℓ-th order *column distance* $d_c(\ell)$ as the minimum Hamming distance between all pairs of such sequences. For the computation of $d_c(\ell)$, one of the sequences of the pair can be the all-zero sequence. Therefore, we have to consider all sequences, up to the depth ℓ in the code trellis, such that they disagree at the first branch from the all-zero sequence. The column distance $d_c(\ell)$ is the minimum weight of this set of code sequences. The column distance $d_c(\ell)$ is a nondecreasing function of the depth ℓ. By letting the value of ℓ go to infinity, we obtain the so-called *free distance* d_f of the convolutional code, defined as

$$d_f \triangleq \lim_{\ell \to \infty} d_c(\ell) \tag{11.8}$$

From (11.8), we see that the free distance of the code is the minimum Hamming distance between infinitely long encoded sequences.

It can be found on the code trellis by looking for those sequences (paths) that, after diverging from the all-zero sequence, merge again into it. The free distance is the minimum weight of this set of encoded sequences.

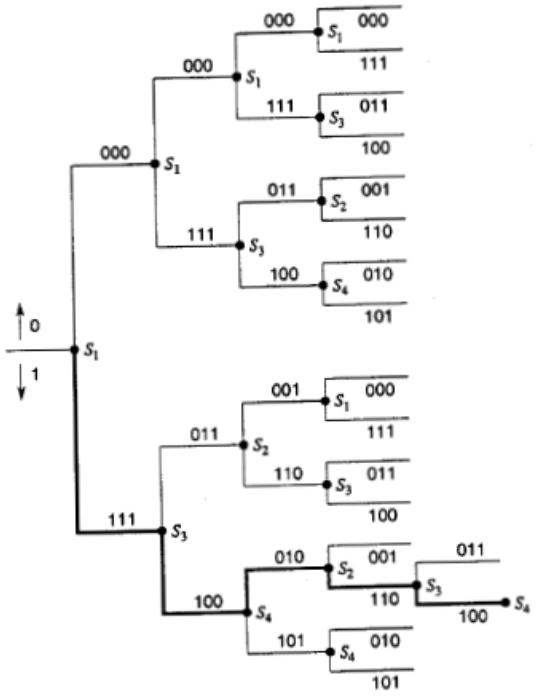


Figure 11.8: *Tree diagram for the (3,1,3) convolutional code of Example 11.1. The solid path corresponds to the input sequence 11011.*

A straightforward algorithm to compute d_f is based on the following steps:

1. Set $\ell = 0$
2. $\ell \rightarrow \ell + 1$
3. Compute $d_c(\ell)$
4. If the sequence giving $d_c(\ell)$ merges into the all-zero sequence, keep its weight as d_f and go to 6.
5. Return to 2
6. Stop.

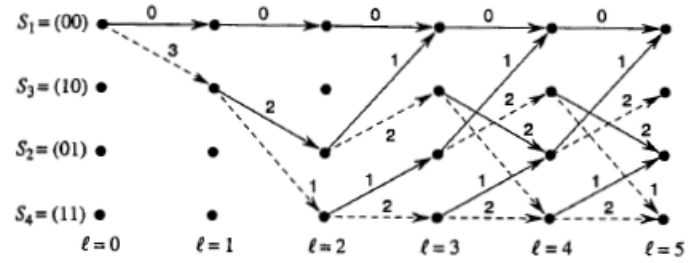


Figure 11.9: *Part of the trellis diagram for the (3,1,3) code of Example 11.4 for the computation of the code distance properties. The trellis is the same as in Fig. 11.7, but the labels represent here the weight of the output sequence of three digits associated with each edge.*

Example 11.4 We want to reconsider the (3,1,3) convolutional code, whose trellis is given in Fig. 11.7, to find the distances just defined. Let us consider Fig. 11.9. Part of the trellis is reproduced in the figure, with the following features. Only the all-zero sequence and the sequences diverging from it at the first branch are reproduced. Furthermore, each edge is labeled with the weight of the encoded sequence. The column distance of the code can be found by inspection. We get

ℓ	$d_c(\ell)$
1	3
2	4
3	5
4	$6 \leftarrow d_f$

Since the constraint length of this code is $N = 3$, we have the first merge of one sequence into the all-zero sequence for $\ell = 3$. However, the merging sequence has weight 6, and does not give $d_c(3)$, which is instead equal to 5. Thus, we must keep looking for d_f. For $\ell = 4$, we have a merging sequence giving $d_c(4)$. Its weight is 6, and therefore we conclude that $d_f = 6$. □

The computation of d_f, although straightforward, may require the examination of exceedingly long sequences. In practice, the problem is amenable to an algorithmic solution based on the state diagram of the code. We take again the case of Fig. 11.6 as a guiding example. The state diagram is redrawn in Fig. 11.10 with certain modifications made in view of our goal. First, the edges are labeled with an indeterminate D raised to an exponent that represents the weight

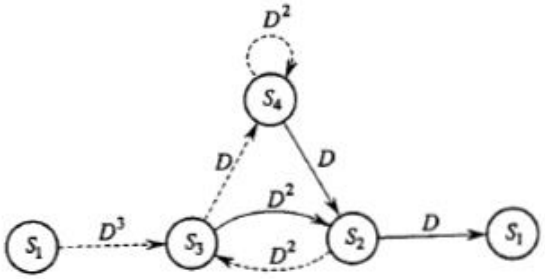


Figure 11.10: *State diagram for the (3,1,3) convolutional code of Fig. 11.6. The labels allow the computation of the weight enumerating function* $T(D)$.

(or, equivalently, the Hamming distance from the all-zero sequence) of the encoded sequence corresponding to that state transition. Furthermore, the self-loop at state S_1 has been eliminated, since it does not contribute to the weight of a sequence. Finally, the state S_1 has been split into two states, one of which represents the input and the other the output of the state diagram.

Let us now define the label of a path as the product of the labels of all its edges. Therefore, among all the infinitely many paths starting in S_1 and merging again into S_1, we are looking for the path whose label D is raised to the smallest exponent. This exponent is indeed d_f. By inspection of Fig. 11.10, we can verify that the path $S_1S_3S_2S_1$ (see Example 11.4) has label D^6, and indeed this code has $d_f = 6$. We can define a *weight enumerating function* [2] $T(D)$ of the output sequence weights as a series that gives all the information about the weights of the paths starting from S_1 and merging again into S_1. This weight enumerating function can be computed as the transfer function of the signal-flow graph of Fig. 11.10. Using standard techniques for the study of directed graphs (see Appendix D), the transfer function for the graph of Fig. 11.10 is given by

$$T(D) = \frac{2D^6 - D^8}{1 - (D^2 + 2D^4 - D^6)} = 2D^6 + D^8 + 5D^{10} + \ldots \triangleq \sum_{d=d_f}^{\infty} A_d D^d \quad (11.9)$$

where A_d is the number (multiplicity) of paths with weight d diverging from state S_1 and remerging into it later. Thus, we deduce from (11.9) that there are

[2] The function $T(D)$ is more often called the *generating* or *transfer function* of the convolutional code, and, sometimes, we will use this denomination, too. The term "weight enumerating function," however, is more appropriate, because, apart from the length of the described code words, which can be infinite for convolutional codes, its meaning is the same as for the function $A(D)$ of block codes defined in Eq. (10.18). The only difference is that $T(D)$ does not contain the all-zero sequence, so in its development as a power series the "1" is missing.

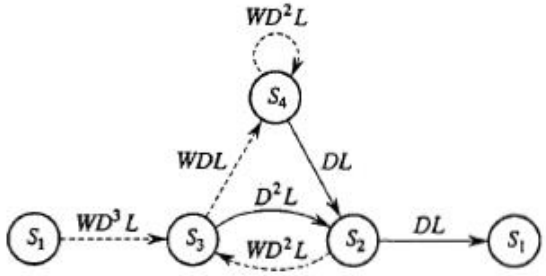


Figure 11.11: *State diagram for the (3,1,3) code of Fig. 11.6. The labels allow the computation of the input-output weight enumerating function* $T_3(W, D, L)$.

two paths of weight 6, one path of weight 8, five paths of weight 10, and so on. Using the terminology of Chapter 4, we can also say that the all-zero path has two *nearest neighbors* at Hamming (instead of Euclidean) distance 6.

Different forms of transfer functions can be used to provide additional information on the code properties. This is done by considering the modified graph of Fig. 11.11. Each edge has now a label containing three indeterminates, W, D, L. The exponent of W is the weight of the input data frame (a single digit, in this example) that caused the transition, so that the exponent of W for a given path will represent the Hamming weight of the information sequence that generated it. The indeterminate D has the same meaning as before, and finally, L is present in each edge, so that its exponent will count the length of the paths. According to the expanded labels, we have a new *input-output weight enumerating function* denoted by $T_3(W, D, L)$, where the subscript refers to the number of indeterminates.

For the state diagram of Fig. 11.11 we obtain

$$\begin{aligned} T_3(W, D, L) &= \frac{WD^6L^3(1 + WL - WD^2L)}{1 - WD^2L(1 + D^2L + WD^2L^2 - WD^4L^2)} \\ &= WD^6L^3(1 + WL) + W^3D^8L^5 + \ldots \\ &\triangleq \sum_{w=1}^{\infty}\sum_{d=d_f}^{\infty}\sum_{\ell=1}^{\infty} C_{w,d,\ell} W^w D^d L^\ell \end{aligned} \quad (11.10)$$

where $C_{w,d,\ell}$ is the number of paths diverging from state S_1 and remerging into it later generated by an information sequence of weight w, having weight d, and with length ℓ. From (11.10) we see that the two paths of weight 6 have lengths 3 and 4, respectively, and that the weights of the input sequences are 1 and 2. The path of weight 8 has length 5, and the corresponding input sequence has weight 3. And so on. These numbers can be checked immediately in Example 11.4.

Comparing (11.9) with (11.10), we realize that $T(D)$ can be obtained from $T_3(W, D, L)$ by setting $W = L = 1$.

Sometimes, the length of the path is not important. In this case, we have a third version of the input-output weight enumerating function, $T_2(W, D)$, which contains only two indeterminates. With obvious notations, it is defined as

$$T_2(W, D) \triangleq \sum_{w=1}^{\infty} \sum_{d=d_f}^{\infty} B_{w,d} W^w D^d \tag{11.11}$$

where $B_{w,d}$ is the number of paths diverging from state S_1 and remerging into it later with weight d, generated by an information sequence of weight w. $T_2(W, D)$ can be obtained from $T_3(W, D, L)$ by setting $L = 1$.

We have thus three distinct weight generating functions. The first, $T(D)$, characterizes the *distance spectrum* of the convolutional code through the pairs (A_d, d) yielding the weights d of the code sequences and their multiplicities A_d. The second, the input-output weight enumerating function $T_2(W, D)$, provides information on the encoder mapping between input and code sequences, by keeping distinct code sequences of the same weight generated by input sequences of different weights. The third, $T_3(W, D, L)$, finally, adds to T_2 the information about the length of sequences in terms of number of trellis branches.

The multiplicities $A_d, B_{w,d}, C_{w,d,\ell}$ satisfy the following relationships:

$$A_d = \sum_{w=1}^{\infty} B_{w,d} = \sum_{w=1}^{\infty} \sum_{\ell=1}^{\infty} C_{w,d,\ell}\,, \qquad B_{w,d} = \sum_{\ell=1}^{\infty} C_{w,d,\ell} \tag{11.12}$$

We have determined the properties of all code paths with reference to a simple convolutional code. The same techniques can be applied to any code of arbitrary rate and constraint length. We shall see in the next sections how the weight enumerating functions of the code can be used to bound the error probabilities of convolutional codes.

11.1.2. Best known short-constraint-length convolutional codes

When considering the weight enumerating function $T(D)$ of a convolutional code, it was implicitly assumed that $T(D)$ converges. Otherwise, the expansions of (11.9) and (11.10) are not valid. This convergence cannot occur for all values of the indeterminate, because the coefficients are nonnegative. In some cases, certain coefficients are infinite, and the code is called *catastrophic*. An example is given in Problem 11.4. The code is a (2,1,3) code. Its state diagram shows that the self-loop at state S_4 does not increase the distance from the all-zero sequence, i.e., its label has an exponent of D equal to zero. Therefore, the path $S_1S_3S_4 \ldots S_4S_2S_1$ will be at distance 6 from the all-zero path no matter how

	d_f (Rate 1/2)		d_f (Rate 1/3)	
ν	Systematic	Nonsystematic	Systematic	Nonsystematic
1	3	3	5	5
2	4	5	6	8
3	4	6	8	10
4	5	7	9	12
5	6	8	10	13
6	6	10	12	15
7	7	10	12	16

Table 11.1: *Maximum free distances achievable with systematic codes and nonsystematic noncatastrophic codes with memory ν and rates 1/2 and 1/3.*

many times it circulates in the self-loop at state S_4. We have the unfortunate circumstance where a finite-weight code sequence corresponds to an infinite-weight information sequence. Thus, it is possible to have an arbitrarily large number of decoding errors even for a fixed finite number of channel errors. This explains the name given to these codes.

The presence in the trellis of a self-loop, different from the one in state S_1, with zero weight associated, is a sufficient condition for the code to be catastrophic. We may have, however, closed loops (i.e., paths from state S_i to state S_i) in the state diagram longer than one trellis branch, and with overall zero weight. In this case, too, the code is catastrophic.

Conditions can be established on the code generators that form the transfer function matrix (11.6) of the code to avoid catastrophic codes. For rate $1/n_0$ codes, the condition is particularly simple, and states that the code generators, in polynomial form, must be relatively prime to avoid catastrophicity (see Problem 11.5). The general conditions can be found in Massey and Sain (1968).

An important consideration here is that systematic convolutional codes cannot be catastrophic. Unfortunately, however, the free distances that are achievable by systematic codes realized with the feed-forward encoder[3] of Fig. 11.1 are usually lower than for nonsystematic codes of the same constraint length N. Table 11.1 shows the maximum free distances achievable with systematic (generated by feed-forward encoders) and nonsystematic noncatastrophic codes of rates 1/2 and 1/3 for increasing values of the code memory ν.

Computer search methods have been used to find convolutional codes opti-

[3] We insist on the role of the encoder structure, since in Section 11.1.6 we will show that every nonsystematic convolutional encoder admits an equivalent systematic encoder, provided that the encoder is not constrained to be feed-forward.

mum in the sense that, for a given rate and a given constraint length, they have the largest possible free distance. These results were obtained by Odenwalder (1970), Larsen (1973), Paaske (1974), Daut *et al.* (1982), and recently by Chang *et al.* (1997). While the first searches used as selection criterion the maximization of the free distance, the recent search by Chang *et al.* (1997) is aimed at optimizing the input-output weight-enumerating function previously introduced. This criterion, as we will see in Section 11.1.5, is equivalent to minimizing the upper bounds to bit and error event probabilities. The best codes are reproduced in part in Tables 11.2 through 11.7. For the rates and number of states included in the search by Chang *et al.* (1997), the tables reproduce those codes since they have been found using the more complete optimization criterion. The codes are identified by their transfer function matrix defined in (11.6), in which the generators are represented as octal numbers. So, for example, an (n_0, k_0) code will be represented by $k_0 \times n_0$ octal numbers organized in a matrix with k_0 rows and n_0 columns. The tables also give, when available, upper bounds on d_f derived in Heller (1968) for codes of rate $1/n_0$ and extended to codes of rate k_0/n_0 by Daut, Modestino, and Wismer (1982). The Heller bound is described later in this chapter.

Example 11.5 The rate 1/2 convolutional code of memory $\nu = 3$ of Table 11.2 has generators 15 and 17, which means

$$\begin{aligned} g_{1,1} &= (1101) \\ g_{1,2} &= (1111) \end{aligned}$$

The block diagram of the encoder is shown in Fig. 11.12. For the rate 2/3 code of memory $\nu = 3$ in Table 11.6, the transfer function matrix is

$$\mathbf{G} = \begin{bmatrix} 3 & 2 & 1 \\ 4 & 2 & 7 \end{bmatrix}$$

and the block diagram of the encoder is shown in Fig. 11.13. □

Punctured convolutional codes

An appropriate measure of the maximum-likelihood decoder complexity for a convolutional code (see next section) is the number of visited edges per decoded bit. Now, a rate k_0/n_0 code has 2^{k_0} edges leaving and entering each trellis state and a number of states $N_\sigma = 2^\nu$, where ν is the memory of the encoder. Thus, each trellis section, corresponding to k_0 input bits, has a total number of edges

Memory ν	Generators in octal notation		d_f	Upper bound on d_f
1	1	3	3	3
2	5	7	5	5
3	15	17	6	6
4	23	35	7	8
5	53	75	8	8
6	133	171	10	10
7	247	371	10	11
8	561	753	12	12
9	1131	1537	12	13
10	2473	3217	14	14
11	4325	6747	15	15
12	10627	16765	16	16
13	27251	37363	16	17

Table 11.2: *Feed-forward nonsystematic encoders generating maximum free distance convolutional codes of rate 1/2 and memory ν. (Chang* et al., *1997).*

Memory ν	Generators in octal notation			d_f	Upper bound on d_f
1	1	3	3	5	5
2	5	7	7	8	8
3	13	15	17	10	10
4	25	33	37	12	12
5	47	53	75	13	13
6	117	127	155	15	15
7	225	331	367	16	16
8	575	623	727	18	18
9	1167	1375	1545	20	20
10	2325	2731	3747	22	22
11	5745	6471	7553	24	24
12	10533	10675	17661	24	24
13	21645	35661	37133	26	26

Table 11.3: *Feed-forward nonsystematic encoders generating maximum free distance convolutional codes of rate 1/3 and memory ν. (Larsen, 1973, and Chang* et al., *1997).*

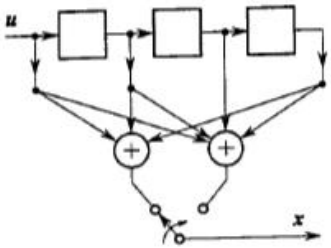


Figure 11.12: *Encoder for the (2,1,4) convolutional code of Example 11.5.*

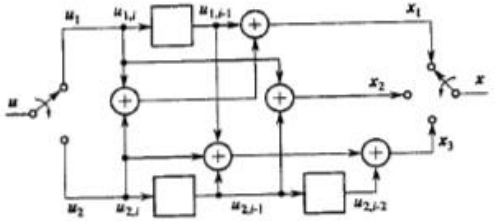


Figure 11.13: *Encoder for the (3,2,2) convolutional code of Example 11.5.*

equal to $2^{k_0+\nu}$. As a consequence, an (n_0, k_0, N) code has a decoding complexity

$$\mathcal{D} = \frac{2^{k_0+\nu}}{k_0} \tag{11.13}$$

The increase of complexity inherent in passing from rate $1/n_0$ to rate k_0/n_0 codes can be mitigated using the so-called *punctured* convolutional codes. A rate k_0/n_0 punctured convolutional code can be obtained by starting from a rate $1/n_0$ and deleting parity-check symbols. An example will clarify the concept.

Example 11.6 Consider the 4-state convolutional encoder of Fig. 11.14 (a). For each input bit entering the encoder, two bits are sent through the channel, so that the code generated has rate 1/2. Its trellis is also shown in Fig. 11.14 (b). Suppose now that for every four parity-check digits generated by the encoder, one (the last) is punctured, i.e., not transmitted. In this case, for every two input bits three bits are generated by the encoder, thus producing a rate 2/3 code. The trellis for the punctured code is shown in Fig. 11.14 (c), and the letter "x" denotes a punctured output bit. As an example, the input sequence $\mathbf{u} = 101101\ldots$ would yield $\mathbf{x} = 111000010100$ for the rate 1/2 code, and $\mathbf{x} = 111000010$ for the punctured rate 2/3 code. In a similar way, higher rates can be obtained by increasing the number of punctured parity-check bits.

Memory ν	Generators in octal notation				d_f	Upper bound on d_f
1	1	1	3	3	6	6
2	5	5	7	7	10	10
3	13	13	15	17	13	15
4	25	27	33	37	16	16
5	45	53	67	77	18	18
6	117	127	155	171	20	20
7	257	311	337	355	22	22
8	533	575	647	711	24	24
9	1173	1325	1467	1751	27	27
10	2387	2353	2671	3175	29	29
11	4767	5723	6265	7455	32	32
12	11145	12477	15537	16727	33	33
13	21113	23175	35527	35537	36	36

Table 11.4: *Feed-forward nonsystematic encoders generating maximum free distance convolutional codes of rate 1/4 and memory ν. (Larsen, 1973, and Chang* et al., *1997).*

Memory ν	Generators in octal notation					d_f	Upper bound on d_f
2	7	7	7	5	5	13	13
3	17	17	13	15	15	16	16
4	37	27	33	25	35	20	20
5	75	71	73	65	57	22	22
6	175	131	135	135	147	25	25
7	257	233	323	271	357	28	28

Table 11.5: *Feed-forward nonsystematic encoders generating maximum free distance convolutional codes of rate 1/5 and memory ν (Modestino and Wismer, 1982).*

It is interesting to note that the punctured rate 2/3 code so obtained is equivalent to the unpunctured rate 2/3 code depicted in Fig. 11.15, for which one stage of the trellis corresponds to two stages of the trellis of the punctured code. □

Of course, the way parity-check digits are deleted, or "punctured," should be optimized in order to maximize the free distance of the code (see Problem 11.6). Tables of optimum punctured codes can be found in Cain *et al.* (1979) and Ya-

Constraint length N	Memory ν	Transfer function matrix in octal notation	d_f	Upper bound on d_f
2	2	$\begin{pmatrix} 3 & 1 & 0 \\ 2 & 3 & 3 \end{pmatrix}$	3	4
3	3	$\begin{pmatrix} 3 & 2 & 1 \\ 4 & 2 & 7 \end{pmatrix}$	4	-
3	4	$\begin{pmatrix} 6 & 5 & 1 \\ 7 & 2 & 5 \end{pmatrix}$	5	6
4	5	$\begin{pmatrix} 07 & 06 & 03 \\ 12 & 01 & 13 \end{pmatrix}$	6	-
4	6	$\begin{pmatrix} 06 & 13 & 13 \\ 13 & 06 & 17 \end{pmatrix}$	7	7
5	7	$\begin{pmatrix} 16 & 13 & 03 \\ 25 & 05 & 34 \end{pmatrix}$	8	-
5	8	$\begin{pmatrix} 37 & 31 & 16 \\ 23 & 14 & 35 \end{pmatrix}$	8	-
6	9	$\begin{pmatrix} 27 & 23 & 16 \\ 47 & 17 & 41 \end{pmatrix}$	9	-
6	10	$\begin{pmatrix} 63 & 51 & 34 \\ 52 & 37 & 55 \end{pmatrix}$	10	-

Table 11.6: *Feed-forward nonsystematic encoders generating maximum free distance convolutional codes of rate 2/3 and constraint length N. (Chang* et al., *1997).*

suda *et al.* (1984). They yield rate k_0/n_0 codes from a single rate $1/n_0$ "mother" code.

From the previous example, we can derive the conclusion that a rate k_0/n_0 convolutional code can be obtained considering k_0 trellis sections of a rate 1/2 mother code. Measuring the decoding complexity as done before in (11.13), we obtain for the punctured code

$$\mathcal{D}_{\text{punc}} = \frac{k_0 2^{\nu+1}}{k_o} \tag{11.14}$$

so that the ratio between the case of the unpunctured to the punctured solution yields

$$\frac{\mathcal{D}}{\mathcal{D}_{\text{punc}}} = \frac{2^{k_0}}{2k_o} \tag{11.15}$$

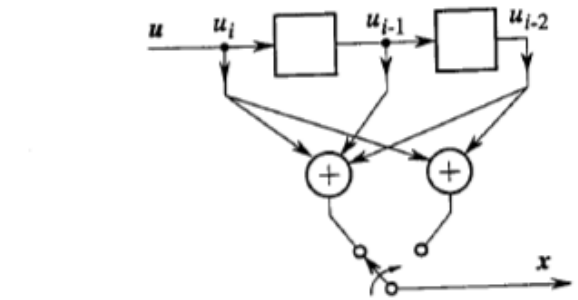
which shows that, for $k_0 > 2$, there is an increasing complexity reduction yielded

Constraint length N	Memory ν	Transfer function matrix in octal notation	d_f	Upper bound on d_f
2	2	$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 3 & 0 & 0 & 1 \\ 3 & 2 & 0 & 2 \end{pmatrix}$	3	-
2	3	$\begin{pmatrix} 3 & 2 & 1 & 0 \\ 3 & 1 & 2 & 1 \\ 2 & 2 & 2 & 3 \end{pmatrix}$	4	4
3	4	$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 4 & 1 & 5 \end{pmatrix}$	4	-
3	5	$\begin{pmatrix} 3 & 3 & 2 & 2 \\ 5 & 2 & 7 & 0 \\ 4 & 7 & 0 & 1 \end{pmatrix}$	5	-
3	6	$\begin{pmatrix} 5 & 4 & 3 & 2 \\ 4 & 6 & 5 & 5 \\ 6 & 1 & 4 & 3 \end{pmatrix}$	6	-
4	7	$\begin{pmatrix} 02 & 03 & 04 & 07 \\ 03 & 07 & 03 & 05 \\ 15 & 02 & 02 & 17 \end{pmatrix}$	6	-
4	8	$\begin{pmatrix} 04 & 06 & 07 & 07 \\ 01 & 12 & 05 & 14 \\ 00 & 07 & 14 & 11 \end{pmatrix}$	7	-
4	9	$\begin{pmatrix} 03 & 06 & 10 & 15 \\ 00 & 16 & 03 & 13 \\ 16 & 05 & 02 & 17 \end{pmatrix}$	8	-

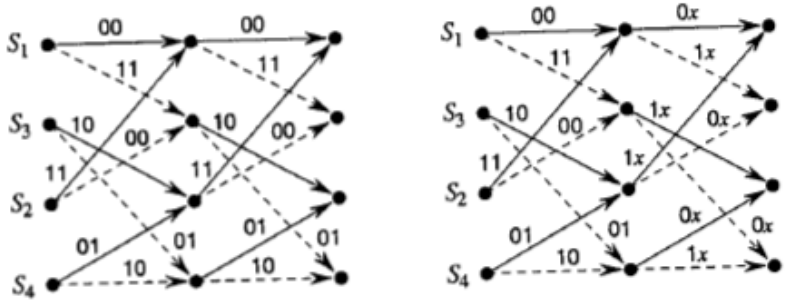
Table 11.7: *Feed-forward nonsystematic encoders generating maximum free distance convolutional codes of rate 3/4 and constraint length N. (Chang* et al., *1997).*

by the punctured solution. Also, with puncturing, one can obtain several rates from the same mother code, thus simplifying the implementation through a sort of "universal" encoder, and this fact is greatly exploited in VLSI implementations.

There are at least two downsides to the punctured solution. First, punctured codes are normally slightly worse in terms of distance spectrum with respect to unpunctured codes of the same rate (see also Problem 11.6). Second, since the trellis of a punctured (n_0, k_0) code is time-varying with period k_0, the decoder needs to acquire frame synchronization.



(a)

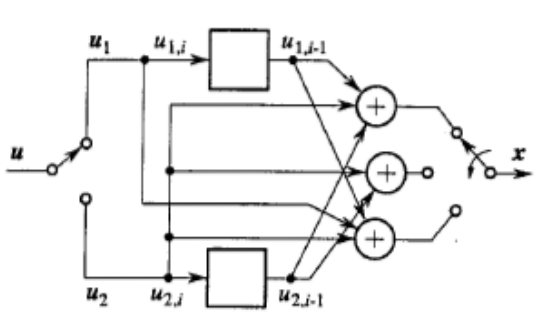


(b) (c)

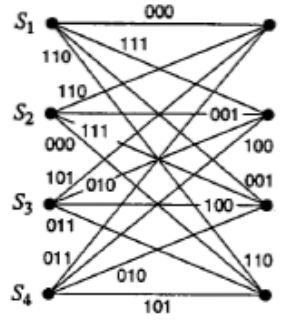
Figure 11.14: *Encoder (a) and trellis (b) for a (2,1,3) convolutional code. The trellis (c) refers to the rate 2/3 punctured code described in Example 11.6.*

11.1.3. Maximum-likelihood decoding of convolutional codes and the Viterbi algorithm

We have already seen that ML decoding of block codes is achieved when the decoder selects the code word whose distance from the received sequence is minimum. In the case of hard decoding, the distance considered is the Hamming distance, while for soft decoding it is the Euclidean distance. Unlike a block code, a convolutional code has no fixed block length. But it is intuitive that the same principle works also for convolutional codes. In fact, each possible encoded sequence is a path into the code trellis. Therefore, the optimum decoder must choose that path into the trellis that is *closest* to the received sequence. Also in this case, the distance measure will be the Hamming distance for hard



(a)



(b)

Figure 11.15: *Encoder (a) and trellis (b) for the (3,2,3) convolutional code equivalent to the rate 2/3 punctured code described in Example 11.6.*

decoding and the Euclidean distance for unquantized soft decoding.

Let us start with hard decoding. We assume binary antipodal modulation, and, consequently, the equivalent discrete channel is a BSC with error probability p. Denoting with $\mathbf{y}$ and $\mathbf{x}^{(r)}$ the received sequence and the rth path in the trellis, respectively, the optimum ML decoder must choose the path $\mathbf{x}^{(r)}$ of the trellis for which the conditional probability $P(\mathbf{y} \mid \mathbf{x}^{(r)})$ is maximum. We may take the logarithm of this probability as well. Therefore, the ML decoder must find the path corresponding to ,

$$\begin{aligned} U(\sigma_{K-1}) &\triangleq \max_r U^{(r)}(\sigma_{K-1}) \triangleq \max_r P(\mathbf{y} \mid \mathbf{x}^{(r)}) \\ &\equiv \max_r \left[\ln \prod_{\ell=0}^{K-1} P(\mathbf{y}_\ell \mid \mathbf{x}_\ell^{(r)}) \right] = \max_r \left[\sum_{\ell=0}^{K-1} \ln P(\mathbf{y}_\ell \mid \mathbf{x}_\ell^{(r)}) \right] \end{aligned} \tag{11.16}$$

where the symbol "$\equiv$" means "equivalent." In (11.16), K indicates the length of the path into the trellis, or, equivalently, Kn_0 is the length of the binary received sequence, $\mathbf{y}_\ell$ is the sequence of n_0 binary digits supplied to the decoder by the demodulator between discrete times ℓ and $(\ell+1)$, and $\mathbf{x}_\ell^{(r)}$ is the n_0-digit label of the r-th path in the code trellis between states σ_ℓ and $\sigma_{\ell+1}$.

The maximization of the RHS of (11.16) is already formulated in terms suitable for the application of the Viterbi algorithm and, henceforth, it is assumed that the reader is familiar with the contents of Appendix F. The metric for each branch of the code trellis is defined as

$$V_\ell^{(r)}(\sigma_{\ell-1}, \sigma_\ell) \triangleq \ln P(\mathbf{y}_\ell \mid \mathbf{x}_\ell^{(r)}) \tag{11.17}$$

and therefore

$$U(\sigma_{K-1}) = \max_r \sum_{\ell=0}^{K-1} V_\ell^{(r)}(\sigma_{\ell-1}, \sigma_\ell) \tag{11.18}$$

If we denote with $d_\ell^{(r)}$ the Hamming distance between the two sequences $\mathbf{y}_\ell$ and $\mathbf{x}_\ell^{(r)}$, and use (10.15), we can rewrite (11.17) as

$$V_\ell^{(r)}(\sigma_{\ell-1}, \sigma_\ell) = -d_\ell^{(r)} \ln \frac{1-p}{p} + n_0 \ln(1-p) = -\alpha d_\ell^{(r)} - \beta \tag{11.19}$$

with α and β positive constants (if $p < 0.5$).

Using (11.19) into (11.18), and dropping unessential constants, the problem is reduced to finding

$$U'(\sigma_{K-1}) \triangleq \min_r \sum_{\ell=0}^{K-1} d_\ell^{(r)} \tag{11.20}$$

As expected intuitively, (11.20) states that ML decoding requires the minimization of the Hamming distance between the received sequence and the path chosen into the code trellis. This conclusion is perfectly consistent with the ML decoding of block codes, provided that the infinitely-long sequences are replaced by n-bit code words. The form of (11.20) is such that the minimization can be accomplished with the Viterbi algorithm (described in Appendix F), the metric on each branch being the Hamming distance between binary sequences.

Example 11.7 We apply the Viterbi decoding algorithm to the code whose trellis is shown in Fig. 11.7, corresponding to the state diagram of Fig. 11.6. We know already (see Example 11.4) that this code has $d_f = 6$. Assume that the transmitted information sequence is 01000000..., whose corresponding encoded sequence is 000 111 011 001 000 000 000 000.... Furthermore, assume that the received sequence is instead 110 111 011 001 000 000 000 000.... It contains two errors in the first triplet of digits, and therefore it does not correspond to any path through the trellis. To apply the Viterbi algorithm, it is more useful to refer to a trellis similar to that of Fig. 11.9, in which, now, the label of each edge corresponds to the Hamming distance between the three digits associated to that edge and the corresponding three received bits. The successive steps of the Viterbi algorithm are shown in Fig. 11.16 and Fig. 11.17. The algorithm, at each step ℓ into the trellis, stores for each state the *surviving* path (the minimum distance path from the starting state ($\sigma_0 = S_1$)) and the corresponding accumulated metric. Consider, for example, the situation at step $\ell = 4$. We have

State σ_4	Surviving path	Metric
S_1	$S_1S_1S_3S_2S_1$	2
S_3	$S_1S_3S_2S_1S_3$	5
S_2	$S_1S_3S_2S_3S_2$	7
S_4	$S_1S_3S_4S_4S_4$	6

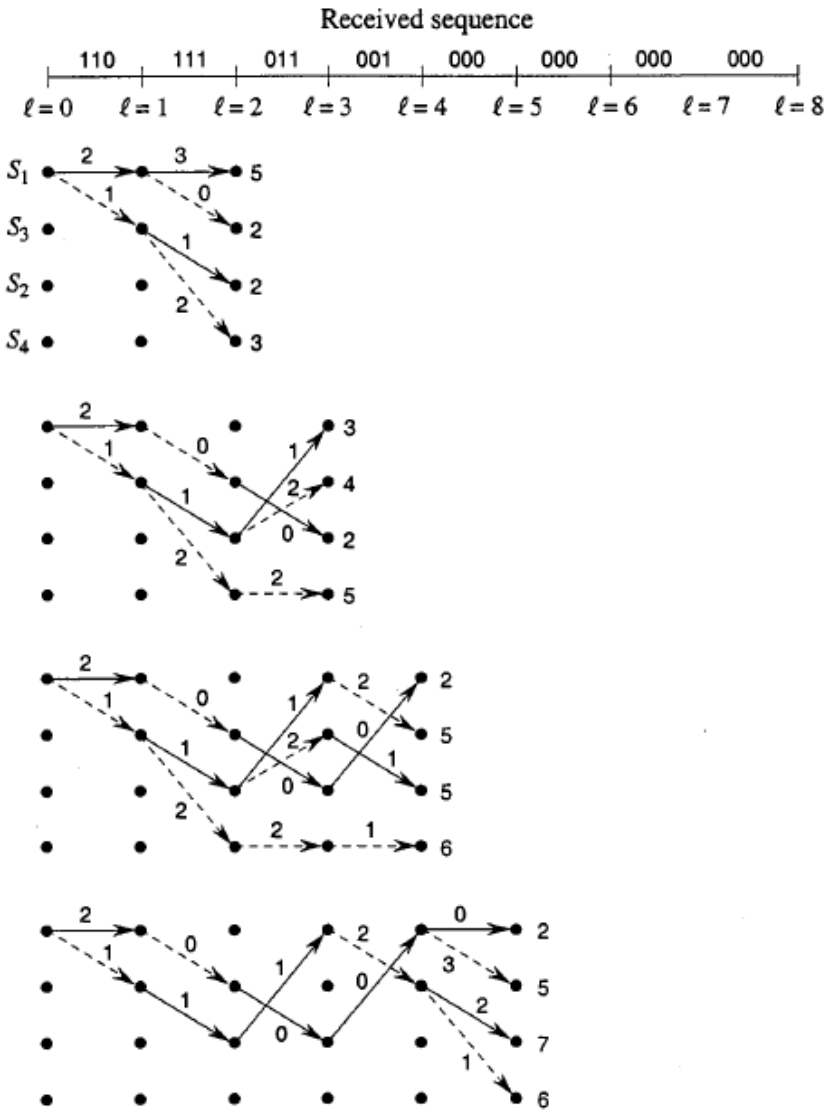


Figure 11.16: *Viterbi decoding algorithm applied to the (3,1,3) convolutional code of Fig. 11.6. The decoded sequence is 01000000.*

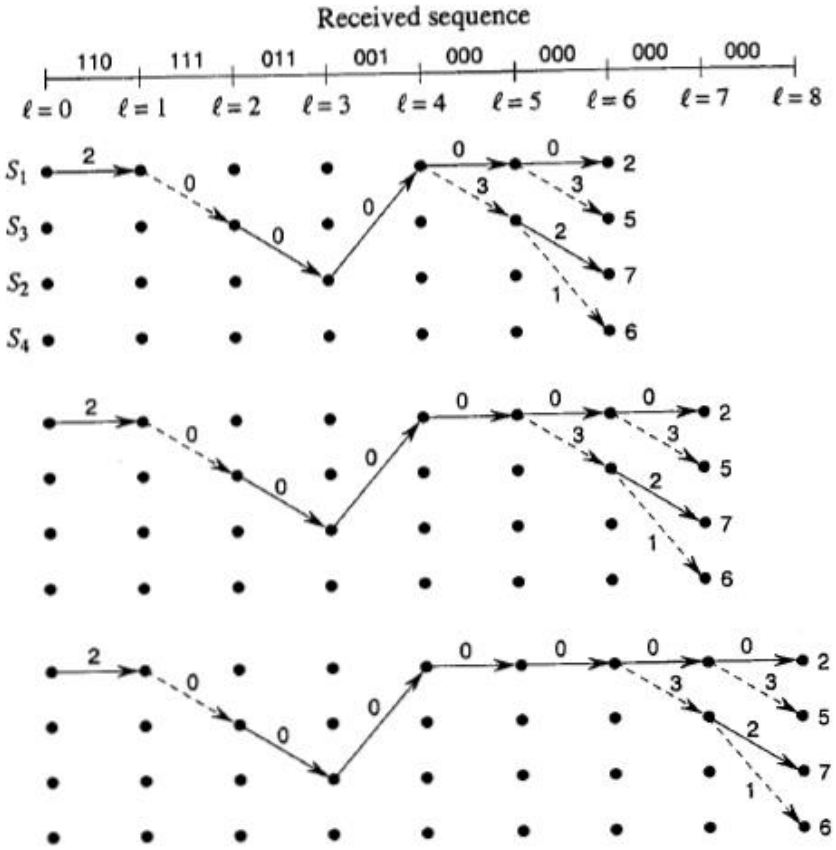


Figure 11.17: *Continuation of Fig. 11.16: Viterbi decoding algorithm applied to the (3,1,3) convolutional code of Fig. 11.6. The decoded sequence is 01000000.*

Therefore, at step $\ell = 4$, the ML path (the one with the smallest distance) is $S_1S_1S_3S_2S_1$, and the corresponding information sequence is 0100 (follow the dashed and solid edges on the trellis). Consequently, in spite of the two initial channel errors, already at this step the correct information sequence is identified. In case of a tie, i.e., when two or more states exhibit the same lowest path metric, any of the corresponding paths can be chosen.

Let us see in detail how the algorithm proceeds one step farther to compute the situation at $\ell = 5$. Consider first the state $\sigma_5 = S_1$. From the trellis diagram of Fig. 11.7, we can verify that the state S_1 can be reached either from the state S_1 with a transition corresponding to an encoded triplet 000, or from the state S_2 with a transition corresponding to an encoded triplet 001. The received triplet during this transition is 000. Therefore, from (11.19) and (11.18) we have, as potential candidates to $U'(\sigma_5)$

$$U'(\sigma_5)|_{\sigma_5=S_1} = U'(\sigma_4)|_{\sigma_4=S_1} + V_5^{(1)}(S_1, S_1) = 2 + 0 = 2$$
$$U'(\sigma_5)|_{\sigma_5=S_1} = U'(\sigma_4)|_{\sigma_4=S_2} + V_5^{(1)}(S_2, S_1) = 5 + 1 = 6$$

Thus, the minimum-distance path leading to S_1 at $\ell = 5$ comes from S_1, and the transition from S_2 is dropped. The metric $U'(\sigma_5)$ at S_1 will be 2, and the surviving path will be that of $\sigma_4 = S_1$ (i.e., 0100) with a new 0 added (i.e., 01000). The interesting feature is that at $\ell = 6$ all surviving paths *merge* at state $\sigma_4 = S_1$. This means that at this step the first four information digits are uniquely decoded in the correct way and the two channel errors are corrected. □

For a general (n_0, k_0, N) convolutional code, there are 2^ν states at each step in the trellis. Consequently, the Viterbi decoding algorithm requires the storage of 2^ν surviving paths and 2^ν metrics. At each step, there are 2^{k_0} paths reaching each state, and therefore 2^{k_0} metrics must be computed for each state. Only one of the 2^{k_0} paths reaching each state does survive, and this is the minimum-distance path from the received sequence up to that transition. The complexity of the Viterbi decoder, measured in terms of number of visited trellis edges per decoded bit, is then

$$\mathcal{D} = \frac{2^{k_0+\nu}}{k_0} \tag{11.21}$$

and grows exponentially[4] with k_0 and ν. For this reason, practical applications are confined to the cases for which $k_0 + \nu$ is in the range 2 to 15.[5] The Viterbi algorithm is basically simple, and has properties that yield easy VLSI implementations. Actually, Viterbi decoding has been widely applied and is presently one of the most practical techniques for providing large coding gains.

The trellis structure of the decoding process has the following consequence. If at some point an incorrect path is chosen, it is highly probable that it will merge with the correct path at a later time. Therefore, the typical error sequences of convolutional codes, when decoded by a Viterbi decoder, result in bursts of errors due to the incorrect path diverging from the correct one and soon merging again into it. Typical bursts have a length of a few constraint lengths.

One final consideration concerns the technique used to output the decoded digits. The optimum procedure would be to decode the sequence only at the end

[4] The exponential growth with k_0 can be avoided using punctured codes, as seen previously.

[5] To our knowledge, the most complex implementation of the Viterbi algorithm concerns a code with $k_0 = 1$ and $\nu = 14$ for deep-space applications (see Dolinar, 1988).

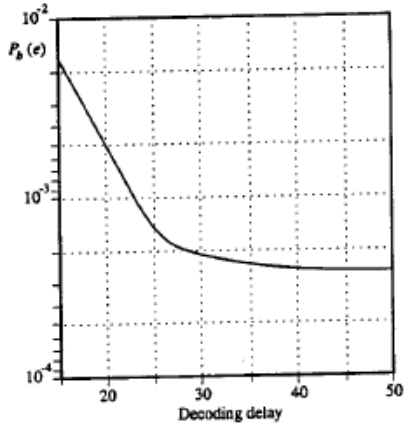


Figure 11.18: *Simulated bit error probability versus the decoding delay for the decoding of the rate 1/2 (2,1,7) convolutional code of Table 11.2. The signal-to-noise ratio $\mathcal{E}/N_0$ is 3 dB.*

of the whole receiving process. However, this would result in unacceptably long decoding delays and excessive memory storage for the surviving sequences. We have seen in the example that all surviving paths tend to merge into one single path when proceeding deeply enough into the trellis. A solution to this problem is thus to use the truncated Viterbi algorithm, described in Appendix F. This forces the decision on the oldest symbol of the minimum distance path after a fixed and sufficiently long delay. Computer simulations show that a delay on the order of $5N$ results in a negligible degradation with respect to the optimum performance. This is shown in Fig. 11.18, where we report the bit error probability evaluated by simulation as a function of the decoding delay for the constraint length 7, rate 1/2 code of Table 11.2.

An important feature of the Viterbi algorithm is that soft-decision decoding, unlike for block codes, requires only a trivial modification of the procedure discussed previously. In fact, it is sufficient to replace the Hamming metric with the Euclidean metric and let all the other decoding operations remain the same. Therefore, the implementation complexity for soft-decision decoding is not significantly different from the hard-decision case.

Let us now derive an expression for the branch metric (11.17) in the case of unquantized soft decisions. In practice, 3-bit quantization of the branch metrics is sufficient to obtain almost ideal performance (Jacobs, 1974).

If $\{y_{j\ell}\}_{j=1}^{n_0}$ is the set of received demodulator outputs in the case of binary

antipodal modulation (with transmitted and received energy $\mathcal{E}$) and assuming that the ℓth branch of the rth path has been transmitted, we have

$$y_{j\ell} = \sqrt{\mathcal{E}}(2x_{j\ell}^{(r)} - 1) + \nu_j \tag{11.22}$$

which is obtained from (10.81). Here, $x_{j\ell}^{(r)}$ is a binary digit and ν_j is a Gaussian RV with zero mean and variance $N_0/2$. Therefore, from (11.22) we get

$$\begin{aligned} P(\mathbf{y}_\ell \mid \mathbf{x}_\ell^{(r)}) &= \prod_{j=1}^{n_0} P(y_{j\ell} \mid x_{j\ell}^{(r)}) \\ &= \prod_{j=1}^{n_0} \frac{1}{\sqrt{\pi N_0}} \exp\left\{ -\frac{\left[y_{j\ell} - \sqrt{\mathcal{E}}(2x_{j\ell}^{(r)} - 1)\right]^2}{N_0} \right\} \end{aligned} \tag{11.23}$$

Inserting (11.23) into (11.17) and neglecting all the terms that are common to all branch metrics, we get

$$V_\ell^{(r)}(\sigma_{\ell-1}, \sigma_\ell) = \sum_{j=1}^{n_0} y_{j\ell}(2x_{j\ell}^{(r)} - 1) \tag{11.24}$$

This is the branch metric to be used by the soft-decision Viterbi decoder. It is called, for obvious reasons, *correlation* metric. The best path would correspond in this case to the highest metric. As an alternative, one can also use the *distance* metric, which should be minimized.

11.1.4. Other decoding techniques for convolutional codes

The computational effort and the storage size required to implement the Viterbi algorithm limit its application to convolutional codes with small-medium values of the memory ν (typically, $1 \leq \nu \leq 14$). Other decoding techniques can be applied to convolutional codes. These techniques preceded the Viterbi algorithm historically, and are quite useful in certain applications. In fact, they can use longer code constraint lengths than those allowed by practical implementations of the Viterbi algorithm, and hence yield larger coding gains.

Sequential decoding techniques

As already pointed out, the operation of a convolutional encoder can be described as the choice of a path through a binary tree in which each path represents an encoded sequence. The sequential decoding techniques (in their several variants) share with the Viterbi algorithm the idea of a probabilistic search of the correct path, but, unlike the Viterbi algorithm, the search does not extend to all paths

that can potentially be the best. Only some subsets of paths that appear to be the most probable ones are extended. For this reason, sequential decoding is not an optimal (ML) algorithm as the Viterbi algorithm. Nevertheless, sequential decoding is one of the most powerful tools for decoding convolutional codes of long constraint length. Its error performance is not significantly worse than that of Viterbi decoding.

The decoding approach can be conceived as a trial-and-error technique for searching out the correct path into the tree. Let us consider a qualitative example by looking at the code tree of Fig. 11.8. In the absence of noise, the code sequences of length $n_0 = 3$ are received without errors. Consequently, the receiver can start its walk into the tree from the root and then follow a path by simply replicating at each node the encoding process and making its binary decision after comparing the locally generated sequence with the received one. The transmitted message will be recovered directly from the path followed into the tree.

The presence of noise introduces errors, and hence the decoder can find itself in a situation in which the decision entails risk. This happens when the received sequence is different from all the possible alternatives that are locally generated by the receiver. Consider again the code tree of Fig. 11.8, and assume that the transmitted sequence is the one denoted by the heavy line. Let, for instance, the received sequence be 111 100 111.... Starting from the root, the first two choices are not ambiguous. But, when reaching the second-order node, the decoder must choose between the upward path (sequence 010) and the downward path (sequence 101), having received the sequence 110. The choice that sounds more reasonable is to go downward in the tree, since the Hamming distance between the received and locally generated sequences is only one, instead of two. With this choice, however, the decoder would proceed on a wrong path in the tree, and the continuation would be in error. If the branch metric is the Hamming distance, the decoder can track the cumulative Hamming distance between the received sequence and the path followed into the tree, and eventually notice that this distance grows higher than expected. In this case, the decoder can decide to go back to the node at which an apparent error was made and try the other choice. This process of going forward and backward into the tree is the rationale behind sequential decoding. This movement can be guided by modifying the metric of the Viterbi algorithm (the Hamming distance for hard decisions) with the addition of a negative constant at each branch. The value of this constant is selected such that the metric for the correct path decreases on the average, while that for any incorrect path increases. By comparing the accumulated metric with a moving threshold, the decoder can detect and discard the incorrect paths.

Sequential algorithms trade with the Viterbi algorithm a larger decoding delay with a smaller storage need. Unlike for the Viterbi algorithm, both the decoding delay and the computational complexity are not constant. Instead, they are random variables that depend, among other factors, on the signal-to-noise ratio. When there is little noise, the decoder is usually following the correct path requiring only one computation to advance one node deeper into the code tree. However, when the noise becomes significant, the metric along the correct path may increase and be higher than the metric along an incorrect path. This forces the decoder to follow an incorrect path, so that a large number of steps (and computations) may be required to return to the correct path. To make this statement quantitative, we refer to the *cutoff rate* of the channel R_0 already introduced in Section 10.4.2. When the code rate R_c is larger than the channel cutoff rate, the average computational load of sequential decoding, defined as the average number of computations per decoded branch, is unbounded. For this reason, R_0 is often called the *computational* cutoff rate, as it indicates a limit on the code rates beyond which sequential decoding becomes impractical (for a proof of these statements, see Lin and Costello, 1983).

The M-algorithm

The idea of the M-algorithm is to look at the best M (M less than the number of trellis states N_σ) paths at each depth of the trellis, and to keep only these paths while proceeding into the trellis (no backtracking allowed). For $M = N_\sigma$, it becomes the Viterbi algorithm. The choice of M trades performance for complexity. Unlike sequential decoding, the M-algorithm has the advantage of fixed complexity and decoding delay. For details on the M-algorithm, see Anderson and Mohan (1991).

Syndrome decoding techniques

Unlike sequential decoding, these techniques are deterministic and rely on the algebraic properties of the code. Typically, a syndrome sequence is calculated (as for block codes). It provides a set of linear equations that can be solved to determine the minimum-weight error sequence. The two most widely used among such techniques are *feedback decoding* (Heller, 1975) and *threshold decoding* (Massey, 1963). They have the advantage of simple circuitry and small decoding delays, thus allowing high-speed decoding. However, since the allowable codes presenting the required algebraic properties are rather poor, only moderate coding gain values are achievable with these techniques.

The maximum-a-posteriori (MAP) symbol decoding algorithm

The Viterbi algorithm performs the ML estimate of the transmitted sequence. Its output is the code sequence closest in some sense (Hamming or Euclidean distance) to the received one. The Viterbi algorithm thus minimizes the *sequence* error probability when the information sequences are assumed to be equally likely.

In the most general case, the decision rule minimizing the *bit* error probability should be based on the maximization of the a posteriori probabilities (APP) of each individual bit in the sequence

$$P(u_k \mid \mathbf{y}), \quad k = 0, \ldots, K-1 \tag{11.25}$$

where u_k is the transmitted bit at time k, $\mathbf{y}$ is the entire received sequence, and K is the sequence length. We recall that MAP decoding also differs from ML decoding in that it does not assume equally likely information symbols.

The simplest algorithm to compute the a-posteriori probabilities (11.25) was proposed by Bahl *et al.* (1974), but until recently it received very little attention because its complexity exceeds that of the Viterbi algorithm, yet the advantage in bit error rate performance is small. It is described in Appendix F under the name of BCJR algorithm, from the initials of the researchers who proposed it.

The big difference between Viterbi and APP algorithms consists in their outputs. The Viterbi algorithm outputs a *hard* decision on the transmitted digits, whereas the APP algorithm provides the a posteriori probability, which may be interpreted as a *soft* estimate of the transmitted digits reliability, actually the best possible one. When a convolutional code is employed in a *concatenated* coding scheme, like those examined in the last section of this chapter, this difference becomes fundamental, and explains the recent great revival of interest for APP algorithms.

As a final, important comment, we can say that the Viterbi algorithm, and consequently also the other algorithms briefly described previously, is applicable to the decoding of *any* code whose code words can be associated to paths in a trellis. As a consequence, the soft decoding of block codes, which can be represented by time-varying trellises (see, for example, Wolf, 1978), can be performed by using the Viterbi algorithm.

11.1.5. Performance evaluation of convolutional codes with ML decoding

In this section, we will derive upper bounds to the *error event* and *bit* error probabilities of a convolutional code. Since the results are based on the union bound, they provide tight approximations to the actual probabilities for medium-high signal-to-noise ratios. For lower signal-to-noise ratios, these bounds diverge,

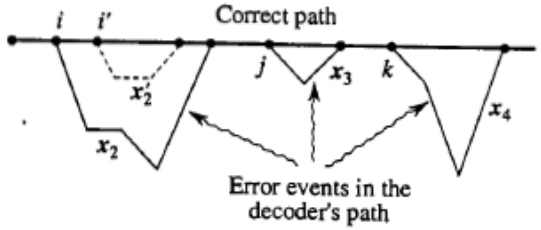


Figure 11.19: *Trellis paths showing possible error events of a Viterbi decoder.*

and one should resort to simulation,[6] or to more sophisticated techniques, like the one described in Poltyrev (1996) and references therein. Moreover, we do not consider the suboptimality of the truncated Viterbi algorithm involved in forcing premature decisions (see Hemmati and Costello, 1977, and Onyszchuk, 1991).

Error event probability

Before discussing techniques for bounding the bit error probability $P_b(e)$, it is necessary to analyze in some detail the concept of *error event* in the Viterbi decoding. Since convolutional codes are linear and the uniform error property holds for them, we can assume that the all-zero sequence has been transmitted and evaluate error probabilities under this hypothesis. We denote as the *correct path* the all-zero horizontal path at the top of the trellis diagram (Fig. 11.19).

The decoder bases its decisions on the received noisy version of the transmitted sequence and can choose a path different from the correct one on the basis of the accumulated metric. For a given discrete time k, an error event is defined as an incorrect path segment that, after diverging from the correct path at trellis depth k, merges again into it at a successive depth. Since the trellis of a convolutional code is time-invariant (with the exception of the initial transient from the zero state to all other states), the performance will not depend on the starting time k of the error event, which can thus be omitted. Fig. 11.19 shows three error events starting at nodes i, j, k, and corresponding to the sequences $\mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4$. Notice also that the dotted path, corresponding to the sequence $\mathbf{x}_2'$ diverging at node i', may have a metric higher than the correct path and yet not be selected, because its accumulated metric is smaller than that of the solid path corresponding to the sequence $\mathbf{x}_2$.

[6] Fortunately, simulation is required for low signal-to-noise ratios, where the error probabilities are high, say greater that 10^{-3}, so that the required computer time is often quite reasonable.

The term "error event" comes from the fact that, when the decoder chooses an incorrect path forming an error event, the errors accumulated during the periods of divergence cannot be corrected, since, after remerging, the correct and error event paths will accumulate the same metrics. We may conclude that a necessary and sufficient condition for an error event to occur at a certain node is that the metric of an incorrect path, diverging from the correct one at that node, accumulates higher metric increments than the correct path over the unmerged path segment.

Thus, we can define an error event at a certain node as the set of all paths diverging from the correct path at that node, and having a path metric larger than the correct one.

If we denote by $\mathbf{x}_d$ a path of weight d diverging from the all-zero path, by $\mathbf{x}_1$ the correct path (all-zero sequence), and by $P(\mathbf{x}_1 \to \mathbf{x}_d)$ the *pairwise* error probability between the two sequences[7] $\mathbf{x}_1$ and $\mathbf{x}_d$, then the probability $P(e)$ of an error event can be upper bounded, using the union bound, as

$$P(e) \leq \sum_{d=d_f}^{\infty} A_d P(\mathbf{x}_1 \to \mathbf{x}_d) \tag{11.26}$$

where A_d is the number of paths of weight d diverging from the all-zero path. To proceed further, we must distinguish the cases of hard and soft-decoding.

Hard decoding Using (10.76), we have

$$P(\mathbf{x}_1 \to \mathbf{x}_d) \leq \left[\sqrt{4p(1-p)}\right]^d \tag{11.27}$$

Introducing (11.27) into (11.26) and recalling the definition (11.9) of the weight enumerating function $T(D)$, we finally obtain

$$P(e) \leq T(D)|_{D=\sqrt{4p(1-p)}} \tag{11.28}$$

This result emphasizes the role of the weight enumerating function $T(D)$ for the computation of the probability $P(e)$ of an error event, paralleling the results obtained for the word error probability of block codes.

Soft decoding The case of soft decoding is the same as the hard decoding one, except that the metric is the Euclidean distance, and, consequently, assuming a

[7] The pairwise error probability was defined in Chapter 4 and used in Chapter 10. Here, it represents the conditional probability that the sequence $\mathbf{x}_d$ has a larger metric than the correct sequence $\mathbf{x}_1$.

serial transmission of the coded bits using a binary antipodal modulation with energy $\mathcal{E}_b$, we can write the pairwise error probability as

$$P(\mathbf{x}_1 \to \mathbf{x}_d) = \frac{1}{2}\text{erfc}\left(\sqrt{\frac{dR_c\mathcal{E}_b}{N_0}}\right) \tag{11.29}$$

where R_c is the code rate. Substituting the right-hand side of (11.29) into (11.26), we get the union bound to the error event probability

$$P(e) \leq \frac{1}{2}\sum_{d=d_f}^{\infty} A_d \,\text{erfc}\left(\sqrt{\frac{dR_c\mathcal{E}_b}{N_0}}\right) \tag{11.30}$$

Using the inequality (A.5) $\frac{1}{2}\text{erfc}(\sqrt{x}) < \frac{1}{2}e^{-x}$, we can express (11.30) as

$$P(e) < \frac{1}{2}T(D)\Big|_{D=e^{-R_c(\mathcal{E}_b/N_0)}} \tag{11.31}$$

The difference between the two bounds (11.30) and (11.31), is that the first is tighter. However, the second can be evaluated from the closed-form knowledge of the weight enumerating function, whereas the first requires the *distance spectrum* of the code, i.e., the pairs $\{A_d, d\}_{d=d_f}^{\infty}$, which can be obtained from the power series expansion of $T(D)$. Usually, a small number of pairs are sufficient to obtain a close approximation. A bound in closed form tighter than (11.31) can also be derived (see Problem 11.7).

Bit error probability

The computation of an upper bound to the bit error probability is more difficult, and, in fact, the result is not always rigorously derived in textbooks.

Consider the trellis section between time m and time $m+1$, and assume that the all-zero sequence has been transmitted. Let $E(w, d, \ell)$ be the event that an error event with input information weight w, code sequence weight d, and length ℓ is active, i.e., has the highest path metric, in the interval $(m, m+1)$. Also, denote with $e(w, d, \ell)$ an error event starting at time m with input information weight w, code sequence weight d, and length ℓ.

The probability of the event $E(w, d, \ell)$ is easily bounded as

$$P[E(w, d, \ell)] \leq \ell P[e(w, d, \ell)] = \ell C_{w,d,\ell} P(\mathbf{x}_1 \to \mathbf{x}_d) \tag{11.32}$$

where $P[e(w, d, \ell)]$ is the probability of an error event produced by an incorrect path $\mathbf{x}_d$ of weight d, length ℓ and input weight w, and $C_{w,d,\ell}$ is the multiplicity of such incorrect paths. The first inequality in (11.32) relies on the fact that, to be