can be adjusted readily to accommodate a wide range of message sizes. Moreover, Reed–Solomon codes provide a wide range of code rates that can be chosen to optimize performance, and efficient techniques are available for their use in certain practical applications. In particular, a distinctive feature of Reed–Solomon codes is their ability to correct *bursts of errors*, hence their application in wireless communications to combat the fading phenomenon.

10.6 Convolutional Codes

In block coding, the encoder accepts a k-bit message block and generates an n-bit codeword, which contains $n - k$ parity-check bits. Thus, codewords are produced on a block-by-block basis. Clearly, provision must be made in the encoder to buffer an entire message block before generating the associated codeword. There are applications, however, where the message bits come in *serially* rather than in large blocks, in which case the use of a buffer may be undesirable. In such situations, the use of *convolutional coding* may be the preferred method. A convolutional coder generates redundant bits by using *modulo-2 convolutions*; hence the name *convolutional codes*[6].

The encoder of a binary convolutional code with rate $1/n$, measured in bits per symbol, may be viewed as a *finite-state machine* that consists of an M-stage shift register with prescribed connections to n modulo-2 adders and a multiplexer that serializes the outputs of the adders. A sequence of message bits produces a coded output sequence of length $n(L + M)$ bits, where L is the length of the message sequence. The *code rate* is therefore given by

$$\begin{aligned} r &= \frac{L}{n(L+M)} \\ &= \frac{1}{n(1+M/L)} \quad \text{bits/symbol} \end{aligned} \tag{10.47}$$

Typically, we have $L \gg M$, in which case the code rate is approximately defined by

$$r \approx \frac{1}{n} \quad \text{bits/symbol} \tag{10.48}$$

An important characteristic of a convolutional code is its constraint length, which we define as follows:

> The constraint length of a convolutional code, expressed in terms of message bits, is the number of shifts over which a single incoming message bit can influence the encoder output.

In an encoder with an M-stage shift register, the *memory* of the encoder equals M message bits. Correspondingly, the constraint length, denoted by ν, equals $M + 1$ shifts that are required for a message bit to enter the shift register and finally come out.

Figure 10.13 shows a convolutional encoder with the number of message bits $n = 2$ and constraint length $\nu = 3$. In this example, the code rate of the encoder is 1/2. The encoder operates on the incoming message sequence, one bit at a time, through a convolution process; it is therefore said to be a *nonsystematic* code.

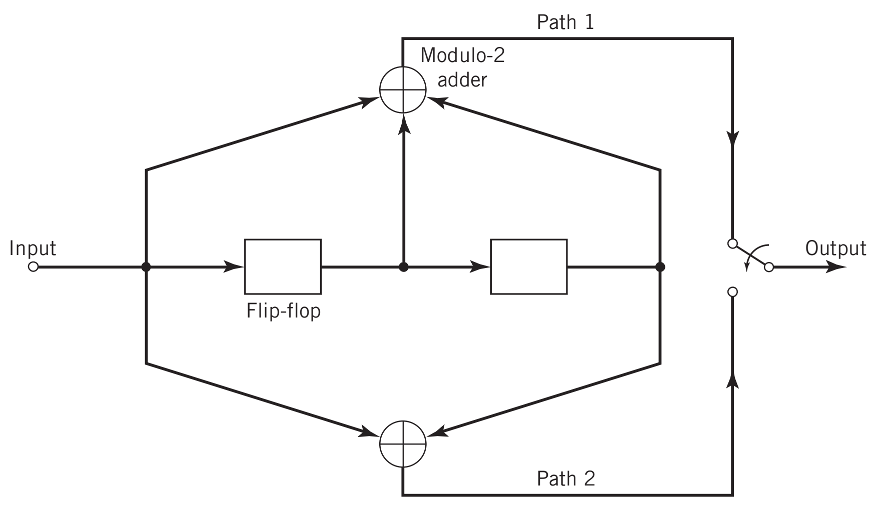

Figure 10.13
Constraint length-3, rate -1/2 convolutional encoder.

Each path connecting the output to the input of a convolutional encoder may be characterized in terms of its *impulse response*, defined as follows:

> The impulse response of a particular path in the convolutional encoder is the response of that path in the encoder to symbol 1 applied to its input, with each flip-flop in the encoder set initially to the zero state.

Equivalently, we may characterize each path in terms of a generator polynomial, defined as the *unit-delay transform* of the impulse response. To be specific, let the *generator sequence* $\left(g_0^{(i)}, g_1^{(i)}, g_2^{(i)}, \ldots, g_M^{(i)}\right)$ denote the impulse response of the ith path, where the coefficients $g_0^{(i)}, g_1^{(i)}, g_2^{(i)}, \ldots, g_M^{(i)}$ equal symbol 0 or 1. Correspondingly, the *generator polynomial* of the ith path is defined by

$$g^{(i)}(D) = g_0^{(i)} + g_1^{(i)}D + g_2^{(i)}D^2 + \cdots + g_M^{(i)}D^M \tag{10.49}$$

where D denotes the *unit-delay variable*. The complete convolutional encoder is described by the set of generator polynomials $\{g^{(i)}(D)\}_{i=1}^{M}$.

EXAMPLE 4 Convolutional Encoder

Consider again the convolutional encoder of Figure 10.13, which has two paths numbered 1 and 2 for convenience of reference. The impulse response of path 1 (i.e., upper path) is (1, 1, 1). Hence, the generator polynomial of this path is

$$g^{(1)}(D) = 1 + D + D^2$$

The impulse response of path 2 (i.e., lower path) is (1, 0, 1). The generator polynomial of this second path is

$$g^{(2)}(D) = 1 + D^2$$

For an incoming message sequence given by (10011), for example, we have the polynomial representation

$$m(D) = 1 + D^3 + D^4$$

As with Fourier transformation, convolution in the time domain is transformed into multiplication in the D-domain. Hence, the output polynomial of path 1 is given by

$$\begin{aligned} c^{(1)}(D) &= g^{(1)}(D)m(D) \\ &= (1 + D + D^2)(1 + D^3 + D^4) \\ &= 1 + D + D^2 + D^3 + D^6 \end{aligned}$$

where it is noted that the sums $D^4 + D^4$ and $D^5 + D^5$ are both zero in accordance with the rules of binary arithmetic. We therefore immediately deduce that the output sequence of path 1 is (1111001). Similarly, the output polynomial of path 2 is given by

$$\begin{aligned} c^{(2)}(D) &= g^{(2)}(D)m(D) \\ &= (1 + D^2)(1 + D^3 + D^4) \\ &= 1 + D^2 + D^3 + D^4 + D^5 + D^6 \end{aligned}$$

The output sequence of path 2 is therefore (1011111). Finally, *multiplexing* the two output sequences of paths 1 and 2, we get the encoded sequence

$$\mathbf{c} = (11, 10, 11, 11, 01, 01, 11)$$

Note that the message sequence of length $L = 5$ bits produces an encoded sequence of length $n(L + \nu - 1) = 14$ bits. Note also that for the shift register to be restored to its initial all-zero state, a terminating sequence of $\nu - 1 = 2$ zeros is appended to the last input bit of the message sequence. The terminating sequence of $\nu - 1$ zeros is called the *tail of the message*.

Code Tree, Trellis Graph, and State Graph

Traditionally, the structural properties of a convolutional encoder are portrayed in graphical form by using any one of three equivalent graphs: code tree, trellis graph, and state graph.

Although, indeed, these three graphical representations of a convolutional encoder look different, their compositions follow the same underlying rule:

> A code branch produced by input bit 0 is drawn as a solid line, whereas a code branch produced by input bit 1 is a dashed line.

Hereafter, we refer to this convention as the *graphical rule* of a convolutional encoder.

We will use the convolutional encoder of Figure 10.13 as a running example to illustrate the insights that each one of these three diagrams provides.

Code Tree

We begin the graphical representation of a convolutional encoder with the *code tree* of Figure 10.14. Each branch of the tree represents an input bit, with the corresponding pair of output bits indicated on the branch. The convention used to distinguish the input bits 0 and 1 follows the graphical rule described above. Thus, a specific *path* in the tree is traced from left to right in accordance with the message sequence. The corresponding coded bits on the branches of that path constitute the message sequence (10011) applied to the input of the encoder of Figure 10.13. Following the procedure just described, we find that the corresponding encoded sequence is (11, 10, 11, 11, 01), which agrees with the first five pairs of bits in the encoded sequence $\{c_i\}$ that was derived in Example 4.

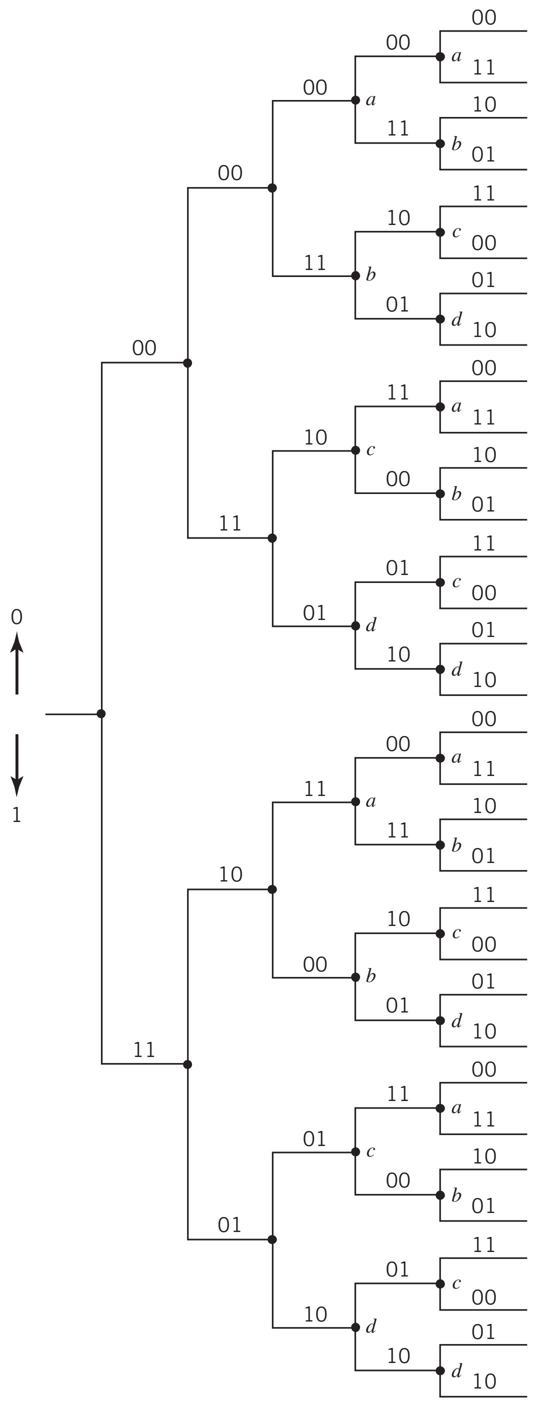

Figure 10.14 Code tree for the convolutional encoder of Figure 10.13.

Trellis Graph

From Figure 10.14, we observe that the tree becomes *repetitive* after the first three branches. Indeed, beyond the third branch, the two nodes labeled *a* are identical and so are all the other node pairs that are identically labeled. We may establish this repetitive property of the tree by examining the associated encoder of Figure 10.13. The encoder has memory $M = \nu - 1 = 2$ message bits. We therefore find that, when the third message bit enters the encoder, the first message bit is shifted out of the register. Consequently, after the third branch, the message sequences ($100\ m_3m_4\ldots$) and ($000\ m_3m_4\ldots$) generate the same code symbols, and the pair of nodes labeled *a* may be joined together. The same reasoning applies to the other nodes in the code tree. Accordingly, we may collapse the code tree of Figure 10.14 into the new form shown in Figure 10.15, which is called a *trellis*. It is so called since a trellis is a treelike structure with re-emerging branches. The convention used in Figure 10.15 to distinguish between input symbols 0 and 1 is as follows:

> A code branch in a trellis produced by input binary symbol 0 is drawn as a solid line, whereas a code branch produced by an input 1 is drawn as a dashed line.

As before, each message sequence corresponds to a specific path through the trellis. For example, we readily see from Figure 10.15 that the message sequence (10011) produces the encoded output sequence (11, 10, 11, 11, 01), which agrees with our previous result.

The Notion of State

In conceptual terms, a trellis is more instructive than a tree. We say so because it brings out explicitly the fact that the associated convolutional encoder is in actual fact a *finite-state machine*. Basically, such a machine consists of a tapped shift register and, therefore, has a finite state; hence the name of the machine. Thus, we may conveniently say the following:

> The state of a rate $1/n$ convolutional encoder is determined by the smallest number of message bits stored in memory (i.e., the shift register).

For example, the convolutional encoder of Figure 10.13 has a shift register made up of two memory cells. With the message bit stored in each memory cell being 0 or 1, it follows that this encoder can assume any one of $2^2 = 4$ possible states, as described in Table 10.5.

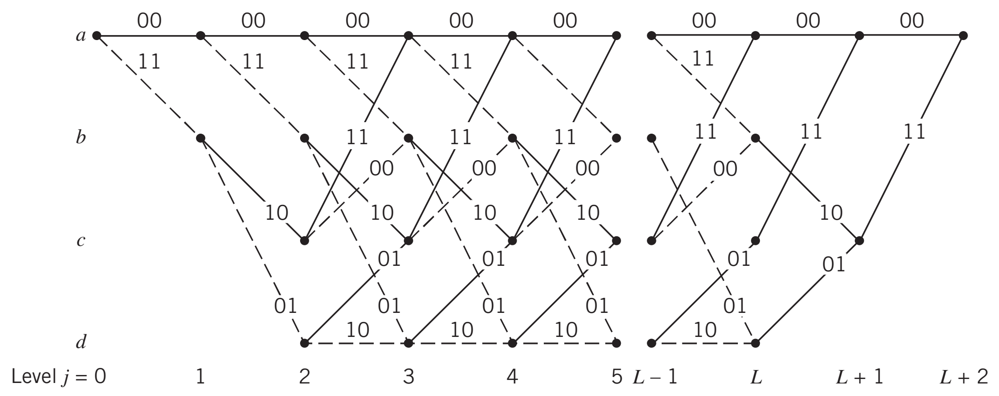

Figure 10.15 Trellis for the convolutional encoder of Figure 10.13.

Table 10.5 State table for the convolutional encoder of Figure 10.13

State	Binary description
a	00
b	10
c	01
d	11

In describing a convolutional encoder, the notion of state is important in the following sense:

> Given the current message bit and the state of the encoder, the codeword produced at the output of the encoder is completely determined.

To illustrate this statement, consider the general case of a rate $1/n$ convolutional encoder of constraint length ν. Let the state of the encoder at time-unit j be denoted by

$$S = (m_{j-1}, m_{j-2}, \dots, m_{j-\nu+1})$$

The jth codeword c_j is completely determined by the state S together with the current message bit m_j.

Now that we understand the notion of state, the trellis graph of the simple convolutional encoder of Figure 10.13 for $\nu = 3$ is presented in Figure 10.15. From this latter figure, we now clearly see a unique characteristic of the trellis diagram:

> The trellis depicts the evolution of the convolutional encoder's state across time.

To be more specific, the first $\nu - 1 = 2$ time-steps correspond to the encoder's departure from the initial zero state and the last $\nu - 1 = 2$ time-steps correspond to the encoder's return to the initial zero state. Naturally, not all the states of the encoder can be reached in these two particular portions of the trellis. However, in the central portion of the trellis, for which time-unit j lies in the range $\nu - 1 \le j \le L$, where L is the length of the incoming message sequence, we do see that all the four possible states of the encoder are reachable. Note also that the central portion of the trellis exhibits a *fixed periodic structure*, as illustrated in Figure 10.16a.

State Graph

The periodic structure characterizing the trellis leads us next to the state diagram of a convolutional encoder. To be specific, consider a central portion of the trellis corresponding to times j and $j + 1$. We assume that for $j \ge 2$ in the example of Figure 10.13, it is possible for the current state of the encoder to be a, b, c, or d. For convenience of presentation, we have reproduced this portion of the trellis in Figure 10.16a. The left nodes represent the four possible current states of the encoder, whereas the right nodes represent the next states. Clearly, we may coalesce the left and right nodes. By so doing, we obtain the *state graph* of the encoder, shown in Figure 10.16b. The nodes of the figure represent the four possible states of the encoder a, b, c, and d, with each node having two incoming branches and two outgoing branches, following the graphical rule described previously.

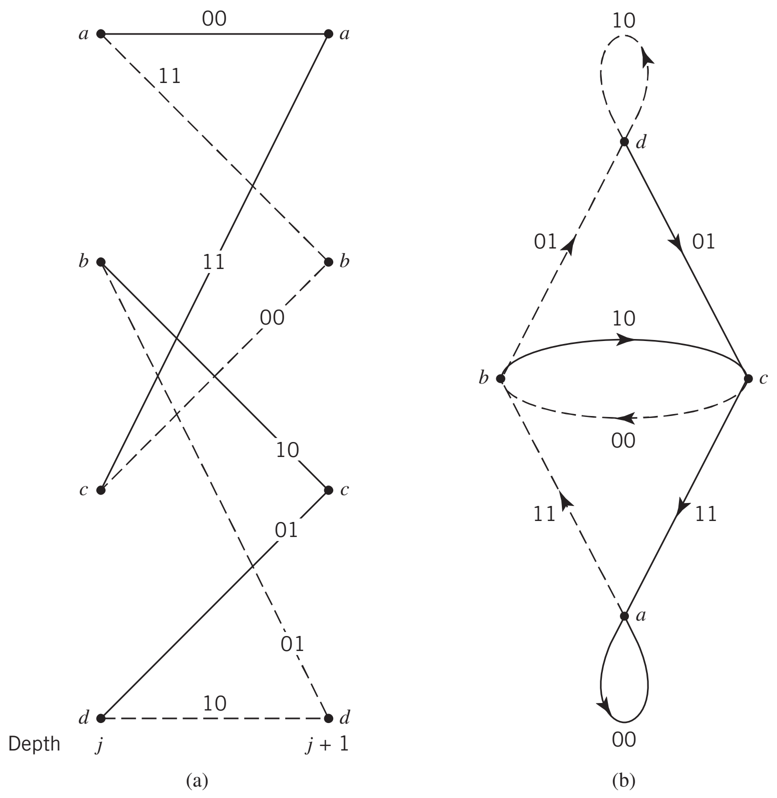

Figure 10.16 (a) A portion of the central part of the trellis for the encoder of Figure 10.13. (b) State graph of the convolutional encoder of Figure 10.13.

The binary label on each branch represents the encoder's output as it moves from one state to another. Suppose, for example, the current state of the encoder is (01), which is represented by node *c*. The application of input symbol 1 to the encoder of Figure 10.13 results in the state (10) and the encoded output (00). Accordingly, with the help of this state diagram, we may readily determine the output of the encoder of Figure 10.13 for any incoming message sequence. We simply start at state *a*, the *initial all-zero state*, and walk through the state graph in accordance with the message sequence. We follow a solid branch if the input is bit 0 and a dashed branch if it is bit 1. As each branch is traversed, we output the corresponding binary label on the branch. Consider, for example, the message sequence (10011). For this input, we follow the path *abcabd*, and therefore output the sequence (11, 10, 11, 11, 01), which agrees exactly with our previous result. Thus, the input–output relation of a convolutional encoder is also completely described by its state graph.

Recursive Systematic Convolutional Codes

The convolutional codes described thus far in this section have been feedforward structures of the nonsystematic variety. There is another type of linear convolutional codes that are the exact opposite, being recursive as well as systematic; they are called *recursive systematic convolutional (RSC) codes*.

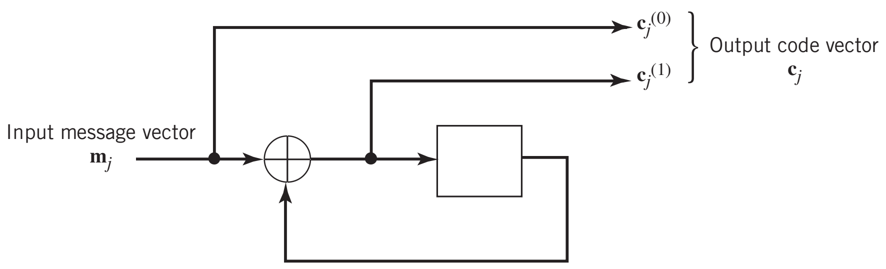
$\mathbf{c}_j^{(0)}$
$\mathbf{c}_j^{(1)}$
Output code vector $\mathbf{c}_j$
Input message vector $\mathbf{m}_j$

Figure 10.17 Example of a recursive systematic convolutional (RSC) encoder.

Figure 10.17 illustrates a simple example of an RSC code, two distinguishing features of which stand out in the figure:

1. The code is *systematic*, in that the incoming message vector $\mathbf{m}_j$ at time-unit j defines the systematic part of the code vector $\mathbf{c}_j$ at the output of the encoder.
2. The code is *recursive* by virtue of the fact that the other constituent of the code vector, namely the parity-check vector $\mathbf{b}_j$, is related to the message vector $\mathbf{m}_j$ by the *modulo-2 recursive equation*

$$\mathbf{m}_j + \mathbf{b}_{j-1} = \mathbf{b}_j \tag{10.50}$$

where $\mathbf{b}_{j-1}$ is the past value of $\mathbf{b}_j$ stored in the memory of the encoder.

From an analytic point of view, in studying RSC codes, it is more convenient to work in the transform D-domain than the time domain. By definition, we have

$$\mathbf{b}_{j-1} = D[\mathbf{b}_j] \tag{10.51}$$

and therefore rewrite (10.50) in the equivalent form:

$$\mathbf{b}_j = \frac{1}{1+D}[\mathbf{m}_j] \tag{10.52}$$

where the transfer function $1/(1 + D)$ operates on $\mathbf{m}_j$ to produce $\mathbf{b}_j$. With the code vector $\mathbf{c}_j$ consisting of the message vector $\mathbf{m}_j$ followed by the parity-check vector $\mathbf{b}_j$, we may express the code vector $\mathbf{c}_j$ produced in response to the message vector $\mathbf{m}_j$ as follows:

$$\begin{aligned}\mathbf{c}_j &= (\mathbf{m}_j, \mathbf{b}_j)\\ &= \left(1, \frac{1}{1+D}\right)\mathbf{m}_j\end{aligned} \tag{10.53}$$

It follows, therefore, that the *code generator* for the RSC code of Figure 10.17 is given by the matrix

$$\mathbf{G}(D) = \left(1, \frac{1}{1+D}\right) \tag{10.54}$$

Generalizing, we may now make the statement:

> For recursive systematic convolutional codes, the transform-domain matrix $\mathbf{G}(D)$ is easier to use as the code generator than the corresponding time-domain matrix $\mathbf{G}$ whose entries contain sequences of infinite length.

The same statement applies equally well to the parity-check generator $\mathbf{H}(D)$ compared with its time-domain counterpart $\mathbf{H}$.

The rationale behind making convolutional codes recursive is to feed one or more of the tap-outputs in the shift register back to the encoder input, which, in turn, makes the internal state of the shift register depend on past outputs. This modification, compared with a feedforward convolutional code, affects the behavior of error patterns in a profound way, which is emphasized in the following statement:

> A single error in the systematic bits of an RSC code produces an infinite number of parity-check errors due to the use of feedback in the encoder.

This property of recursive convolutional codes turns out to be one of the key factors behind the outstanding performance achieved by the class of turbo codes, to be discussed in Section 10.12. Therein, we shall see that feedback plays a key role not only in the encoder of turbo codes but also the decoder. For reasons that will become apparent later, further work on turbo codes will be deferred to Section 10.12.

10.7 Optimum Decoding of Convolutional Codes

In the meantime, we resume the discussion on convolutional codes whose encoders are of the feedforward variety, aimed at the development of two different decoding algorithms, each of which is optimum according to a criterion of its own.

The first algorithm is the *maximum likelihood (ML) decoding algorithm*; the decoder is itself referred to as the *maximum likelihood decoder* (maximum likelihood estimation was discussed in Chapter 3). A distinctive feature of this decoder is that it produces a codeword as output, the conditional probability of which is always maximized on the assumption that each codeword in the code is equiprobable. From Chapter 3 on probability theory, we recall that the conditional probability density function of a random variable X given a quantity θ can be rethought as the likelihood function of θ with that function being dependent on X, given a parameter θ. We may therefore make the statement:

> In the maximum likelihood decoding of a convolutional code, the metric to be maximized is the likelihood function of a codeword, expressed as a function of the noisy channel output.

The second algorithm is the *maximum a posteriori (MAP) probability decoding algorithm*; the decoder is correspondingly referred to as a *MAP decoder*. In light of this second algorithm's name, we may make the statement:

> In MAP decoding of a convolutional code, the metric to be maximized is the posterior of a codeword, expressed as the product of the likelihood function of a given bit and the a priori probability of that bit.

These two decoding algorithms, *optimal* in accordance with their own respective criteria, are distinguished from each other as follows:

> The ML decoding algorithm produces the most likely codeword as its output. On the other hand, the MAP decoding algorithm operates on the received sequence on a bit-by-bit basis to produce the most likely symbol as output.