CHAPTER 13

Suboptimum Decoding of Convolutional Codes

The primary difficulty with Viterbi and BCJR decoding of convolutional codes is that, even though they are optimum decoding methods, the arbitrarily low error probabilities promised by the random coding bound of (1.14) are not achievable in practice at rates close to capacity. This is because the decoding effort is fixed and grows exponentially with constraint length, and thus only short constraint length codes can be used. In this chapter we introduce two suboptimum decoding methods, sequential decoding and majority-logic or threshold decoding, that can be used to decode large constraint length convolutional codes.

We begin by noting that the fixed amount of computation required by the Viterbi and BCJR algorithms is not always needed, particularly when the noise is light. For example, assume that an encoded sequence of length N is transmitted without error over a BSC; that is, $\mathbf{r} = \mathbf{v}$. The Viterbi and BCJR algorithms still perform on the order of 2^{ν} computations per decoded information block, all of which is wasted effort in this case. In other words, it is sometimes desirable to have a decoding procedure whose effort is adaptable to the noise level. *Sequential decoding* is such a procedure. As we shall see, the decoding effort of sequential decoding is essentially independent of ν, so that large constraint lengths can be used. The number of computations per decoded information block is a random variable, however. This asynchronous nature of sequential decoding requires the use of buffers to store incoming data. Although most encoded sequences are decoded very quickly, some undergo long searches, which can lead to buffer overflow, causing information to be lost or erased. This problem limits the application of sequential decoding to rates strictly less than capacity.

An algebraic approach can also be taken to the decoding of convolutional codes. In particular, *majority-logic decoding*, first introduced in Chapter 8 for block codes, is applicable to convolutional codes. (In the soft-decision case, we use the more general term *threshold decoding*.) Majority-logic decoding differs from Viterbi, BCJR, and sequential decoding in that the final decision made on a given information block is based on only $(m + 1)$ received blocks rather than on the entire received sequence. Also, the codes used with majority-logic decoding must be *orthogonalizable*, a constraint that yields codes with relatively poor distance properties. The result is inferior performance when compared with Viterbi, BCJR, or sequential decoding, but the implementation of the decoder is much simpler.

Historically, sequential decoding was introduced by Wozencraft [1, 2] in 1957 as the first practical decoding method for convolutional codes. In 1963 Fano [3] introduced a new version of sequential decoding, subsequently referred to as the *Fano algorithm*. A few years later another version of sequential decoding, called the *ZJ* or *stack algorithm*, was discovered independently by Zigangirov [4] in 1966 and Jelinek [5] in 1969. Majority-logic or threshold decoding of convolutional codes, both the hard-decision and soft-decision (APP) versions, was introduced by Massey [6] in 1963.

13.1 THE ZJ (STACK) SEQUENTIAL DECODING ALGORITHM

In discussing sequential decoding, it is convenient to represent the 2^{kh} codewords of length $N = n(h+m)$ for a rate $R = k/n$ encoder with memory order m and an information sequence of length $K^* = kh$ as paths through a *code tree* containing $h+m+1$ time units or levels. The code tree is just an expanded version of the trellis diagram in which every path is distinct from every other path. The code tree for the (3, 1, 2) feedforward encoder with

$$\mathbb{G}(D) = [1 + D \quad 1 + D^2 \quad 1 + D + D^2] \tag{13.1}$$

is shown in Figure 13.1 for an information sequence of length $h = 5$. The trellis diagram for this encoder was shown in Figure 12.1. The $h+m+1$ tree levels are labeled from 0 to $h+m$ in Figure 13.1. The leftmost node in the tree is called the *origin node* and represents the starting state S_0 of the encoder. For a rate $R = k/n$ encoder, there are 2^k branches leaving each node in the first h levels of the tree. This region is called the *dividing part of the tree*. In Figure 13.1 the upper branch leaving each node in the dividing part of the tree represents the input bit $u_l = 1$, and the lower branch represents $u_l = 0$. After h time units there is only one branch leaving each node, which represents the inputs u_l (= 0, for feedforward encoders), for $l = h, h+1, \cdots, h+m-1$, and corresponds to the encoder's return to the all-zero state S_0. Hence, it is called the *tail part of the tree*, and the 2^{kh} rightmost nodes are called *terminal nodes*. Each branch is labeled with the n outputs $\mathbb{v}_l$ corresponding to the input sequence, and each of the 2^{kh} codewords of length N is represented by a totally distinct path through the tree. For example, the codeword corresponding to the input sequence $\mathbb{u} = (1\ 1\ 1\ 0\ 1\ 0\ 0)$ is shown boldfaced in Figure 13.1. It is important to realize that the code tree contains exactly the same information about the code as the trellis diagram or state diagram. As we shall see, however, the tree is better suited to explaining the operation of a sequential decoder.

There are a variety of tree-searching algorithms that fall under the general heading of sequential decoding. In this chapter we discuss the two most common of these, the ZJ or stack algorithm and the Fano algorithm, in considerable detail. The purpose of a sequential decoding algorithm is to search through the nodes of the code tree in an efficient way, that is, without having to examine too many nodes, in an attempt to find the maximum likelihood path. Each node examined represents a path through part of the tree. Whether a particular path is likely to be part of the maximum likelihood path depends on the metric value associated with that path. The metric is a measure of the "closeness" of a path to the received sequence.

For a binary-input, Q-ary output DMC, the metrics in the Viterbi algorithm are given by the log-likelihood function of (12.3). This function is the optimum metric for the Viterbi algorithm, since the paths being compared at any decoding step are all of the same length. In sequential decoding, however, the set of paths that have been examined after any decoding step are generally of many different lengths. If the log-likelihood metric is used for these paths, a distorted picture of the "closeness" of paths to the received sequence results.

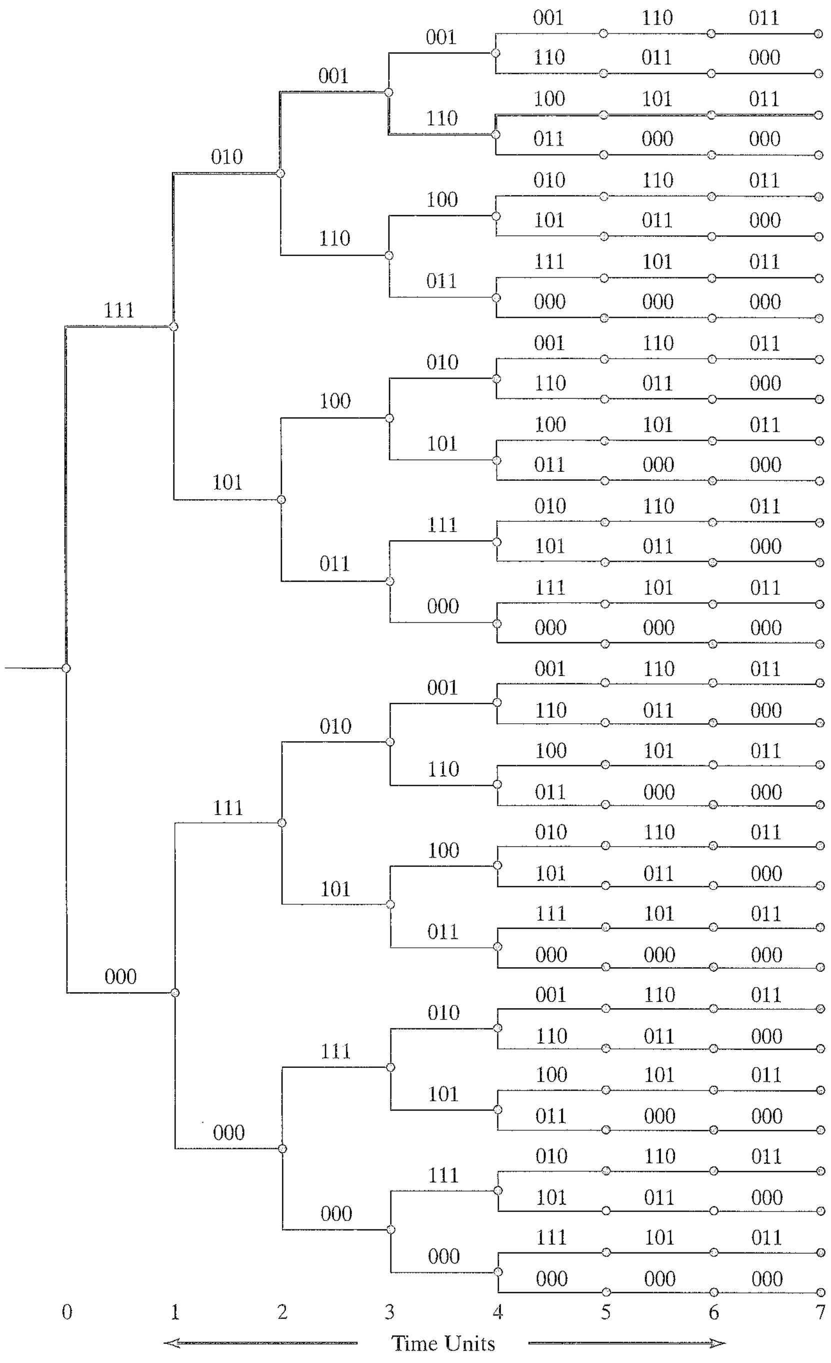


FIGURE 13.1: The code tree for a (3, 1, 2) encoder with $h = 5$.

EXAMPLE 13.1 The Log-Likelihood Metric for Sequential Decoding

Consider the code tree of Figure 13.1. Assume a codeword is transmitted from this code over a BSC, and the sequence

$$\mathbf{r} = (0\,1\,0, 0\,1\,0, 0\,0\,1, 1\,1\,0, 1\,0\,0, 1\,0\,1, 0\,1\,1) \tag{13.2}$$

is received. For a BSC, the log-likelihood metric for a path $\mathbf{v}$ in the Viterbi algorithm is given by $d(\mathbf{r}, \mathbf{v})$, with the maximum likelihood path being the one with the smallest metric. Now, consider comparing the partial path metrics for two paths of different lengths, for example, the truncated codewords $[\mathbf{v}]_5 = (111, 010, 001, 110, 100, 101)$ and $[\mathbf{v}']_0 = (000)$. The partial path metrics are $d([\mathbf{r}]_5, [\mathbf{v}]_5) = 2$ and $d([\mathbf{r}']_0, [\mathbf{v}']_0) = 1$, indicating that $[\mathbf{v}']_0$ is the "better" of the two paths; however, our intuition tells us that the path $[\mathbf{v}]_5$ is more likely to be part of the maximum likelihood path than $[\mathbf{v}']_0$, since to complete a path beginning with $[\mathbf{v}']_0$ requires 18 additional bits compared with only 3 additional bits required to complete the path beginning with $[\mathbf{v}]_5$. In other words, a path beginning with $[\mathbf{v}']_0$ is much more likely to accumulate additional distance from $\mathbf{r}$ than the path beginning with $[\mathbf{v}]_5$.

It is necessary, therefore, to adjust the metric used in sequential decoding to take into account the lengths of the different paths being compared. For a binary-input, Q-ary output DMC, Massey [7] has shown that the best bit metric to use when comparing paths of different lengths is

$$\begin{aligned} M(r_l|v_l) &= \log_2 \frac{P(r_l|v_l)}{P(r_l)} - R \\ &= \log_2 P(r_l|v_l) - \log_2 P(r_l) - R, \end{aligned} \tag{13.3}$$

where $P(r_l|v_l)$ is a channel transition probability, $P(r_l)$ is a channel output symbol probability, and R is the encoder rate. The partial path metric for the first t branches of a path $\mathbf{v}$ is given by

$$M([\mathbf{r}|\mathbf{v}]_t) = \sum_{l=0}^{t-1} M(\mathbf{r}_l|\mathbf{v}_l) = \sum_{l=0}^{nt-1} M(r_l|v_l), \tag{13.4}$$

where $M(\mathbf{r}_l|\mathbf{v}_l)$, the branch metric for the lth branch, is computed by adding the bit metrics for the n bits on that branch. Combining (13.3) and (13.4) we have

$$M([\mathbf{r}|\mathbf{v}]_t) = \sum_{l=0}^{nt-1} \log_2 P(r_l|v_l) - \sum_{l=0}^{nt-1} \log_2 P(r_l) - ntR. \tag{13.5}$$

A binary-input, Q-ary output DMC is said to be *symmetric* if

$$P(j|0) = P(Q-1-j|1), \quad j = 0, 1, \cdots, Q-1. \tag{13.6}$$

(A symmetric channel model results when (1) a symmetric modulator mapping is used, for example, $0 \rightarrow -1$ and $1 \rightarrow +1$; (2) the noise distribution is symmetric; and

(3) the demodulator output quantization is symmetric.) For a symmetric channel with equally likely input symbols,[1] the channel output symbol probabilities satisfy (see Problem 13.2)

$$P(r_l = j) = P(r_l = Q - 1 - j) \leq \tfrac{1}{2}, \text{ for } 0 \leq j \leq Q - 1 \text{ and all } l, \tag{13.7}$$

and (13.5) reduces to

$$\begin{aligned} M([\mathbf{r}|\mathbf{v}]_t) &= \sum_{l=0}^{nt-1} \log_2 P(r_l|v_l) - \sum_{l=0}^{nt-1} \Big[\log_2 P(r_l) + R \Big] \\ &= \underbrace{\sum_{l=0}^{nt-1} \log_2 P(r_l|v_l)}_{ML\ metric} + \underbrace{\sum_{l=0}^{nt-1} \Big[\log_2 \frac{1}{P(r_l)} - R \Big]}_{positive\ bias}. \end{aligned} \tag{13.8}$$

The first term in (13.8) is the maximum likelihood (ML) metric for the Viterbi algorithm (see (12.5)). The second term represents a positive (since $\log_2 \frac{1}{P(r_l)} \geq 1$ and $R \leq 1$) bias that increases with path length. Hence, longer paths have a larger bias than shorter paths, indicating that they are closer to the end of the tree and hence more likely to be part of the ML path. The bit metric of (13.3) was first introduced by Fano [3] on intuitive grounds, and hence it is called the *Fano metric*. It is the metric most commonly used for sequential decoding, although some other metrics have been proposed. When comparing paths of different lengths, the path with the largest Fano metric is considered the "best" path, that is, most likely to be part of the ML path.

EXAMPLE 13.2 The Fano Metric for Sequential Decoding

For a BSC ($Q = 2$) with transition probability p, $P(r_l = 0) = P(r_l = 1) = \frac{1}{2}$ for all l, and the Fano metrics for the truncated codewords $[\mathbf{v}]_5$ and $[\mathbf{v}']_0$ in Example 13.1 are given by

$$\begin{aligned} M([\mathbf{r}|\mathbf{v}]_5) &= 16 \log_2(1-p) + 2\log_2 p + 18(1 - 1/3) \\ &= 16\log_2(1-p) + 2\log_2 p + 12 \end{aligned} \tag{13.9a}$$

and

$$\begin{aligned} M([\mathbf{r}|\mathbf{v}']_0) &= 2\log_2(1-p) + \log_2 p + 3(1 - 1/3) \\ &= 2\log_2(1-p) + \log_2 p + 2. \end{aligned} \tag{13.9b}$$

For $p = .10$,

$$M([\mathbf{r}|\mathbf{v}]_5) = 2.92 > M([\mathbf{r}|\mathbf{v}']_0) = -1.63, \tag{13.10}$$

indicating that $[\mathbf{v}]_5$ is the "better" of the two paths. This result differs from that obtained using the log-likelihood metric, since the bias term in the Fano metric reflects the difference in the path lengths.

[1] Because convolutional codes are linear, the set of all codewords contains an equal number of 0's and 1's, and hence the channel input symbols are equally likely.

In general, for a BSC with transition probability p, $P(r_l = 0) = P(r_l = 1) = \frac{1}{2}$ for all l, and the bit metrics are given by

$$M(r_l|v_l) = \begin{cases} \log_2 p - \log_2 \frac{1}{2} - R & \text{if } r_l \neq v_l \\ \log_2(1-p) - \log_2 \frac{1}{2} - R & \text{if } r_l = v_l \end{cases}$$
$$= \begin{cases} \log_2 2p - R & \text{if } r_l \neq v_l \\ \log_2 2(1-p) - R & \text{if } r_l = v_l \end{cases}. \tag{13.11}$$

EXAMPLE 13.3 Bit Metric for a BSC

For $R = 1/3$ and $p = .10$,

$$M(r_l|v_l) = \begin{cases} -2.65 & \text{if } r_l \neq v_l \\ +0.52 & \text{if } r_l = v_l \end{cases}, \tag{13.12}$$

and we have the metric table shown in Figure 13.2(a). It is common practice to scale the metrics by a positive constant so that they can be closely approximated as integers for ease of computation. In this case, the scaling factor of 1/.52 yields the integer metric table shown in Figure 13.2(b).

For the BSC of Example 13.3, any sequential decoding algorithm that uses the Fano metric will compute the metric of a path by (1) assigning a +1 to every bit in $\mathbf{v}$ that *agrees* with the received sequence $\mathbf{r}$, and (2) assigning a −5 to every bit in $\mathbf{v}$ that *disagrees* with $\mathbf{r}$. Hence, a path with only a few errors, such as, typically, the *correct path*, will tend to have a slowly increasing metric. On the other hand, paths with many errors, such as, typically, all *incorrect paths*, will tend to have a rapidly decreasing metric. Thus, incorrect paths are not extended very far into the tree before being rejected. Therefore, typically, sequential decoding algorithms require much less computation than the Viterbi or BCJR algorithms. Unlike those algorithms, however, the amount of computation required by a sequential decoder depends on the noise and is thus variable. Generally, most received sequences are decoded very quickly, but some noisy sequences may take a long time to decode.

For any BSC, the bit-metric positive bias term in (13.8) is given by

$$\log_2 \frac{1}{P(r_l)} - R = \log_2 2 - R = 1 - R \textit{ for all } l \tag{13.13}$$

$v_l \backslash r_l$	0	1
0	+.52	−2.65
1	−2.65	+.52

(a)

$v_l \backslash r_l$	0	1
0	+1	−5
1	−5	+1

(b)

FIGURE 13.2: Metric tables for a rate $R = 1/3$ encoder and a BSC with $p = .10$.

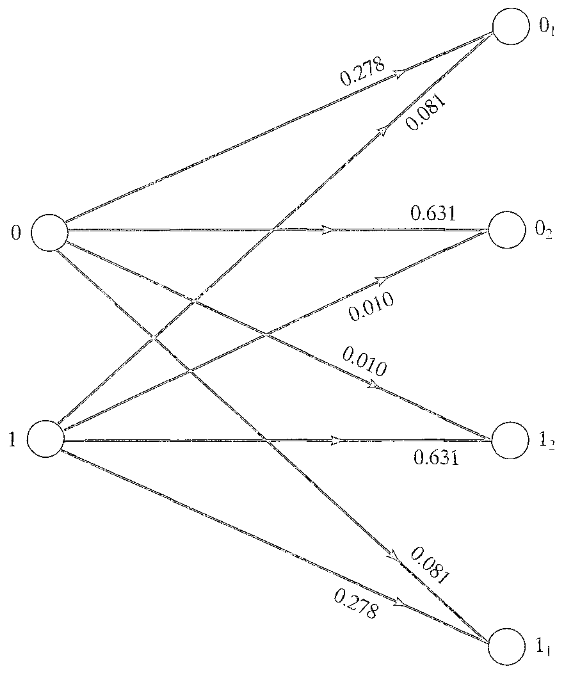


FIGURE 13.3: A binary-input, quaternary-output DMC.

that is, the positive bias term is constant for every bit, and the cumulative positive bias for a path of length nt is $nt(1-R)$. In this case we see that the metric bias increases linearly with path length. On the other hand, for a DMC with $Q > 2$, since not all channel output symbols have the same probability, the value of the bias term will depend on the output symbol. For each output symbol, however, the bias will be positive.

EXAMPLE 13.4 Bit Metrics for a DMC

Consider the binary-input, quaternary output ($Q = 4$) DMC shown in Figure 13.3. For this channel the output symbol probabilities are given by

$$\begin{aligned} P(r_l = 0_1) &= P(v_l = 0)P(r_l = 0_1|v_l = 0) + P(v_l = 1)P(r_l = 0_1|v_l = 1) \\ &= \tfrac{1}{2}(0.631) + \tfrac{1}{2}(0.01) = 0.3205, \end{aligned} \tag{13.14a}$$

$$\begin{aligned} P(r_l = 0_2) &= P(v_l = 0)P(r_l = 0_2|v_l = 0) + P(v_l = 1)P(r_l = 0_2|v_l = 1) \\ &= \tfrac{1}{2}(0.278) + \tfrac{1}{2}(0.081) = 0.1795, \end{aligned} \tag{13.14b}$$

$$P(r_l = 1_2) = P(r_l = 0_2) = 0.1795, \tag{13.14c}$$

$$P(r_l = 1_1) = P(r_l = 0_1) = 0.3205. \tag{13.14d}$$

Hence, there are two possible positive bias terms,

$$\log_2 \frac{1}{P(r_l = 0_1)} - R = \log_2 \frac{1}{P(r_l = 1_1)} - R = 1.64 - R, \tag{13.15a}$$

and

$$\log_2 \frac{1}{P(r_l = 0_2)} - R = \log_2 \frac{1}{P(r_l = 1_2)} - R = 2.48 - R, \tag{13.15b}$$

and the cumulative positive bias for a path depends on the received sequence $\mathbf{r}$.

Example 13.4 illustrates that for a DMC the metric bias always increases monotonically with path length, but the increase is not necessarily linear.

For a binary-input AWGN channel with unquantized outputs, the Fano bit metric is given by

$$M(r_l|v_l) = \log_2 p(r_l|v_l) - \log_2 p(r_l) - R, \tag{13.16}$$

where $p(r_l|v_l)$ is the conditional pdf of the received symbol r_l given the transmitted symbol v_l, $p(r_l)$ is the marginal pdf of r_l, and R is the encoder rate. In this case the transmitted symbols are assumed to be either $+1$ or -1, and since these symbols are equally likely, the marginal pdf of the received symbol r_l is calculated as

$$p(r_l) = \frac{p(r_l|v_l = +1) + p(r_l|v_l = -1)}{2}, \tag{13.17}$$

where $p(r_l|v_l = +1)$ and $p(r_l|v_l = -1)$ are given in (12.13). Finally, the branch metrics and path metrics for an AWGN channel are computed from the bit metrics in the same way as for a DMC.

In the *ZJ* or *stack algorithm*, an ordered list or stack of previously examined paths of different lengths is kept in storage. Each stack entry contains a path along with its metric, the path with the largest metric is placed on top, and the others are listed in order of decreasing metric. Each decoding step consists of extending the top path in the stack by computing the branch metrics of its 2^k succeeding branches and then adding these to the metric of the top path to form 2^k new paths, called the *successors* of the top path. The top path is then deleted from the stack, its 2^k successors are inserted, and the stack is rearranged in order of decreasing metric values. When the top path in the stack is at the end of the tree, the algorithm terminates.

The ZJ Algorithm

Step 1. Load the stack with the origin node in the tree, whose metric is taken to be zero.
Step 2. Compute the metrics of the successors of the top path in the stack.
Step 3. Delete the top path from the stack.
Step 4. Insert the new paths in the stack and rearrange the stack in order of decreasing metric values.
Step 5. If the top path in the stack ends at a terminal node in the tree, stop. Otherwise, return to step 2.

When the algorithm terminates, the top path in the stack is taken as the decoded path. A complete flowchart for the ZJ algorithm is shown in Figure 13.4.

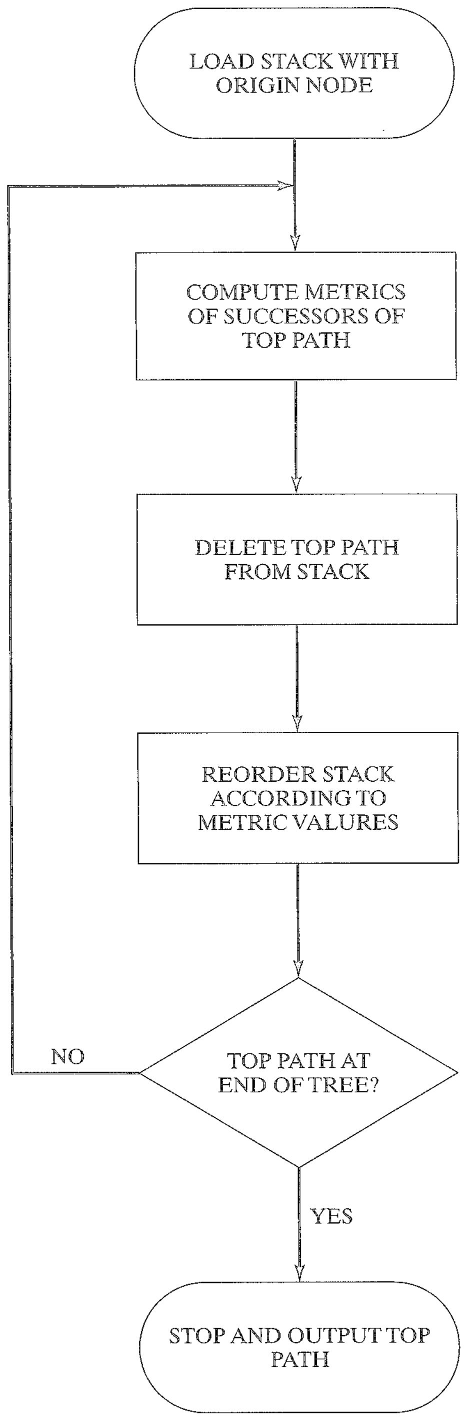

LOAD STACK WITH ORIGIN NODE
COMPUTE METRICS OF SUCCESSORS OF TOP PATH
DELETE TOP PATH FROM STACK
REORDER STACK ACCORDING TO METRIC VALURES
TOP PATH AT END OF TREE?
NO
YES
STOP AND OUTPUT TOP PATH

FIGURE 13.4: Flowchart for the ZJ algorithm.

In the dividing part of the tree, there are 2^k new metrics to be computed at step 2. In the tail of the tree, only one new metric is computed. Note that for $(n, 1, \nu)$ codes, the size of the stack increases by one for each decoding step in the dividing part of the tree, but does not increase at all when the decoder is in the tail of the tree. Because the dividing part of the tree is typically much longer than the tail ($h >> m$), the size of the stack is roughly equal to the number of decoding steps when the algorithm terminates.

EXAMPLE 13.5 The ZJ Algorithm

Consider the application of the ZJ algorithm to the code tree of Figure 13.1. Assume a codeword is transmitted from this code over a BSC with $p = .10$, and the sequence

$$\mathbf{r} = (0\,1\,0, 0\,1\,0, 0\,0\,1, 1\,1\,0, 1\,0\,0, 1\,0\,1, 0\,1\,1) \tag{13.18}$$

is received. Using the integer metric table of Figure 13.2(b), we show the contents of the stack after each step of the algorithm in Figure 13.5 and illustrate the decoding process in Figure 13.6. The algorithm terminates after 10 decoding steps, and the final decoded path is

$$\hat{\mathbf{v}} = (1\,1\,1, 0\,1\,0, 0\,0\,1, 1\,1\,0, 1\,0\,0, 1\,0\,1, 0\,1\,1), \tag{13.19}$$

corresponding to the information sequence $\hat{\mathbf{u}} = (1\ 1\ 1\ 0\ 1)$. In this example, ties in the metric values were resolved by placing the longest path on top, which had the effect of slightly reducing the total number of decoding steps. In general, however, the resolution of ties is arbitrary and does not affect the error probability of the decoder.

Step 1	Step 2	Step 3	Step 4	Step 5
0(−3)	00(−6)	000(−9)	1(−9)	11(−6)
1(−9)	1(−9)	1(−9)	0001(−12)	0001(−12)
	01(−12)	01(−12)	01(−12)	01(−12)
		001(−15)	001(−15)	001(−15)
			0000(−18)	0000(−18)
				10(−24)

Step 6	Step 7	Step 8	Step 9	Step 10
111(−3)	1110(0)	11101(+3)	111010(+6)	1110100(+9)
0001(−12)	0001(−12)	0001(−12)	0001(−12)	0001(−12)
01(−12)	01(−12)	01(−12)	01(−12)	01(−12)
001(−15)	001(−15)	11100(−15)	11100(−15)	11100(−15)
0000(−18)	1111(−18)	001(−15)	001(−15)	001(−15)
110(−21)	0000(−18)	1111(−18)	1111(−18)	1111(−18)
10(−24)	110(−21)	0000(−18)	0000(−18)	0000(−18)
	10(−24)	110(−21)	110(−21)	110(−21)
		10(−24)	10(−24)	10(−24)

FIGURE 13.5: Stack contents in Example 13.5.

It is interesting to compare the number of decoding steps required by the ZJ algorithm with the number required by the Viterbi algorithm. A decoding step or computation for the Viterbi algorithm is the ACS operation performed for each state in the trellis diagram beyond time unit m. Hence, the Viterbi algorithm would require 15 computations to decode the received sequence in Example 13.5 (see the trellis diagram of Figure 12.6). In the ZJ algorithm, a single execution of steps 2–4 is counted as a computation.[2] Thus, the ZJ algorithm requires only 10

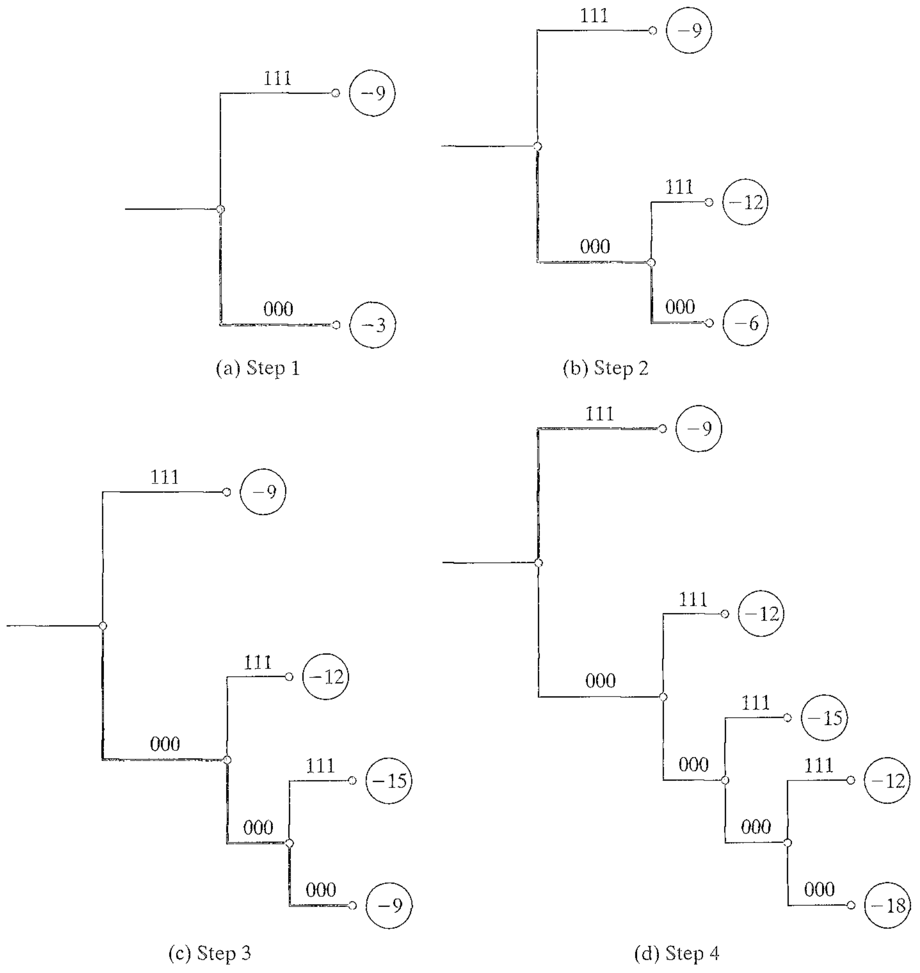
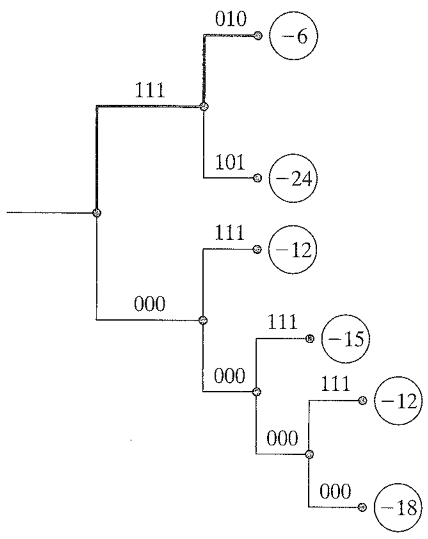
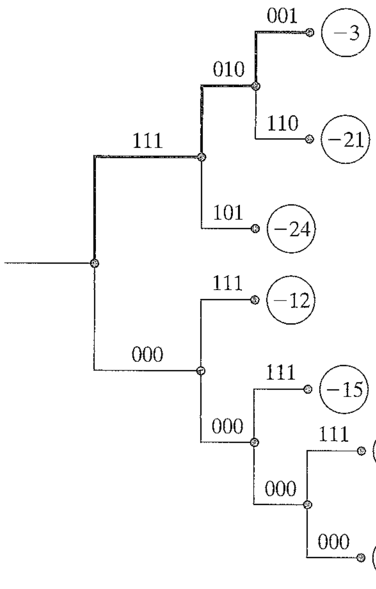


FIGURE 13.6: The decoding process in Example 13.5.

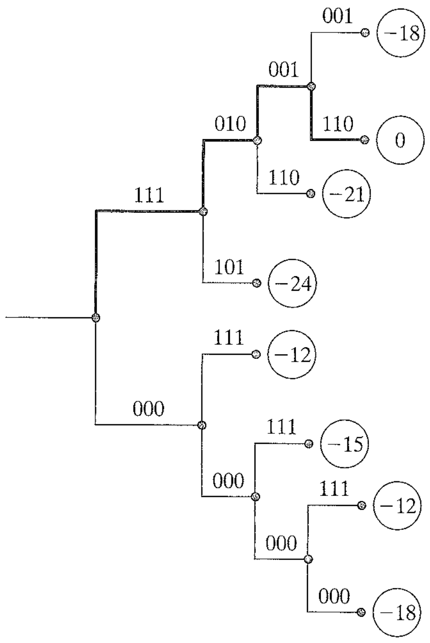
[2]A computation in the ZJ algorithm is somewhat more complicated than the ACS operation in the Viterbi algorithm, since the stack must be reordered after each path is extended; however, this reordering problem is eliminated, with very little loss in performance, in the stack-bucket algorithm [5], to be discussed later in this section.



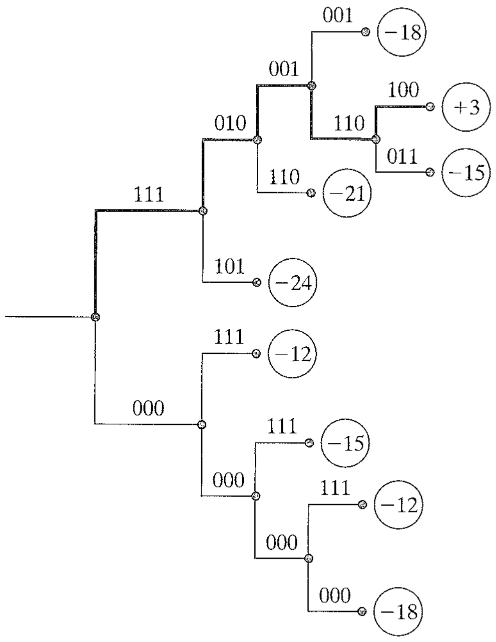
(e) Step 5



(f) Step 6

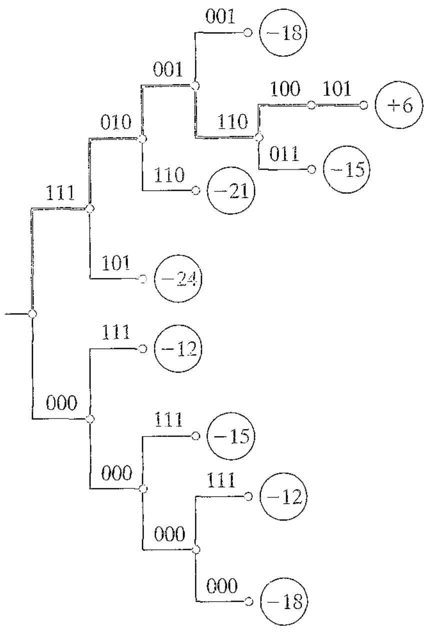


(g) Step 7

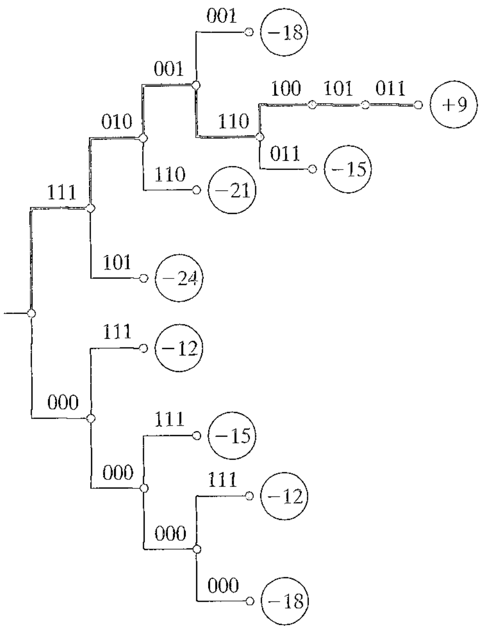


(h) Step 8

FIGURE 13.6: (*continued*)



(i) Step 9



(j) Step 10

FIGURE 13.6: (*continued*)

computations to decode the received sequence in Example 13.5. This computational advantage of sequential decoding over the Viterbi algorithm is typical when the received sequence is not too noisy, that is, when it contains a fraction of errors not too much greater than the channel transition probability p. Note that $\hat{\mathbf{v}}$ in Example 13.5 disagrees with $\mathbf{r}$ in only 2 positions and agrees with $\mathbf{r}$ in the other 19 positions. Assuming that $\hat{\mathbf{v}}$ was actually transmitted, the fraction of errors in $\mathbf{r}$ is $\frac{2}{21} = 0.095$, which is roughly equal to the channel transition probability of $p = .10$ in this case.

The situation is somewhat different when the received sequence $\mathbf{r}$ is very noisy, however, as illustrated in Example 13.6.

EXAMPLE 13.6 The ZJ Algorithm Revisited

For the same code, channel, and metric table as in Example 13.5, assume that the sequence

$$\mathbf{r} = (1\,1\,0, 1\,1\,0, 1\,1\,0, 1\,1\,1, 0\,1\,0, 1\,0\,1, 1\,0\,1) \qquad (13.20)$$

is received. The contents of the stack after each step of the algorithm are shown in Figure 13.7. The algorithm terminates after 20 decoding steps, and the final decoded path is

$$\hat{\mathbf{v}} = (1\,1\,1, 0\,1\,0, 1\,1\,0, 0\,1\,1\,, 1\,1\,1, 1\,0\,1, 0\,1\,1), \qquad (13.21)$$

corresponding to the information sequence $\hat{\mathbf{u}} = (1\,1\,0\,0\,1)$.

Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7
1(−3)	11(−6)	110(−3)	1100(−6)	11000(−9)	0(−9)	1101(−12)
0(−9)	0(−9)	0(−9)	0(−9)	0(−9)	1101(−12)	01(−12)
	10(−12)	10(−12)	1101(−12)	1101(−12)	10(−12)	10(−12)
		111(−21)	10(−12)	10(−12)	11001(−15)	11001(−15)
			111(−21)	11001(−15)	110000(−18)	110000(−18)
				111(−21)	111(−21)	00(−18)
						111(−21)

Step 8	Step 9	Step 10	Step 11	Step 12	Step 13	Step 14
11011(−9)	01(−12)	10(−12)	11001(−15)	110010(−12)	101(−15)	011(−15)
01(−12)	10(−12)	11001(−15)	101(−15)	101(−15)	011(−15)	110110(−18)
10(−12)	11001(−15)	011(−15)	011(−15)	011(−15)	110110(−18)	110000(−18)
11001(−15)	110110(−18)	110110(−18)	110110(−18)	110110(−18)	110000(−18)	1010(−18)
110000(−18)	110000(−18)	110000(−18)	110000(−18)	110000(−18)	00(−18)	00(−18)
00(−18)	00(−18)	00(−18)	00(−18)	00(−18)	1100100(−21)	1100100(−21)
111(−21)	111(−21)	010(−21)	100(−21)	100(−21)	100(−21)	100(−21)
11010(−27)	11010(−27)	111(−21)	010(−21)	010(−21)	010(−21)	010(−21)
		11010(−27)	111(−21)	111(−21)	111(−21)	111(−21)
			11010(−27)	11010(−27)	11010(−27)	1011(−24)
						11010(−27)

Step 15	Step 16	Step 17	Step 18	Step 19	Step 20
110110(−18)	110000(−18)	0110(−18)	1010(−18)	00(−18)	1100100(−21)
110000(−18)	0110(−18)	1010(−18)	00(−18)	1100100(−21)	10100(−21)
0110(−18)	1010(−18)	00(−18)	1100100(−21)	10100(−21)	01100(−21)
1010(−18)	00(−18)	1100100(−21)	01100(−21)	01100(−21)	001(−21)
00(−18)	1100100(−21)	100(−21)	100(−21)	100(−21)	100(−21)
1100100(−21)	100(−21)	010(−21)	010(−21)	010(−21)	010(−21)
100(−21)	010(−21)	111(−21)	111(−21)	111(−21)	111(−21)
010(−21)	111(−21)	0111(−24)	0111(−24)	0111(−24)	0111(−24)
111(−21)	0111(−24)	1011(−24)	1011(−24)	1011(−24)	1011(−24)
0111(−24)	1011(−24)	1100000(−27)	1100000(−27)	1100000(−27)	1100000(−27)
1011(−24)	1101100(−27)	1101100(−27)	1101100(−27)	1101100(−27)	1101100(−27)
11010(−27)	11010(−27)	11010(−27)	01101(−27)	10101(−27)	10101(−27)
			11010(−27)	01101(−27)	01101(−27)
				11010(−27)	11010(−27)
					000(−27)

FIGURE 13.7: Stack contents in Example 13.6.

In this example the sequential decoder performs 20 computations, whereas the Viterbi algorithm would again require only 15 computations. This points out one of the most important differences between sequential decoding and Viterbi decoding, that is, the number of computations performed by a sequential decoder is a random variable, whereas the computational load of the Viterbi algorithm is fixed. If we assume that $\hat{\mathbf{v}}$ was actually transmitted in Example 13.6, the fraction of errors in the received sequence $\mathbf{r}$ is $\frac{7}{21} = 0.333$, which is much greater than the channel transition probability of $p = .10$ in this case. This illustrates that very noisy received sequences typically require a large number of computations with a sequential decoder, sometimes more than the *fixed* number of computations required by the Viterbi algorithm; however, since very noisy received sequences do not occur very often, the *average* number of computations performed by a sequential decoder is normally much less than the *fixed* number performed by the Viterbi algorithm.

The received sequence in Example 13.6 was also decoded using the Viterbi algorithm in Section 12.1. The final decoded path was the same in both cases. (It is easy to verify that this is also true for Example 13.5.) This result illustrates the important fact that a sequential decoder almost always produces the ML path, even when the received sequence is very noisy. Hence, for a given code, the error probability of sequential decoding, even though it is suboptimum, is essentially the same as for Viterbi decoding.

There are some problems associated with the implementation of the stack sequential decoding algorithm, however, that limit its overall performance. First, since the decoder traces a somewhat random path back and forth through the code tree, jumping from node to node, the decoder must have an input buffer to store incoming blocks of the received sequence while they are waiting to be processed. Depending on the *speed factor* of the decoder, that is, the ratio of the speed at which computations are performed to the speed of the incoming data (in branches received per second), long searches can cause the input buffer to overflow, resulting in a loss of data, or an *erasure*. The buffer accepts data at the fixed rate of $1/nT$ branches per second, where T is the time interval allotted for each transmitted bit, and outputs these branches to the decoder asynchronously as demanded by the algorithm. Normally, the information is divided into *frames* of h branches, each terminated by a sequence of $m << h$ input blocks to return the encoder to the all-zero state. Even if the input buffer is one or two orders of magnitude larger than h, there is some probability that it will fill up during the decoding of a given frame and that the next branch received from the channel will then cause an undecoded branch from the frame being processed to be shifted out of the buffer. This data is then lost, and the frame must be erased. Erasure probabilities of around 10^{-3} are not unusual in sequential decoding, which means that a particular frame has a probability of 10^{-3} of not being decoded owing to an overflow of the input buffer.

The number of computations performed by a sequential decoder, and also its erasure probability, are essentially independent of the encoder constraint length ν. Therefore, codes with large values of ν,[3] and hence large free distance, can be selected for use with sequential decoding. Undetected errors (complete but incorrect decoding) then become extremely unlikely, and the major limitation on performance is the erasure probability due to buffer overflow.

Even though an erasure probability of around 10^{-3} can be rather troublesome in some systems, it may actually be beneficial in others. Because erasures usually occur when the received sequence is very noisy, if decoding is completed, and even if the ML path is obtained, there is a fairly high probability that the estimate will be incorrect. In many systems it is more desirable to erase such a frame than to decode it incorrectly. In other words, a complete decoder, such as the Viterbi algorithm,

[3] $\nu \approx km$ cannot be made too large for fixed h, since then the fractional rate loss (see (11.92)) would be significant; however, values of ν up to about 50 are certainly feasible for sequential decoders with $h \approx 1000$.

would always decode such a frame, even though it is likely to be decoded incorrectly, whereas a sequential decoder trades errors for erasures by "sensing" noisy frames and then erasing them. This property of sequential decoding can be used in an ARQ retransmission scheme (see Chapter 22) as an indicator of when a frame should be retransmitted.

A second problem in any practical implementation of the stack algorithm is that the size of the stack must be finite. In other words, there is always some probability that the stack will fill up before decoding is completed (or the buffer overflows) on a given frame. The most common way of handling this problem is simply to allow the path at the bottom of the stack to be pushed out of the stack on the next decoding step. This path is then "lost" and can never return to the stack. For typical stack sizes on the order of 1000 entries, the probability that a path on the bottom of the stack will ever recover to reach the top of the stack and be extended is so small that the loss in performance is negligible.

A related problem has to do with the reordering of the stack after each decoding step. This can become quite time-consuming as the number of entries in the stack becomes large and places severe limitations on the decoding speed that can be achieved with the basic algorithm. Jelinek [5] has proposed a modified algorithm, called the *stack-bucket* algorithm, in which the contents of the stack do not have to be reordered after each decoding step. In the stack-bucket algorithm, the range of possible metric values (from $+21$ to -110 in the preceding examples) is quantized into a fixed number of intervals. Each metric interval is assigned a certain number of locations in storage, called a *bucket*. When a path is extended, it is deleted from storage and each new path is inserted as the top entry in the bucket that includes its metric value. No reordering of paths within buckets is required. The top path in the top nonempty bucket is chosen to be extended. A computation now involves only finding the correct bucket in which to place each new path, which is independent of the number of previously extended paths, and is therefore faster than reordering an increasingly large stack. The disadvantage of the stack-bucket algorithm is that it is not always the best path that gets extended but only a "very good" path, that is, a path in the top nonempty bucket, or the "best" bucket. Typically, though, if the quantization of metric values is not too coarse and the received sequence is not too noisy, the best bucket contains only the best path, and the degradation in performance from the basic algorithm is very slight. The speed savings of the bucket algorithm is considerable, though, and all practical implementations of the ZJ algorithm have used this approach.

13.2 THE FANO SEQUENTIAL DECODING ALGORITHM

Another approach to sequential decoding, the *Fano algorithm*, sacrifices some speed compared with the ZJ algorithm but requires essentially no storage. The speed disadvantage of the Fano algorithm is due to the fact that it generally extends more nodes than the ZJ algorithm, although this is mitigated somewhat by the fact that no stack reordering is necessary. In the Fano algorithm the decoder examines a sequence of nodes in the tree, starting with the origin node and ending with one of the terminal nodes. The decoder never jumps from node to node, as in the ZJ algorithm, but always moves to an adjacent node, either forward to one of the 2^k nodes leaving the present node, or backward to the node leading to the present node. The metric of the next node to be examined can then be computed by adding (or subtracting) the metric of the connecting branch to the metric of the present node. This process eliminates the need for storing the metrics of previously examined nodes, as required by the stack algorithm; however, some nodes are visited more than once, and in this case their metric values must be recomputed. The decoder moves forward through the tree as long as the metric value along the path being examined continues to increase. When the metric value dips below a threshold the decoder backs up and begins to examine other paths. If no path can be found whose metric value stays above the threshold, the threshold is then lowered, and the decoder attempts to move forward again with a lower threshold. Each time a given node is visited in the forward direction, the threshold is lower than on the previous visit to that node. This prevents looping in the algorithm, and the decoder eventually must reach the end of the tree, at which point the algorithm terminates. When this occurs, the path that reached the end of the tree is taken as the decoded path.

A complete flowchart of the Fano algorithm is shown in Figure 13.8. The decoder starts at the origin node with the threshold $T = 0$ and the metric value $M = 0$. It then looks forward to the best of the 2^k succeeding nodes, that is, the one with the largest metric. If M_F is the metric of the forward node being examined, and if $M_F \geq T$, then the decoder moves to this node. After checking whether the end of the tree has been reached, the decoder performs a "threshold tightening" if this node is being examined for the first time; that is, T is increased by the largest multiple of a *threshold increment* Δ so that the new threshold does not exceed the current metric. If this node has been examined previously, no threshold tightening is performed. Then, the decoder again looks forward to the best succeeding node. If $M_F < T$, the decoder looks backward to the preceding node. If M_B is the metric of the backward node being examined, and if $M_B < T$, then T is lowered by Δ and the look forward to the best node step is repeated. If $M_B \geq T$, the decoder moves back to the preceding node. Call this node P. If this backward move was from the worst of the 2^k nodes succeeding node P, then the decoder again looks back to the node preceding node P. If not, the decoder looks forward to the next best of the 2^k nodes succeeding node P and checks if $M_F \geq T$. If the decoder ever looks backward from the origin node, we assume that the preceding node has a metric value of $-\infty$, so that the threshold is always lowered by Δ in this case. Ties in metric values can be resolved arbitrarily without affecting average decoder performance.

We now repeat an example worked earlier for the ZJ algorithm, this time using the Fano algorithm.

EXAMPLE 13.7 The Fano Algorithm

For the same code, metric table, and received sequence decoded by the ZJ algorithm in Example 13.5, the steps of the Fano algorithm are shown in Figure 13.9 for a value of $\Delta = 1$, and the set of nodes examined is shown in Figure 13.10. The algorithm terminates after 40 decoding steps, and the final decoded path is the same one found

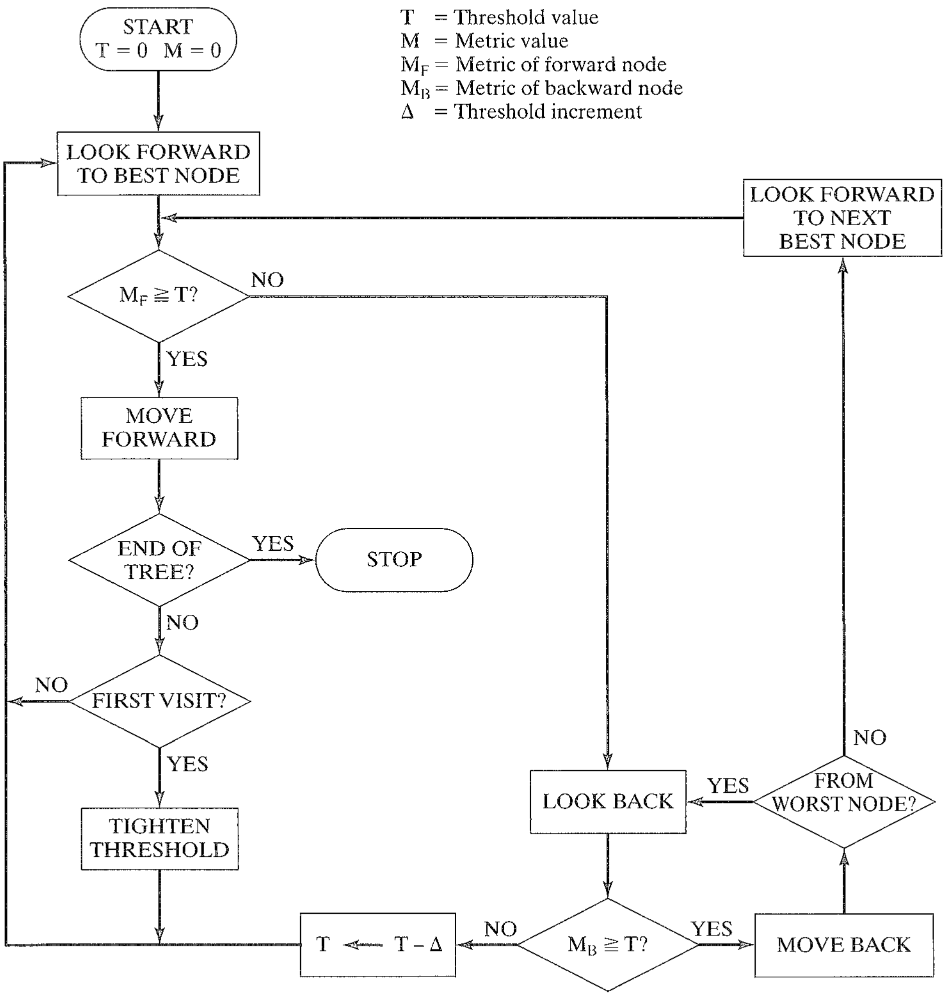


FIGURE 13.8: Flowchart for the Fano algorithm.

by both the ZJ algorithm and the Viterbi algorithm. In Figure 13.9, LFB means "look forward to best node", LFNB means "look forward to next best node", and X denotes the origin node.

A computation in the Fano algorithm is usually counted each time the look-forward step is performed. Hence, in Example 13.7 the Fano algorithm requires 40 computations, compared with only 10 for the ZJ algorithm. Note that some nodes are visited several times. In fact, the origin node is visited 8 times, the path 0 node 11 times, the path 00 node 5 times, and the nodes representing the paths 000, 1, 11, 111, 1110, 11101, 111010, and 1110100 one time each, for a total of 32

Step	Look	M_F	M_B	Node	Metric	T
0	—	—	—	X	0	0
1	LFB	−3	$-\infty$	X	0	−1
2	LFB	−3	$-\infty$	X	0	−2
3	LFB	−3	$-\infty$	X	0	−3
4	LFB	−3	—	0	−3	−3
5	LFB	−6	0	X	0	−3
6	LFNB	−9	$-\infty$	X	0	−4
7	LFB	−3	—	0	−3	−4
8	LFB	−6	0	X	0	−4
9	LFNB	−9	$-\infty$	X	0	−5
10	LFB	−3	—	0	−3	−5
11	LFB	−6	0	X	0	−5
12	LFNB	−9	$-\infty$	X	0	−6
13	LFB	−3	—	0	−3	−6
14	LFB	−6	—	00	−6	−6
15	LFB	−9	−3	0	−3	−6
16	LFNB	−12	0	X	0	−6
17	LFNB	−9	$-\infty$	X	0	−7
18	LFB	−3	—	0	−3	−7
19	LFB	−6	—	00	−6	−7
20	LFB	−9	−3	0	−3	−7
21	LFNB	−12	0	X	0	−7
22	LFNB	−9	$-\infty$	X	0	−8
23	LFB	−3	—	0	−3	−8
24	LFB	−6	—	00	−6	−8
25	LFB	−9	−3	0	−3	−8
26	LFNB	−12	0	X	0	−8
27	LFNB	−9	$-\infty$	X	0	−9
28	LFB	−3	—	0	−3	−9
29	LFB	−6	—	00	−6	−9
30	LFB	−9	—	000	−9	−9
31	LFB	−12	−6	00	−6	−9
32	LFNB	−15	−3	0	−3	−9
33	LFNB	−12	0	X	0	−9
34	LFNB	−9	—	1	−9	−9
35	LFB	−6	—	11	−6	−6
36	LFB	−3	—	111	−3	−3
37	LFB	0	—	1110	0	0
38	LFB	+3	—	11101	+3	+3
39	LFB	+6	—	111010	+6	+6
40	LFB	+9	—	1110100	+9	Stop

FIGURE 13.9: Decoding steps for the Fano algorithm with $\Delta = 1$.

nodes visited. This is fewer than the 40 computations, since not every forward look results in a move to a different node but sometimes only in a threshold lowering. In this example the threshold was lowered 8 times. It is also interesting to note that the Fano algorithm examined the same set of nodes as the ZJ algorithm in this case, as can be seen by comparing Figures 13.6(j) and 13.10.

The number of computations performed by the Fano algorithm depends on how the threshold increment Δ is selected. In general, if Δ is too small, a large

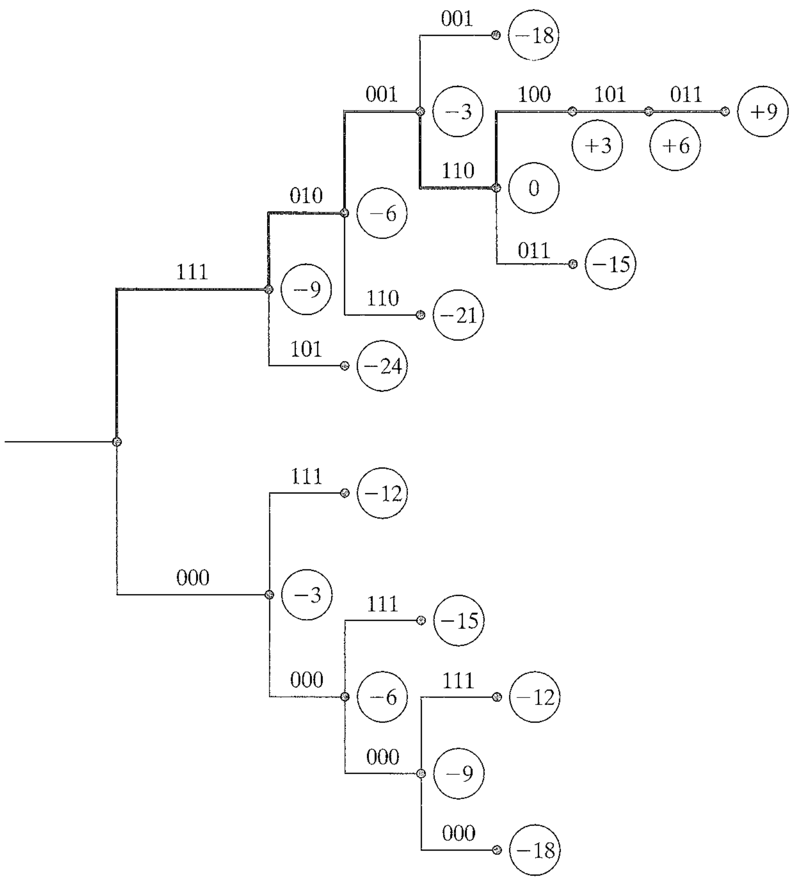


FIGURE 13.10: Set of nodes examined in Example 13.7.

number of computations results, as in the preceding example. Making Δ larger usually reduces the number of computations.

EXAMPLE 13.8 The Fano Algorithm Revisited

The Fano algorithm is repeated for the same code, metric table, and received sequence decoded in Example 13.7 and for a value of $\Delta = 3$. The results are shown in Figure 13.11. Twenty-two computations are required, 20 nodes are visited, and the same path is decoded. Hence, in this case raising Δ to 3 reduced the number of computations by almost a factor of 2.

The threshold increment Δ cannot be raised indefinitely, however, without affecting the error probability. For the algorithm to find the ML path, T must at some point be lowered below the minimum metric along the ML path. If Δ is too

Step	Look	M_F	M_B	Node	Metric	T
0	—	—	—	X	0	0
1	LFB	−3	$-\infty$	X	0	−3
2	LFB	−3	—	0	−3	−3
3	LFB	−6	0	X	0	−3
4	LFNB	−9	$-\infty$	X	0	−6
5	LFB	−3	—	0	−3	−6
6	LFB	−6	—	00	−6	−6
7	LFB	−9	−3	0	−3	−6
8	LFNB	−12	0	X	0	−6
9	LFNB	−9	$-\infty$	X	0	−9
10	LFB	−3	—	0	−3	−9
11	LFB	−6	—	00	−6	−9
12	LFB	−9	—	000	−9	−9
13	LFB	−12	−6	00	−6	−9
14	LFNB	−15	−3	0	−3	−9
15	LFNB	−12	0	X	0	−9
16	LFNB	−9	—	1	−9	−9
17	LFB	−6	—	11	−6	−6
18	LFB	−3	—	111	−3	−3
19	LFB	0	—	1110	0	0
20	LFB	+3	—	11101	+3	+3
21	LFB	+6	—	111010	+6	+6
22	LFB	+9	—	1110100	+9	Stop

FIGURE 13.11: Decoding steps for the Fano algorithm with $\Delta = 3$.

large, when T is lowered below the minimum metric of the ML path it may also be lowered below the minimum metric of several other paths, thereby making it possible for any of these to be decoded before the ML path. Making Δ too large also can cause the number of computations to increase again, since more "bad" paths can be followed further into the tree if T is lowered too much. Experience has shown that if unscaled metrics are used, Δ should be chosen between 2 and 8. If the metrics are scaled, Δ should be scaled by the same factor. In the preceding example the scaling factor was $\frac{1}{.52}$, indicating that Δ should be chosen between 3.85 and 15.38. A choice of Δ between 5 and 10 would be a good compromise in this case (see Problem 13.8).

In the preceding examples of the application of the Fano algorithm, the same path chosen by the ZJ algorithm was decoded in both cases. The Fano algorithm almost always examines the same set of nodes and decodes the same path as the ZJ algorithm (this depends somewhat on the choice of Δ [8]); however, in the Fano algorithm, some nodes are examined several times, whereas in the ZJ algorithm, nodes are examined no more than once. Thus, the Fano algorithm requires more computations than the ZJ algorithm. Because the Fano algorithm is not slowed down by stack control problems, however, it sometimes decodes faster than the ZJ algorithm. On noisier channels, though, when the Fano algorithm must do significant backsearching, the ZJ algorithm is faster [9]. Because it does not require any storage and suffers a speed disadvantage only on very noisy channels, the Fano algorithm is usually selected in practical implementations of sequential decoding.

13.3 PERFORMANCE CHARACTERISTICS OF SEQUENTIAL DECODING

The performance of sequential decoding is not as easy to characterize as Viterbi decoding because of the interplay between errors and erasures and the need to assess computational behavior. Because the number of computations performed in decoding a frame of data is a random variable, its probability distribution must be computed to determine computational performance. A great deal of work has gone into a random coding analysis of this probability distribution. Only a brief summary of these results is given here. Readers interested in more detail are referred to [10, 11, 12, and 13].

The performance of sequential decoding is determined by three quantities:

1. the *computational distribution* $Pr[C_l \geq \eta]$,
2. the *erasure probability* $P_{erasure}$, and
3. the *undetected bit-error probability* $P_b(E)$.

Let the lth *incorrect subset* of the code tree be the set of all nodes branching from the lth node on the correct path, $0 \leq l \leq h-1$, as shown in Figure 13.12, where h represents the length of the information sequence in branches. If C_l, $0 \leq l \leq h-1$, represents the number of computations performed in the lth incorrect subset, then the *computational distribution* averaged over the ensemble of all convolutional codes satisfies

$$Pr[C_l \geq \eta] \approx A\eta^{-\rho}, \quad 0 < \rho < \infty, \quad 0 \leq l \leq h-1, \tag{13.22}$$

where A is a constant depending on the particular version of sequential decoding used. Experimental studies indicate that A is typically between 1 and 10 [14, 15]. The parameter ρ is related to the encoder rate R by the parametric equation

$$R = \frac{E_0(\rho)}{\rho}, \quad 0 < R < C, \tag{13.23}$$

where C is the channel capacity in bits per transmitted symbol. The function $E_0(\rho)$ is called the *Gallager function*, and for any binary-input symmetric DMC it is given by

$$E_0(\rho) = \rho - \log_2 \frac{1}{2} \sum_j [P(j|0)^{1/(1+\rho)} + P(j|1)^{1/(1+\rho)}]^{1+\rho}, \tag{13.24}$$

where the $P(j|i)$'s are the channel transition probabilities. For the special case of a BSC,

$$E_0(\rho) = \rho - \log_2[p^{1/(1+\rho)} + (1-p)^{1/(1+\rho)}]^{1+\rho}, \tag{13.25}$$

where p is the channel transition probability. It is important to note here that the probability distribution of (13.22) depends only on the encoder rate and the channel,

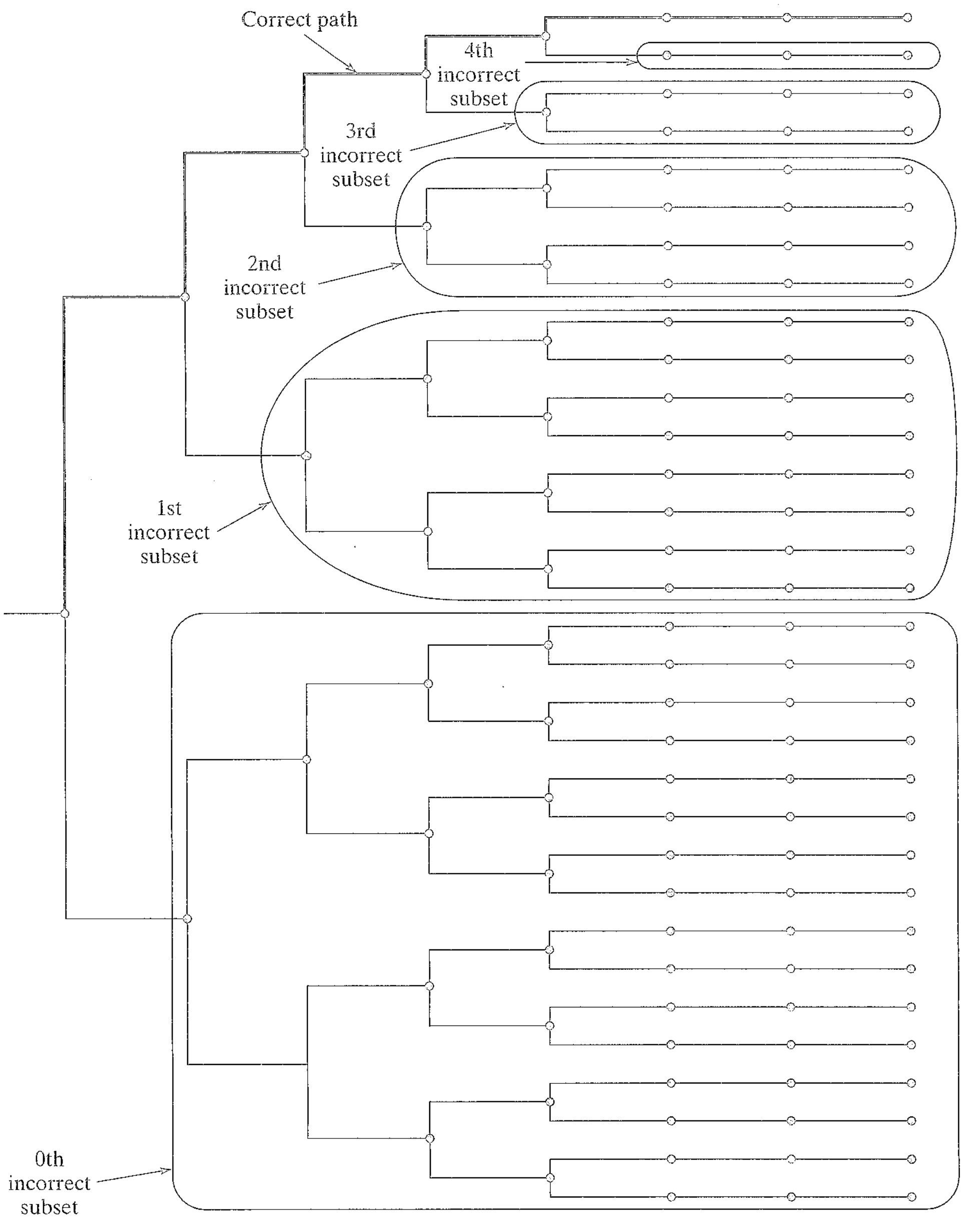


FIGURE 13.12: Incorrect subsets for a code tree with $h = 5$.

and hence the computational behavior of sequential decoding is *independent* of the code constraint length ν.

The distribution of (13.22) is a *Pareto distribution*, and ρ is called the *Pareto exponent*. ρ is an extremely important parameter for sequential decoding, since it determines how rapidly $Pr[C_l \geq \eta]$ decreases as a function of η. For example, for a fixed rate R and a BSC, ρ can be calculated by solving (13.23) using the expression

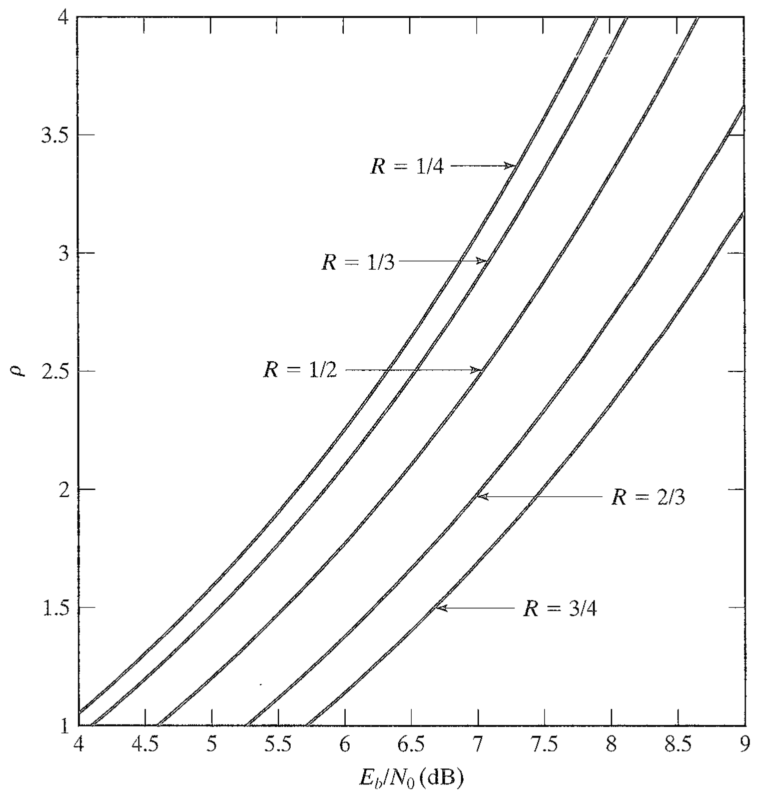


FIGURE 13.13: The Pareto exponent as a function of E_b/N_0 for a BSC.

of (13.25) for $E_0(\rho)$. Results are shown in Figure 13.13 for $R = 1/4, 1/3, 1/2, 2/3$, and $3/4$, where ρ is plotted as a function of the SNR $E_b/N_0 = E_s/RN_0$, and $p = Q(\sqrt{2E_s/N_0}) = Q(\sqrt{2RE_b/N_0})$. Note that ρ increases with increasing SNR E_b/N_0 and with decreasing rate R.

For the case when $\rho = 1$,

$$R = E_0(1) = 1 - \log_2 \frac{1}{2} \sum_j \left[\sqrt{P(j|0)} + \sqrt{P(j|1)}\right]^2, \tag{13.26}$$

and for a BSC

$$\begin{aligned} R = E_0(1) &= 1 - \log_2 \left[\sqrt{p} + \sqrt{1-p}\right]^2 \\ &= 1 - \log_2 \left[1 + 2\sqrt{p(1-p)}\right]. \end{aligned} \tag{13.27}$$

The significance of $E_0(1)$ for sequential decoding is related to the moments of the computational distribution. To compute $E[C_l^i]$, the ith moment of C_l, we first form

$$F_{C_l}(X) = Pr[C_l \le X] = 1 - Pr[C_l \ge X] = 1 - AX^{-\rho}, \tag{13.28}$$

the cumulative distribution function of C_l. Differentiating this expression we obtain the probability density function

$$f_{C_l}(X) = \frac{dF_{C_l}(X)}{dX} = \rho A X^{-\rho-1}. \tag{13.29}$$

Now, we can compute the ith moment $E[C_l^i]$ as

$$\begin{aligned} E[C_l^i] &= \int_1^{\infty} X^i f_{C_l}(X)\,dX \\ &= \int_1^{\infty} \rho A X^{i-\rho-1}\,dX \\ &= \frac{\rho A}{i-\rho} X^{i-\rho}\Big|_1^{\infty} \\ &= \lim_{X\to\infty} \frac{\rho A}{i-\rho}(X^{i-\rho}-1). \end{aligned} \tag{13.30}$$

For the ith moment to be finite, that is,

$$E[C_l^i] = \lim_{X\to\infty} \frac{\rho A}{i-\rho}(X^{i-\rho}-1) < \infty, \tag{13.31}$$

we require that $\rho > i$. In other words, if $\rho \leq 1$, the *average number of computations* per decoded branch in the lth incorrect subset is unbounded. It can be shown from (13.23) and (13.24) that $\rho \to \infty$ as $R \to 0$, and $\rho \to 0$ as $R \to C$; that is, $R = \frac{E_0(\rho)}{\rho}$ and ρ are inversely related (see Problem 13.10). Hence, the requirement that $\rho > 1$ to guarantee a bounded average number of computations is equivalent to requiring that

$$R = \frac{E_0(\rho)}{\rho} < E_0(1) \triangleq R_0, \tag{13.32}$$

where R_0 is called the *computational cutoff rate* of the channel. In other words, R must be less than R_0 for the average computational load of sequential decoding to be finite, independent of the code constraint length. For example, for a BSC, (13.27) implies that $R_0 = E_0(1) = 1/2$ when $p = .045$, and thus for a rate $R = 1/2$ encoder, $p < .045$ ensures that $R < R_0$ and the average computational load is finite. Thus, R_0 is called the computational cutoff rate because sequential decoding becomes computationally intensive for rates above R_0. Because the cutoff rate R_0 is always strictly less than the channel capacity C (see Problem 13.11), sequential decoding is suboptimum at rates between R_0 and C; however, since it can decode large constraint length codes efficiently, sequential decoding achieves excellent performance at rates below R_0.

The probability distribution of (13.22) can also be used to calculate the probability of buffer overflow, or the *erasure probability* $P_{erasure}$. Let B be the size of the input buffer in branches (nB bits), and let μ be the speed factor of the decoder; that is, the decoder can perform μ branch computations in the time required to receive one branch. Then, if more than μB computations are required in the lth incorrect subset, the buffer will overflow before the lth received branch can be discarded, even if the buffer was initially empty before the lth branch was received. From (13.22) this probability is approximately $A(\mu B)^{-\rho}$. Because there are h information branches in a frame, the erasure probability for a frame is approximated by

$$P_{erasure} \approx hA(\mu B)^{-\rho}, \tag{13.33}$$

where ρ must satisfy (13.23). Note that (13.33) is independent of the code constraint length ν.

EXAMPLE 13.9 Estimating the Erasure Probability

Assume $h = 1000$, $A = 3$, $\mu = 10$, and $B = 10^4$. For a BSC with transition probability $p = .08$, (13.27) implies that $R_0 = 3/8$. Choosing $R = 1/3 < R_0 = 3/8$, and solving for ρ from (13.23) and (13.25) gives us $\rho = 1.31$. Substituting these values into (13.33) we obtain an erasure probability of $P_{erasure} \approx 8.5 \times 10^{-4}$.

The *undetected bit-error probability* $P_b(E)$, that is, the bit-error probability excluding erasures, of sequential decoding has also been analyzed using random coding arguments [13]. The results indicate that for rates $R < R_0$, the undetected bit-error probability of sequential decoding (1) decreases exponentially with d_{free}, the free distance, (2) increases linearly with $B_{d_{free}}$, the number of nonzero information bits on all paths of weight d_{free}, and (3) is slightly suboptimum compared with ML decoding. (We note here that, in principle, erasures can be eliminated by increasing the buffer size B and/or the decoder speed factor μ.) Because ML decoding is practical only for small values of ν, and the computational behavior of sequential decoding is independent of ν, the suboptimum performance of sequential decoding can be compensated for by using codes with larger values of ν. Thus, the practical limitation of ML decoding to small values of ν means that for rates $R < R_0$, sequential decoding can achieve lower undetected error probabilities than ML decoding.

The overall performance of sequential decoding can be judged only by considering the undetected bit-error probability, the erasure probability, and the average computational load. It is possible to obtain trade-offs among these three factors by adjusting various parameters. For example, reducing the threshold increment Δ in the Fano algorithm (or the bucket quantization interval in the stack algorithm) increases $E[C_l]$ and $P_{erasure}$ but reduces $P_b(E)$, whereas increasing the size B of the input buffer reduces $P_{erasure}$ but increases $E[C_l]$ and $P_b(E)$.

The choice of metric can also affect the overall performance of sequential decoding. Although the Fano metric of (13.3) is normally selected, it is not always necessary to do so. The bias term $-R$ is chosen to achieve a reasonable balance among $P_b(E)$, $P_{erasure}$, and $E[C_l]$. For example, according to the integer metric table of Figure 13.2(b) for a rate $R = 1/3$ code and a BSC with $p = .10$, a path of length 12 that is a distance of 2 from the received sequence $\mathbf{r}$ would have a metric value

$v_l \backslash r_l$	0	1
0	+1	−3
1	−3	+1

(a)

$v_l \backslash r_l$	0	1
0	+1	−8
1	−8	+1

(b)

FIGURE 13.14: Effect of the bias term on the integer metric table.

of 0, the same as a path of length 6 that is a distance of 1 from $\mathbf{r}$. This is intuitively satisfying, since both paths contain the same percentage of transmission errors, and leads to a certain balance among errors, erasures, and decoding speed; however, if no bias term were used in the definition of $M(r_l|v_l)$, the integer metric table shown in Figure 13.14(a) would result. In this case the length-12 path would have a metric of +4, and the length-6 path a metric of +2. This "bias" in favor of the longer path would result in less searching for the ML path and hence faster decoding and fewer erasures at the expense of more errors. On the other hand, if a larger bias term, say $-1/2$, were used, the integer metric table of Figure 13.14(b) would result. The length-12 path would then have a metric of -6, and the length-6 path a metric of -3. This "bias" in favor of the shorter path would result in more searching for the ML path and hence fewer errors at the expense of more erasures and slower decoding. Therefore, although the Fano metric is optimum in the restricted sense of identifying the one path of all those examined up to some point in the algorithm that is most likely to be part of the ML path, it is not necessarily the best metric to use in all cases.

The performance results for sequential decoding mentioned so far have all been random coding results; that is, they are representative of the average performance over the ensemble of all codes. Bounds on computational behavior and error probability for specific codes have also been obtained [16]. These bounds relate code performance to specific code parameters and indicate how codes should be selected to provide the best possible performance. As such, they are analogous to the performance bounds for specific codes given in Chapter 12 for ML decoding. For a BSC with transition probability p and an (n, k, ν) code with column distance function (CDF) d_l,

$$Pr[C_l \geq \eta] < \sigma n_{l_0} e^{-\mu d_{l_0} + \phi l_0}, \tag{13.34}$$

where σ, μ, and ϕ are functions of p and R only, $l_0 \triangleq \lfloor \log_{2^k} \eta \rfloor$, $\lfloor x \rfloor$ denotes the integer part of x, n_{l_0} is the number of codewords of length $l_0 + 1$ branches with weight d_{l_0}, and R satisfies

$$R < 1 + 2p \log_2 p + (1 - 2p) \log_2(1 - p) \triangleq R_{max}. \tag{13.35}$$

The bound of (13.35) indicates a maximum rate R_{max} for which (13.34) is known to hold. The significance of (13.34) is that it shows the dependence of the distribution of computation on the code's CDF. Fast decoding speeds and low erasure probabilities require that $Pr[C_l \geq \eta]$ decrease rapidly as a function of η. From (13.34) this implies that the CDF should increase rapidly. The logarithm in the CDF's index has the effect of enhancing the significance of the initial portion of the CDF, as illustrated in Figure 13.15. The CDFs of two (2, 1, 16) codes (Code 1: $\mathbf{g}^{(0)} = (220547)$, $\mathbf{g}^{(1)} = (357231)$; Code 2: $\mathbf{g}^{(0)} = (346411)$, $\mathbf{g}^{(1)} = (357231)$) are shown in Figure 13.15(a).

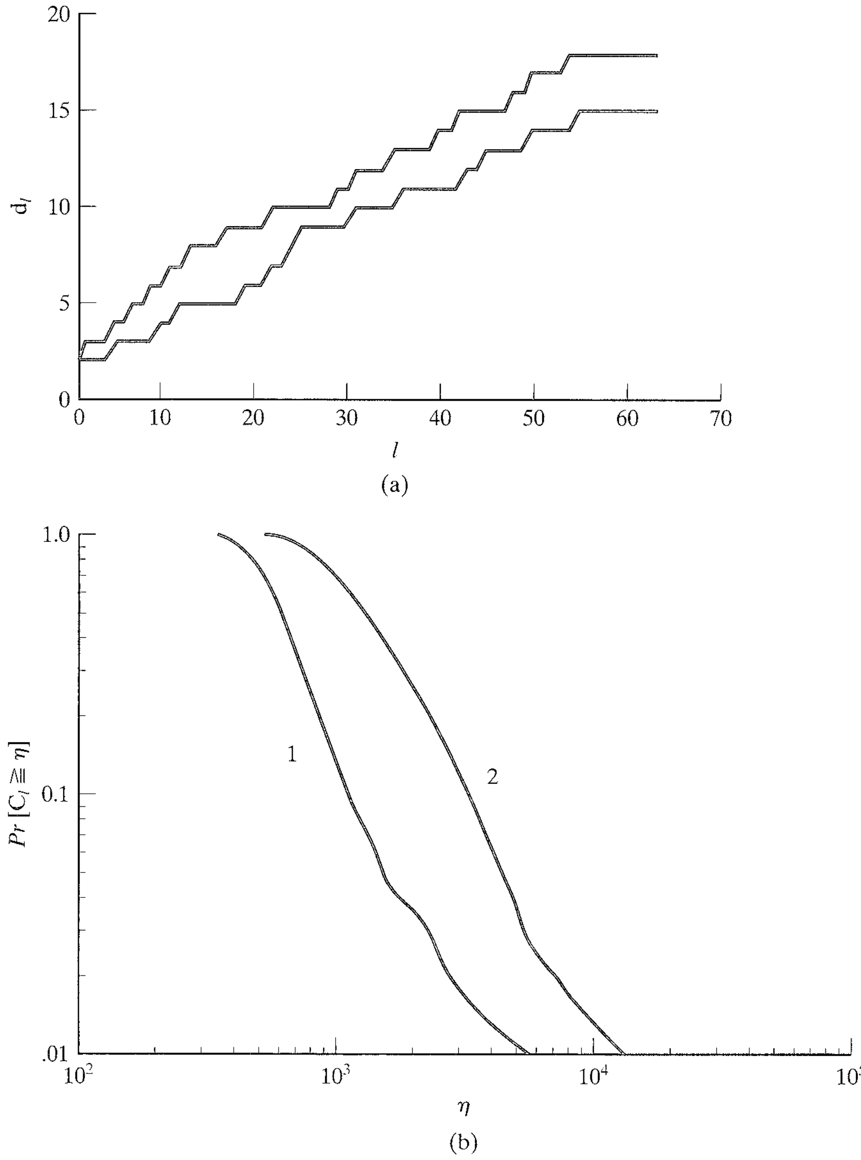


FIGURE 13.15: (a) The CDF and (b) $\Pr[C_l \geq \eta]$ for two (2, 1, 16) codes.

Their computational distributions obtained using extensive computer simulations [17] are shown in Figure 13.15(b). Note that the code with the faster column distance growth has a much better computational distribution. In fact, $E[C_l] = 3.14$ for code 1, and $E[C_l] = 7.24$ for code 2, a difference in average decoding speed of more than a factor of 2.[4]

The event- and bit-error probabilities of a specific code when used with sequential decoding are bounded by functions that decrease exponentially with d_{free}, the free distance, and increase linearly with $A_{d_{free}}$, the number of codewords with weight d_{free}, and $B_{d_{free}}$, the total information weight of those codewords, respectively[16]. This is the same general performance established in Section 12.2 for a specific code with ML decoding. In other words, the undetected bit-error probability of a code used with sequential decoding will differ little from that obtained with ML decoding, as noted previously in our discussion of random coding results.

The performance of a (2, 1, 46) systematic code ($\mathbf{g}^{(1)} = (2531746407671547)$) on a hard-decision Fano sequential decoder has been thoroughly tested in hardware by Forney and Bower [14]. The results of some of these tests are shown in Figure 13.16. Three of the figures are for an incoming data rate of 1 Mbps; the other three are for a 5 Mbps data rate. The computational rate of the decoder was 13.3 MHz, corresponding to decoder speed factors of 13.3 and 2.66, respectively. The BSC transition probability was varied between $p = .016$ and $p = .059$, representing an E_b/N_0 range from 3.9 dB to 6.65 dB (see (12.32) and (12.35)) and an R_0 range from .44 to .68 (see (13.27)). Rather than continuously vary the metric values with p, the authors chose two sets of bit metrics, $+1/-9$ and $+1/-11$. Input buffer sizes ranging from $2^{10} = 1024$ to $2^{16} = 65536$ branches were tested. Erasures were eliminated by a technique called *backsearch limiting*, in which the decoder is never allowed to move more than some maximum number J levels back from its farthest penetration into the tree. Whenever a forward move is made to a new level, the k information bits J branches back are decoded. When the backsearch limit is reached, the decoder is resynchronized by jumping forward to the most recently received branch and (for systematic encoders) accepting the received information bits on the intervening branches as decoded information. This resynchronization of the decoder typically introduces errors into the decoded sequence as the price to be paid for eliminating long searches leading to erasures.

The performance curves of Figure 13.16(a) plot bit-error probability $P_b(E)$ as a function of E_b/N_0, with buffer size as a parameter. The bit metrics were $+1/-11$, the data rate was 1 Mbps, and the backsearch limit was $J = 240$. Identical conditions held for Figure 13.16(b), except that the data rate was increased to 5 Mbps. Note that performance at a bit-error probability of 10^{-5} is as much as 0.8 dB worse in this case, indicating that the higher incoming data rate means that the decoder must jump farther ahead during a resynchronization period to catch up with the received sequence, thereby introducing additional errors. In Figures 13.16(c) and 13.16(d), the backsearch limit J is a parameter, the buffer size is 2^{16} branches, and the data rates are 1 Mbps and 5 Mbps, respectively. Note that in this case the lower data

[4]As an interesting comparison, the Viterbi algorithm would require a fixed number $2^{\nu} = 65536$ computations per decoded information bit to decode a (2, 1, 16) code.

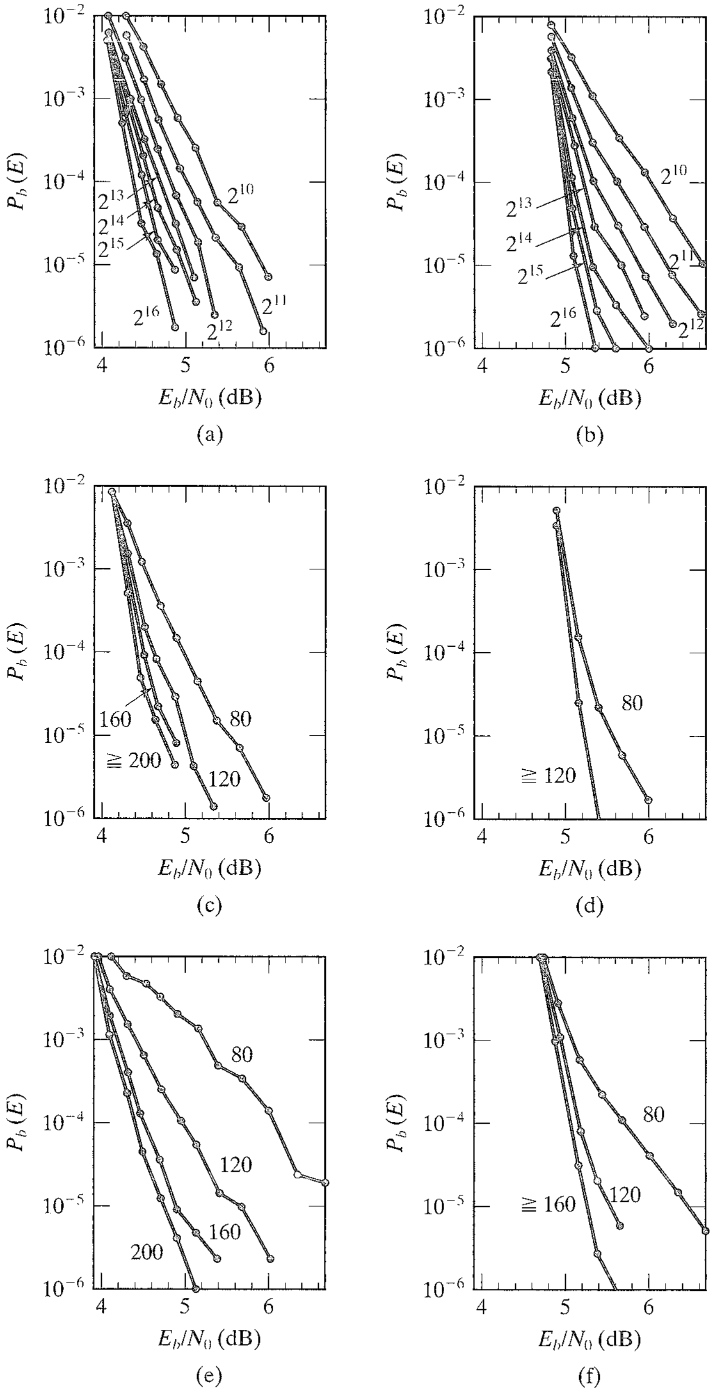


FIGURE 13.16: Test results for a (2, 1, 46) systematic code.

rate gives as much as 0.6 dB better performance if J is large enough; however, if J is too small, little improvement is obtained, indicating that the basic limitation on decoder performance is now J rather than the data rate. Figures 13.16(e) and 13.16(f) repeat the test conditions of Figures 13.16(c) and 13.16(d) with bit metrics of $+1/-9$ instead of $+1/-11$. Performance is about the same for large J but as much as 1.1 dB worse for small J. This indicates that the $+1/-9$ bit metrics generate more searching before an incorrect path is eliminated, thereby causing increased reliance on the backsearch limit to force the decoder forward. Note that values of E_b/N_0 below 4.6 dB correspond to $R_0 < R = 1/2$ and that performance does not degrade catastrophically in this range. In other words, the computational cutoff rate R_0 is obtained from an ensemble average upper bound, and although it generally indicates the maximum rate at which good performance can be obtained, it does not mean that sequential decoders can be operated only at rates $R < R_0$.

It is usually impossible to directly compare the performance of sequential decoding with that of Viterbi decoding because of the interplay between errors and erasures in a sequential decoder; however, the elimination of erasures in sequential decoding by backsearch limiting makes a rudimentary comparison possible. Figure 12.17(b) shows the performance of a hard-decision Viterbi decoder for optimum nonsystematic rate $R = 1/2$ codes with $\nu = 2$ through 7. To obtain $P_b(E) = 10^{-4}$ requires E_b/N_0 to be in the range 5.4 dB to 7.0 dB. The decoder in this case must perform 2^ν computations per received branch, which implies a decoder speed factor ranging from 4 to 128 (in the absence of parallel decoding). The decoder must also be capable of storing 2^ν 32-bit paths. Figure 13.16(a) shows the performance of a hard-decision Fano sequential decoder for a suboptimum systematic rate $R = 1/2$ code[5] with $\nu = 46$ and buffer sizes from 2^{10} through 2^{16} branches. To obtain $P_b(E) = 10^{-4}$ requires E_b/N_0 in the range 4.4 dB to 5.3 dB, a 1.0-dB to 1.7-dB improvement over the Viterbi decoder. The speed factor of the sequential decoder is 13.3, comparable to that of the Viterbi decoder. The sequential decoder memory requirement of from 2^{11} to 2^{17} bits somewhat exceeds the Viterbi decoder's requirement of from 2^7 to 2^{12} bits. To pick a specific point of comparison, consider $\nu = 5$ for the Viterbi decoder and $B = 2^{11}$ branches for the sequential decoder. At $P_b(E) = 10^{-4}$, the sequential decoder requires $E_b/N_0 = 5.0$ dB compared with the Viterbi algorithm's requirement of $E_b/N_0 = 6.0$ dB, a 1.0-dB advantage for the sequential decoder. The Viterbi decoder requires a speed factor of 32, roughly 2.5 times that of the sequential decoder, whereas the sequential decoder's memory requirement of 2^{12} bits is four times that of the Viterbi decoder. It is well to note here that this comparison ignores the different hardware needed to perform a "computation" in the two algorithms, as well as the differing amounts of control logic required. In addition, when parallel decoding architectures are considered, the balance tends to shift toward Viterbi decoding. A thorough discussion of the comparison between sequential and Viterbi decoding is given in [18].

[5]Optimum codes of this length are unknown. Because the computational behavior of sequential decoding is independent of ν, systematic codes with large ν are preferred to simplify decoder resynchronization. Nonsystematic codes with the same d_{free} and smaller values of ν would perform approximately the same.

Similar to backsearch limiting, an alternative version of sequential decoding, the *multiple stack algorithm*, introduced by Chevillat and Costello [19], eliminates erasures entirely, and thus its performance can be compared directly with that of the Viterbi algorithm. The multiple stack algorithm begins exactly like the ordinary stack algorithm, except that the stack size is limited to a fixed number of entries Z_1. Starting with the origin node, the top node in the stack is extended. After its elimination from the stack, the successors are inserted and the stack is ordered in the usual way. If decoding is completed before the stack fills up, the multiple stack algorithm outputs exactly the same decoding decision as the ordinary stack algorithm; however, if the received sequence is one of those few that requires extended searches, that is, a potential erasure, the stack fills up. In this case, the best T paths in the stack are transferred to a smaller second stack with $Z << Z_1$ entries. Decoding then proceeds in the second stack using only these T transferred nodes. If the best path in the second stack reaches the end of the tree before the stack fills up, this path is stored as a tentative decision. The decoder then deletes the remaining paths in the second stack and returns to the first stack, where decoding continues. If the decoder reaches the end of the tree before the first stack fills up again, the metric of this path is compared with that of the tentative decision. The path with the better metric is retained and becomes the final decoding decision; however, if the first stack fills up again, a new second stack is formed by again transferring the best T paths from the first stack. If the second stack fills up also, a third stack of size Z is formed by transferring the best T paths from the second stack. Additional stacks of the same size are formed in a similar manner until a tentative decision is made. The decoder always compares each new tentative decision with the previous one and retains the path with the best metric. The rest of the paths in the current stack are then deleted and decoding proceeds in the previous stack. The algorithm terminates decoding if it reaches the end of the tree in the first stack. The only other way decoding can be completed is by exceeding a computational limit C_{lim}. In this case the best tentative decision becomes the final decoding decision. A complete flowchart of the multiple stack algorithm is shown in Figure 13.17.

The general philosophy of the multiple stack algorithm emerges from its use of additional stacks. The first stack is made large enough so that only very noisy codewords, that is, potential erasures, force the use of additional stacks. Rather than follow the strategy of the ordinary stack algorithm, which advances slowly in these noisy cases because it is forced to explore many incorrect subsets before extending the correct path, the multiple stack algorithm advances quickly through the tree and finds a reasonably good tentative decision. Only then does it explore in detail all the alternatives in search of the ML path; that is, previous stacks are revisited. This change in priorities is achieved by making the additional stacks substantially smaller than the first one ($Z << Z_1$). Because the T paths at the top of a full stack are almost always farther into the tree than the T paths used to initialize the stack, the creation of each new stack forces the decoder farther ahead until the end of the tree is reached. Hence, if C_{lim} is not too small, at least one tentative decision is always made, and erasures are eliminated. Once a tentative decision is made, the decoder returns to previous stacks trying to improve its final decision.

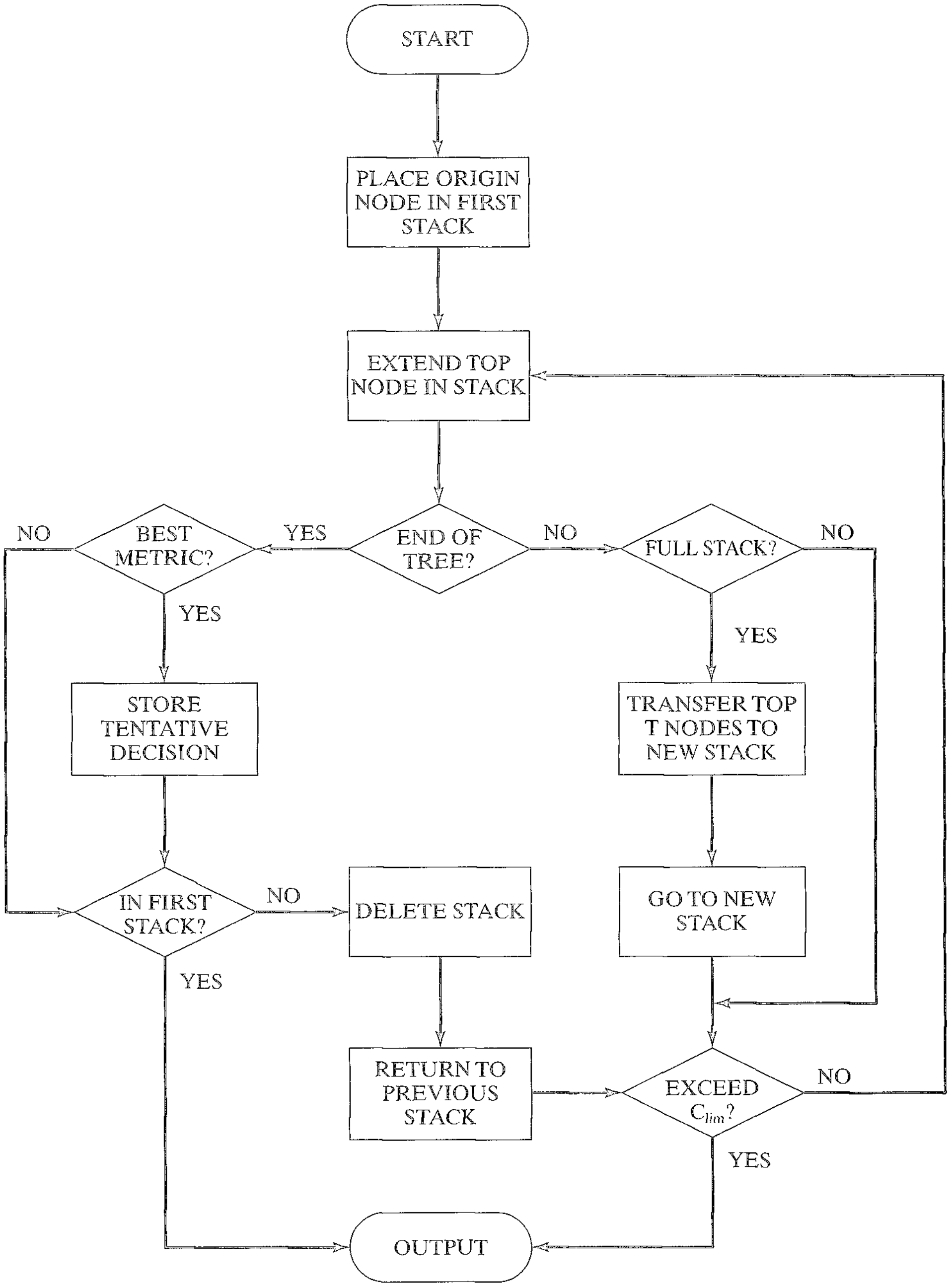


FIGURE 13.17: Flowchart of the multiple stack algorithm.

The performance of the multiple stack algorithm can be compared directly to the Viterbi algorithm, since it is erasure-free. As with ordinary sequential decoding, the multiple stack algorithm's computational effort is independent of code constraint length, and hence it can be used with longer codes than can the Viterbi algorithm. The performance comparison is thus made by comparing decoding speed and implementation complexity at comparable error probabilities rather than for equal constraint lengths. The performance of the Viterbi algorithm on a BSC with the best (2, 1, 7) code (curve 1) is compared with that of the multiple stack algorithm with four different sets of parameters (curves 2, 3, 4, and 5) in Figure 13.18. Curve 2 is for a (2, 1, 12) code with $Z_1 = 1365$, $Z = 11$, $T = 3$, and $C_{\text{lim}} = 6144$. Note that at $E_b/N_0 = 5.5$ dB both decoders achieve a bit-error probability of approximately 7×10^{-5}; however, the Viterbi algorithm requires 128 computations per decoded

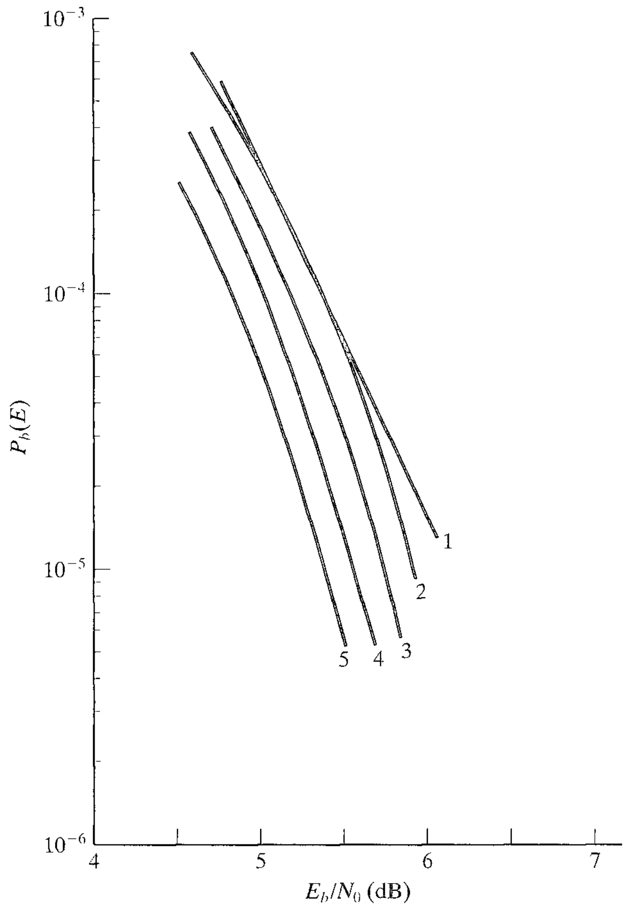


FIGURE 13.18: Performance comparison of the Viterbi algorithm and the multiple stack algorithm on a BSC.

information bit, almost two orders of magnitude larger than the average number of computations $E[C_l] = 1.37$ per decoded bit for the multiple stack algorithm. Because $L = 60$ bit information sequences were used in the simulations, even the computational limit $C_{\text{lim}} = 6144$ of the multiple stack algorithm is smaller than the constant number of almost $60 \times 128 = 7680$ computations that the Viterbi algorithm executes per received sequence. The multiple stack algorithm's stack requires approximately 1640 entries, compared with 128 for the Viterbi algorithm. Increasing the size of the first stack to $Z_1 = 2048$ (curve 3) lowers the bit-error probability to about 3.5×10^{-5} at 5.5 dB with no change in computational effort and a stack storage requirement of about 2400 entries. Use of the multiple stack algorithm (curve 4) with a $(2, 1, 15)$ code, $Z_1 = 3413$, $Z = 11$, $T = 3$, and $C_{\text{lim}} = 8192$ achieves $P_b(E) = 1.5 \times 10^{-5}$ at 5.5 dB with $E[C_l] = 1.41$ and a total stack storage of about 3700. Increasing the first stack size to $Z_1 = 4778$ (curve 5) yields $P_b(E) = 7 \times 10^{-6}$ at 5.5 dB with $E[C_l] = 1.42$ and a total storage of about 5000. In this last example the multiple stack algorithm's bit-error probability is 10 times smaller, and its average computational load about 90 times smaller than that of a Viterbi decoder for the $(2, 1, 7)$ code.

Whether the large differences in computational load noted result in an equally pronounced decoding speed advantage for the multiple stack algorithm depends primarily on the execution time per computation. A trellis node extension for the Viterbi algorithm is usually somewhat faster, since the ordering of the stacks in the multiple stack algorithm is time-consuming with conventionally addressed stacks; however, ordering time can be substantially reduced by using the stack-bucket technique. In view of the large differences in computational load, it thus seems justified to conclude that the multiple stack algorithm decodes considerably faster than the Viterbi algorithm, at least in the absence of parallel processing. The multiple stack algorithm's speed advantage is achieved at the expense of considerable stack and buffer storage, but this seems to be tolerable in view of the rapid progress in large-scale storage devices. The multiple stack algorithm is therefore an attractive alternative to Viterbi decoding when low error probabilities are required at high decoding speeds.

In an effort to extend the decoder cutoff rate, that is, the rate at which the average computational load of the decoder becomes infinite, to rates above R_0, Falconer [20] and Jelinek and Cocke [21] have introduced hybrid sequential and algebraic decoding schemes. The idea is to place algebraic constraints across several frames of convolutionally encoded data. In Falconer's scheme, a certain number of frames are sequentially decoded, with the remainder being determined by the algebraic constraints. This increases overall decoding speed and raises the "effective R_0" of the channel. In the Jelinek and Cocke scheme, each successfully decoded frame is used along with the algebraic constraints in a bootstrapping approach to provide updated estimates of the channel transition probabilities used to compute the bit metrics for the sequential decoder. For the BSC this means that the effective transition probabilities of the channel are altered after each successful decoding. Upper and lower bounds obtained on the "effective R_0" of this bootstrap hybrid decoder indicate further gains over Falconer's scheme, indicating that less information about the state of the channel is wasted. These schemes offer an improvement in the performance of sequential decoding at the cost of some increase in the complexity of implementation.

Haccoun and Ferguson [22] have attempted to close the gap between Viterbi and sequential decoding by considering them as special cases of a more general decoding algorithm called the *generalized stack algorithm*. In this algorithm, paths in an ordered stack are extended as in the stack algorithm, but more than one path can be extended at the same time, and remergers are exploited as in the Viterbi algorithm to eliminate redundant paths from the stack. Analysis and simulation have shown that the variability of the computational distribution is reduced compared with the ordinary stack algorithm, at a cost of a larger average number of computations. The error probability also more closely approaches that of the Viterbi algorithm. The complexity of implementation is somewhat increased, however, over the ordinary stack algorithm.

Another approach to improving the computational performance of sequential decoding has been proposed by de Paepe, Vinck, and Schalkwijk [23]. They have identified a special subclass of rate $R = 1/2$ codes for which a modified stack decoder can achieve a considerable savings in computation and storage by exploiting symmetries in the code. Because these codes are suboptimal, longer constraint lengths are needed to obtain the free distance required for a certain error probability, but this does not affect the computational behavior of the decoder.

13.4 CODE CONSTRUCTION FOR SEQUENTIAL DECODING

The performance results of the previous section can be used to select good codes for use with sequential decoding. In particular, good sequential decoding performance requires rapid initial column distance growth for fast decoding and a low erasure probability, and a large free distance d_{free} and a small number of nearest neighbors $A_{d_{free}}$ for a low undetected error probability. The *distance profile* of a code is defined as its CDF over only the first constraint length, that is, $d_0, d_1, \cdots, d_\nu$. Because the distance profile determines the initial column distance growth and is easier to compute than the entire CDF, it is often used instead of the CDF as a criterion for selecting codes for use with sequential decoding.

A code is said to have a distance profile $d_0, d_1, \cdots, d_\nu$ *superior* to the distance profile $d'_0, d'_1, \cdots, d'_\nu$ of another code of the same constraint length ν if for some $t, 0 \leq t \leq \nu$,

$$d_l \begin{cases} = d'_l, & l = 0, 1, \cdots, t-1 \\ > d'_l, & l = t. \end{cases} \tag{13.36}$$

In other words, the initial portion of the CDF determines which code has the superior distance profile. A code is said to have an *optimum distance profile* (ODP) if its distance profile is superior to that of any other code of the same rate and constraint length.

Because an ODP guarantees fast initial column distance growth, ODP codes with large d_{free} and small $A_{d_{free}}$ make excellent choices for sequential decoding. A list of ODP codes with rates $R = 1/3$, 1/2, and 2/3, generated by both systematic and nonsystematic feedforward encoders, is given in Table 13.1, along with the corresponding values of d_{free} and $A_{d_{free}}$. The computer search algorithms used to construct these codes are described in references [24–30]. Note that d_{free} for the systematic encoders considerably lags d_{free} for the nonsystematic encoders, since, as noted in Chapter 12, more free distance is available with nonsystematic feedforward encoders of a given rate and constraint length than with systematic feedforward encoders. Systematic feedforward encoders are listed, since, as noted previously, they have desirable decoder synchronization properties. This advantage is obtained with essentially no penalty in sequential decoding, since computational behavior is independent of ν, and the deficiency in d_{free} can be overcome simply by choosing codes with larger values of ν. This is not possible with Viterbi or BCJR decoding, where a severe penalty in reduced d_{free} or increased ν (and hence increased computation) is paid for using systematic feedforward encoders.

TABLE 13.1(a): Rate $R = 1/3$ systematic codes with an optimum distance profile.

ν	$\mathbf{g}^{(1)}$	$\mathbf{g}^{(2)}$	d_{free}	$A_{d_{free}}$
1	3	3	5	1
2	5	7	6	1
3	13	17	8	2
4	35	27	9	1
5	75	67	10	1
6	135	157	12	4
7	345	373	12	1
8	465	373	13	1
9	1465	1373	15	3
10	2465	3373	16	4
11	7465	7373	16	1
12	10653	17247	17	1
13	30653	37247	18	1
14	50653	67247	19	2
15	137653	143247	20	2
16	302465	273373	20	1
17	550653	767247	22	2
18	1751145	1545267	24	6
19	3750653	3067247	24	2
20	7150653	5767247	26	4
21	14437653	15043247	26	3
22	36051145	33245267	26	1
23	77150653	65767247	28	3
24	142651145	164645267	28	1
25	274437653	335043247	30	2
26	636051145	573245267	31	2
27	1033437653	1323043247	32	10
28	3033437653	1323043247	33	5
29	7064702465	4275673373	33	3
30	15602651145	13564645267	34	1

Adapted from [30].

TABLE 13.1(b): Rate $R = 1/3$ nonsystematic codes with an optimum distance profile.

ν	$\mathbf{g}^{(0)}$	$\mathbf{g}^{(1)}$	$\mathbf{g}^{(2)}$	d_{free}	$A_{d_{free}}$
1	1	3	3	5	1
2	5	7	7	8	2
3	15	13	17	10	3
4	25	33	37	12	5
5	71	65	57	13	1
6	175	133	117	14	1
7	365	353	227	16	1
8	561	325	747	17	1
9	1735	1063	1257	20	7
10	3645	2133	3347	21	4
11	6531	5615	7523	22	3
12	13471	15275	10637	24	2
13	32671	27643	22617	26	7
14	47371	51255	74263	27	6
15	151711	167263	134337	28	1
16	166255	321143	227277	30	3
17	764255	473143	662277	32	7
18	1070551	1663123	1274677	34	28
19	3624655	2754543	1274677	35	8

Adapted from [30].

TABLE 13.1(c): Rate $R = 1/2$ systematic codes with an optimum distance profile.

ν	$\mathbf{g}^{(1)}$	d_{free}	$A_{d_{free}}$
1	3	3	1
2	7	4	2
3	13	4	1
4	33	5	2
5	67	6	3
6	173	6	1
7	147	6	2
8	473	7	1
9	1547	8	4
10	2547	8	3
11	6547	9	3
12	14473	9	1
13	34473	10	5
14	71147	10	4
15	174473	10	1

TABLE 13.1(c): (*continued*)

ν	$g^{(1)}$	d_{free}	$A_{d_{free}}$
16	334473	12	13
17	334473	12	13
18	1514473	12	4
19	3431147	12	3
20	4371147	12	1
21	14371147	12	1
22	33431147	14	6
23	61454473	14	2
24	153431147	15	5
25	366514473	15	3
26	650371147	16	8
27	1250371147	16	7
28	3353431147	16	3
29	5650371147	18	22
30	13061514473	16	1
31	33061514473	18	11

Adapted from [30].

TABLE 13.1(d): Rate $R = 1/2$ nonsystematic codes with an optimum distance profile.

ν	$g^{(0)}$	$g^{(1)}$	d_{free}	$A_{d_{free}}$
1	3	1	3	1
2	7	5	5	1
3	17	15	6	1
4	23	35	7	2
5	77	51	8	2
6	163	135	10	12
7	323	275	10	1
8	457	755	12	10
9	1337	1475	12	1
10	2457	2355	14	19
11	6133	5745	14	1
12	17663	11271	15	2
13	26651	36477	16	5
14	46253	77361	17	3
15	114727	176121	18	5
16	330747	207225	19	9
17	507517	654315	20	6
18	1342063	1551635	21	13

(*continued overleaf*)

TABLE 13.1(d): (*continued*)

ν	$\mathbf{g}^{(0)}$	$\mathbf{g}^{(1)}$	d_{free}	$A_{d_{free}}$
19	2132245	3235467	22	26
20	7015767	5106461	22	1
21	15302573	11147605	24	40
22	23065513	36766521	24	4
23	75437055	45640557	26	65
24	116765117	143303271	26	10
25	344410533	277032345	27	24

Adapted from [30].

TABLE 13.1(e): Rate $R = 2/3$ systematic codes with an optimum distance profile.

ν	$\mathbb{h}^{(2)}$	$\mathbb{h}^{(1)}$	d_{free}	$A_{d_{free}}$
1	3	3	2	1
2	5	7	3	2
3	15	13	4	7
4	35	23	4	2
5	75	63	5	6
6	155	107	5	2
7	133	217	6	24
8	533	617	6	5
9	1055	1307	6	1
10	3675	2263	7	8
11	3675	6263	8	44
12	14133	16617	8	16
13	27675	32263	8	3
14	57455	62207	8	2
15	170133	116617	8	1
16	270133	316617	10	40
17	647675	552263	10	15
18	1504133	1344617	10	18
19	2631455	3276207	10	2
20	6631455	5276207	11	8

Adapted from [30].

We conclude our discussion of sequential decoding by noting that a sequential decoder can be used to find the distance properties of a code. For example, Forney [31] describes how the Fano algorithm can be modified to compute the CDF of a code, and Chevillat [32] gives a version of the ZJ algorithm that can be used to compute the CDF. More recently, Cedervall and Johannesson [28] introduced an algorithm called FAST, also based on the ZJ algorithm, for computing the free distance d_{free} and the first several low-weight terms $A_{d_{free}}, A_{d_{free}+1}, A_{d_{free}+2}, \cdots$ in the IOWEF $A(W, X)$ of an encoder. These methods are usually feasible for computing

TABLE 13.1(f): Rate $R = 2/3$ nonsystematic codes with an optimum distance profile.

ν	$\mathbb{h}^{(2)}$	$\mathbb{h}^{(1)}$	$\mathbb{h}^{(0)}$	d_{free}	$A_{d_{free}}$
2	3	5	7	3	1
3	17	15	13	4	1
4	05	23	27	5	7
5	53	51	65	6	9
6	121	113	137	6	1
7	271	257	265	7	6
8	563	601	475	8	8
9	1405	1631	1333	8	1
10	1347	3641	3415	9	9
11	5575	7377	4033	10	29
12	14107	13125	11203	10	4
13	20535	31637	27773	11	9
14	66147	41265	57443	12	58
15	100033	167575	155377	12	25
16	353431	300007	267063	12	7
17	631115	405661	352543	13	18
18	1111441	1516615	1202547	14	50
19	2454743	3525715	2004061	14	24
20	4200057	7262173	5122341	14	15
21	15213427	12632451	14223343	15	14
22	34122155	26656021	30523603	16	50
23	72634151	53760245	44400637	16	10

Adapted from [30].

the distance properties of codes with constraint lengths ν on the order of 30 or less. For larger values of ν, computationally efficient algorithms are still lacking.

13.5 MAJORITY-LOGIC DECODING

We begin our discussion of majority-logic decoding by considering a $(2, 1, m)$ systematic feedforward encoder with generator sequences $\mathbb{g}^{(0)} = (100\cdots)$ and $\mathbb{g}^{(1)} = (g_0^{(1)}, g_1^{(1)}, g_2^{(1)}, \cdots, g_m^{(1)})$ and generator matrix

$$\mathbb{G} = \begin{bmatrix} 1\,g_0^{(1)} & 0\,g_1^{(1)} & 0\,g_2^{(1)} & \cdots & 0\,g_m^{(1)} & & \\ & 1\,g_0^{(1)} & 0\,g_1^{(1)} & \cdots & 0\,g_{m-1}^{(1)} & 0\,g_m^{(1)} & \\ & & 1\,g_0^{(1)} & \cdots & 0\,g_{m-2}^{(1)} & 0\,g_{m-1}^{(1)} & 0\,g_m^{(1)} \\ & & \ddots & & & & \ddots \end{bmatrix}. \quad (13.37)$$

For an information sequence $\mathbb{u}$, the encoding equations are given by

$$\mathbb{v}^{(0)} = \mathbb{u} \circledast \mathbb{g}^{(0)} = \mathbb{u} \quad (13.38a)$$

$$\mathbb{v}^{(1)} = \mathbb{u} \circledast \mathbb{g}^{(1)}, \quad (13.38b)$$

and the transmitted codeword is $\mathbf{v} = \mathbf{u}\mathbf{G}$. Next, we develop a version of *syndrome decoding*, first introduced in Chapter 3 for block codes, that is appropriate for convolutional codes. If $\mathbf{v}$ is sent over a binary symmetric channel (BSC), the binary received sequence $\mathbf{r}$ can be written as

$$\mathbf{r} = (r_0^{(0)} r_0^{(1)}, r_1^{(0)} r_1^{(1)}, r_2^{(0)} r_2^{(1)}, \cdots) = \mathbf{v} + \mathbf{e}, \tag{13.39}$$

where the binary sequence $\mathbf{e} = (e_0^{(0)} e_0^{(1)}, e_1^{(0)} e_1^{(1)}, e_2^{(0)} e_2^{(1)}, \cdots)$ is called the *channel error sequence*. An error bit $e_l^{(j)} = 1$ if and only if $r_l^{(j)} \neq v_l^{(j)}$; that is, an error occurred during transmission. A simple model for the BSC is illustrated in Figure 13.19. The received sequence $\mathbf{r}$ can be divided into a *received information sequence*

$$\mathbf{r}^{(0)} = (r_0^{(0)}, r_1^{(0)}, r_2^{(0)}, \cdots) = \mathbf{v}^{(0)} + \mathbf{e}^{(0)} = \mathbf{u} + \mathbf{e}^{(0)} \tag{13.40a}$$

and a *received parity sequence*

$$\mathbf{r}^{(1)} = (r_0^{(1)}, r_1^{(1)}, r_2^{(1)}, \cdots) = \mathbf{v}^{(1)} + \mathbf{e}^{(1)} = \mathbf{u} \circledast \mathbf{g}^{(1)} + \mathbf{e}^{(1)}, \tag{13.40b}$$

where $\mathbf{e}^{(0)} = (e_0^{(0)}, e_1^{(0)}, e_2^{(0)}, \cdots)$ is the *information error sequence*, and $\mathbf{e}^{(1)} = (e_0^{(1)}, e_1^{(1)}, e_2^{(1)}, \cdots)$ is the *parity error sequence*.

The *syndrome sequence* $\mathbf{s} = (s_0, s_1, s_2, \cdots)$ is defined as

$$\mathbf{s} \triangleq \mathbf{r}\mathbb{H}^T, \tag{13.41}$$

where the *parity-check matrix* $\mathbb{H}$ is given by

$$\mathbb{H} = \begin{bmatrix}
g_0^{(1)} & 1 & & & & & & & & & & & & \\
g_1^{(1)} & 0 & g_0^{(1)} & 1 & & & & & & & & & & \\
g_2^{(1)} & 0 & g_1^{(1)} & 0 & g_0^{(1)} & 1 & & & & & & & & \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots & & & & & & & & \\
g_m^{(1)} & 0 & g_{m-1}^{(1)} & 0 & g_{m-2}^{(1)} & 0 & \cdots & g_0^{(1)} & 1 & & & & & \\
 & & g_m^{(1)} & 0 & g_{m-1}^{(1)} & 0 & \cdots & g_1^{(1)} & 0 & g_0^{(1)} & 1 & & & \\
 & & & & g_m^{(1)} & 0 & \cdots & g_2^{(1)} & 0 & g_1^{(1)} & 0 & g_0^{(1)} & 1 & \\
 & & & & & \ddots & & & & & & & & \ddots
\end{bmatrix}. \tag{13.42}$$

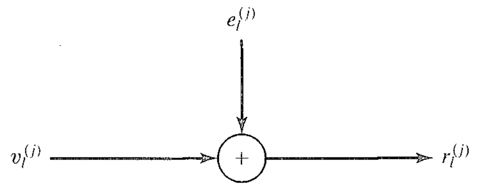


FIGURE 13.19: A simplified model for the BSC.

As is the case with block codes, $\mathbb{G}\mathbb{H}^T = \mathbb{0}$, and $\mathbb{v}$ is a codeword if and only if $\mathbb{v}\mathbb{H}^T = \mathbb{0}$; however, unlike with block codes, $\mathbb{G}$ and $\mathbb{H}$ are semi-infinite matrices, indicating that for convolutional codes the information sequences and codewords are of arbitrary length.

Because $\mathbb{r} = \mathbb{v} + \mathbb{e}$, we can write the syndrome sequence $\mathbb{s}$ as

$$\mathbb{s} = (\mathbb{v} + \mathbb{e})\mathbb{H}^T = \mathbb{v}\mathbb{H}^T + \mathbb{e}\mathbb{H}^T = \mathbb{e}\mathbb{H}^T. \tag{13.43}$$

and we see that $\mathbb{s}$ depends only on the channel error sequence and not on the particular codeword transmitted. For decoding purposes, knowing $\mathbb{s}$ is equivalent to knowing $\mathbb{r}$, and hence the decoder can be designed to operate on $\mathbb{s}$ rather than on $\mathbb{r}$. Such a decoder is called a *syndrome decoder*.

Using the polynomial notation introduced in Chapter 11, we can write

$$\mathbb{v}^{(0)}(D) = \mathbb{u}(D)\mathbb{g}^{(0)}(D) = \mathbb{u}(D) \tag{13.44a}$$

$$\mathbb{v}^{(1)}(D) = \mathbb{u}(D)\mathbb{g}^{(1)}(D), \tag{13.44b}$$

and

$$\mathbb{r}^{(0)}(D) = \mathbb{v}^{(0)}(D) + \mathbb{e}^{(0)}(D) = \mathbb{u}(D) + \mathbb{e}^{(0)}(D) \tag{13.45a}$$

$$\mathbb{r}^{(1)}(D) = \mathbb{v}^{(1)}(D) + \mathbb{e}^{(1)}(D) = \mathbb{u}(D)\mathbb{g}^{(1)}(D) + \mathbb{e}^{(1)}(D). \tag{13.45b}$$

At the receiver, the syndrome sequence is formed as

$$\mathbb{s}(D) = \mathbb{r}^{(0)}(D)\mathbb{g}^{(1)}(D) + \mathbb{r}^{(1)}(D). \tag{13.46}$$

Because $\mathbb{v}^{(1)}(D) = \mathbb{u}(D)\mathbb{g}^{(1)}(D) = \mathbb{v}^{(0)}(D)\mathbb{g}^{(1)}(D)$, forming the syndrome is equivalent to "encoding" $\mathbb{r}^{(0)}(D)$ and then adding it to $\mathbb{r}^{(1)}(D)$. A block diagram of a syndrome former for a $(2, 1, m)$ systematic feedforward encoder is shown in Figure 13.20. Using (13.45) in (13.46) we obtain

$$\begin{aligned}\mathbb{s}(D) &= [\mathbb{u}(D) + \mathbb{e}^{(0)}(D)]\mathbb{g}^{(1)}(D) + \mathbb{u}(D)\mathbb{g}^{(1)}(D) + \mathbb{e}^{(1)}(D)\\ &= \mathbb{e}^{(0)}(D)\mathbb{g}^{(1)}(D) + \mathbb{e}^{(1)}(D),\end{aligned} \tag{13.47}$$

and we again see that $\mathbb{s}(D)$ depends only on the channel error sequence and not on the particular codeword transmitted.

Majority-logic decoding of convolutional codes as a BSC is based on the concept of orthogonal parity-check sums, first introduced in Chapter 8. From (13.43) and (13.47) we see that any syndrome bit, or any sum of syndrome bits, represents a known sum of channel error bits and thus is called a *parity-check sum*, or simply a *check-sum*. If the received sequence is a codeword, then the channel error sequence is also a codeword, and all syndrome bits, and hence all check-sums, must be zero; however, if the received sequence is not a codeword, some check-sums will not be zero. An error bit e_l is said to be *checked* by a check-sum if e_l is included in the sum. A set of J check-sums is *orthogonal* on e_l if each check-sum checks e_l, but no other error bit is checked by more than one check-sum. Given a set of J orthogonal check-sums on an error bit e_l, the *majority-logic decoding rule* can be used to estimate the value of e_l.

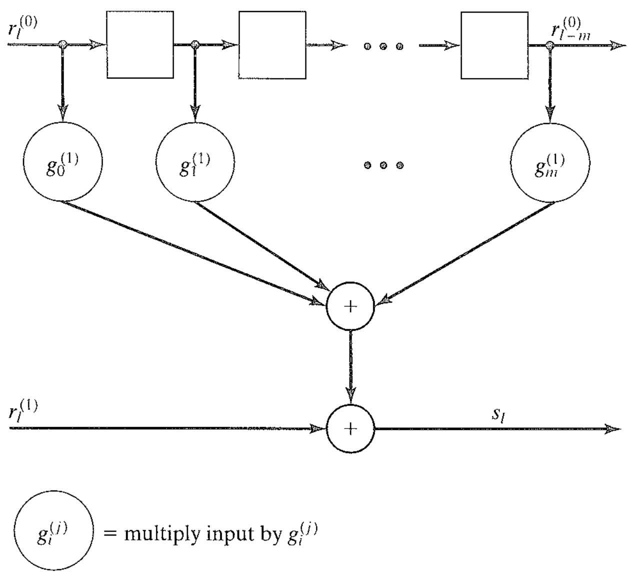


FIGURE 13.20: Syndrome forming circuit for a (2, 1, m) systematic feedforward encoder.

Majority-Logic Decoding Rule (BSC) Define $t_{ML} \triangleq \lfloor J/2 \rfloor$. Choose the estimate $\hat{e}_l = 1$ if and only if more than t_{ML} of the J check-sums orthogonal on e_l have value 1.

Theorem 13.1 If the error bits checked by the J orthogonal check-sums contain t_{ML} or fewer channel errors, the majority-logic decoding rule correctly estimates e_l.

Proof. If $e_l = 0$, the at most t_{ML} errors can cause at most t_{ML} of the J check-sums to have a value of 1. Hence, $\hat{e}_l = 0$, which is correct. On the other hand, if $e_l = 1$, the at most $t_{ML} - 1$ other errors can cause no more than $t_{ML} - 1$ of the J check-sums to have value 0, so that at least $t_{ML} + 1$ will have a value of 1. Hence, $\hat{e}_l = 1$, which is again correct. **Q.E.D.**

As a consequence of Theorem 13.1, for a systematic code with J orthogonal check-sums on each information error bit, t_{ML} is called the *majority-logic error-correcting capability* of the code.

EXAMPLE 13.10 A Rate $R = 1/2$ Self-Orthogonal Code

Consider finding a set of orthogonal check-sums on $e_0^{(0)}$, the first information error bit, for the (2, 1, 6) systematic code with $\mathbf{g}^{(1)}(D) = 1 + D + D^4 + D^6$. First, note from (13.43) and (13.47) that $e_0^{(0)}$ can affect only syndrome bits s_0 through s_6, that is, the first $(m+1)$ syndrome bits. Letting $[\mathbf{s}]_6 = (s_0, s_1, \cdots, s_6)$ and using the notation for truncated sequences and matrices introduced in Section 11.3, we obtain

$$[\mathbb{s}]_6 = [\mathbb{e}]_6[\mathbb{H}^T]_6 = [\mathbb{e}]_6 \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ & 1 & 1 & 0 & 0 & 1 & 0 \\ & 1 & 0 & 0 & 0 & 0 & 0 \\ & & 1 & 1 & 0 & 0 & 1 \\ & & 1 & 0 & 0 & 0 & 0 \\ & & & 1 & 1 & 0 & 0 \\ & & & 1 & 0 & 0 & 0 \\ & & & & 1 & 1 & 0 \\ & & & & 1 & 0 & 0 \\ & & & & & 1 & 1 \\ & & & & & 1 & 0 \\ & & & & & & 1 \\ & & & & & & 1 \end{bmatrix}. \tag{13.48}$$

Taking the transpose of both sides of (13.48), we have

$$[\mathbb{s}^T]_6 = [\mathbb{H}]_6[\mathbb{e}^T]_6 = \begin{bmatrix} 1 & 1 & & & & & & & & & & & & \\ 1 & 0 & 1 & 1 & & & & & & & & & & \\ 0 & 0 & 1 & 0 & 1 & 1 & & & & & & & & \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & & & & & & \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & & & & \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & & \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} [\mathbb{e}^T]_6. \tag{13.49}$$

Note that the even-numbered columns of $[\mathbb{H}]_6$, that is, those columns corresponding to the parity error sequence, form an identity matrix. Hence, we can rewrite (13.49) as

$$[\mathbb{s}^T]_6 = \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \end{bmatrix} = \begin{bmatrix} 1 & & & & & & \\ 1 & 1 & & & & & \\ 0 & 1 & 1 & & & & \\ 0 & 0 & 1 & 1 & & & \\ 1 & 0 & 0 & 1 & 1 & & \\ 0 & 1 & 0 & 0 & 1 & 1 & \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} e_0^{(0)} \\ e_1^{(0)} \\ e_2^{(0)} \\ e_3^{(0)} \\ e_4^{(0)} \\ e_5^{(0)} \\ e_6^{(0)} \end{bmatrix} + \begin{bmatrix} e_0^{(1)} \\ e_1^{(1)} \\ e_2^{(1)} \\ e_3^{(1)} \\ e_4^{(1)} \\ e_5^{(1)} \\ e_6^{(1)} \end{bmatrix}. \tag{13.50}$$

The matrix in (13.50) that multiplies the information error sequence is called the *parity triangle* of the code. Note that the first column of the parity triangle is the generator sequence $\mathbb{g}^{(1)}$, which is shifted down by one and truncated in each succeeding column.

We can now use the parity triangle of the code to select a set of orthogonal check sums on $e_0^{(0)}$. First, note that no syndrome bit can be used in more than one orthogonal check-sum, since then a parity error bit would be checked more than once. Because there are only four syndrome bits that check $e_0^{(0)}$, it is not possible to obtain more than $J = 4$ orthogonal check-sums on $e_0^{(0)}$ in this example. We can illustrate the orthogonal check-sums selected using the parity triangle as follows:

$$
\begin{array}{cccccccc}
\rightarrow & 1 & & & & & & \\
\rightarrow & 1 & \boxed{1} & & & & & \\
 & 0 & 0 & 1 & & & & \\
 & 0 & 0 & 1 & 1 & & & \\
\rightarrow & 1 & 0 & 0 & \boxed{1} & \boxed{1} & & \\
 & 0 & 1 & 0 & 0 & 1 & 1 & \\
\rightarrow & 1 & 0 & \boxed{1} & 0 & 0 & \boxed{1} & \boxed{1}\ .
\end{array}
$$

The arrows indicate the syndrome bits, or sums of syndrome bits, that are selected as orthogonal check sums on $e_0^{(0)}$, and the boxes indicate which information error bits, other than $e_0^{(0)}$, are checked. Clearly, at most one arrow can point to each row, and at most one box can appear in each column. The equations for the orthogonal check-sums are

$$
\begin{array}{llllll}
s_0 = e_0^{(0)} & & & & +e_0^{(1)} & \\
s_1 = e_0^{(0)} + e_1^{(0)} & & & & +e_1^{(1)} & \\
s_4 = e_0^{(0)} & & +e_3^{(0)} + e_4^{(0)} & & +e_4^{(1)} & \\
s_6 = e_0^{(0)} & +e_2^{(0)} & & +e_5^{(0)} + e_6^{(0)} & +e_6^{(1)}. &
\end{array}
\qquad (13.51)
$$

Note that $e_0^{(0)}$ appears in each check-sum, but no other error bit appears more than once. Because each check-sum is a single syndrome bit, and not a sum of syndrome bits, this is called a *self-orthogonal code*. These codes are discussed in detail in Section 13.7. Because a total of 11 different channel error bits are checked by the $J = 4$ orthogonal check-sums of (13.51), the majority-logic decoding rule will correctly estimate $e_0^{(0)}$ whenever $t_{ML} = 2$ or fewer of these 11 error bits are 1's (channel errors). The total number of channel error bits checked by the orthogonal check-sum equations is called the *effective decoding length* n_E of the code. Hence, $n_E = 11$ for this $(2, 1, 6)$ code.

EXAMPLE 13.11 A Rate $R = 1/2$ Orthogonalizable Code

Consider the $(2, 1, 5)$ systematic code with $\mathbf{g}^{(1)}(D) = 1 + D^3 + D^4 + D^5$. We can construct a set of $J = 4$ orthogonal check-sums on $e_0^{(0)}$ from the parity triangle as follows:

$$\begin{array}{llllllll}
\longrightarrow & 1 & & & & & & \\
\longrightarrow & 0 & 1 & & & & & \\
 & 0 & 0 & 1 & & & & \\
\longrightarrow & 1 & 0 & 0 & \boxed{1} & & & \\
\longrightarrow & 1 & \boxed{1} & 0 & 0 & \boxed{1} & & \\
\longrightarrow & 1 & 1 & \boxed{1} & 0 & 0 & \boxed{1} & .
\end{array}$$

The syndrome bits s_1 and s_5 must be added to eliminate the effect of $e_1^{(0)}$, which is already checked by s_4, and hence the code is not self-orthogonal. The $J = 4$ check-sums s_0, s_3, s_4, and $s_5 + s_1$ form an orthogonal set on $e_0^{(0)}$, however, so this is called an *orthogonalizable code*. Methods for constructing orthogonalizable codes are discussed in Section 13.7. The effective decoding length $n_E = 11$ for this code, and the majority-logic decoding rule correctly estimates $e_0^{(0)}$ whenever $t_{ML} = 2$ or fewer of these 11 error bits are 1's.

A majority-logic decoder must be capable of estimating not only $e_0^{(0)}$ but all the other information error bits also. Here, $e_0^{(0)}$ is estimated from the first $(m+1)$ syndrome bits s_0 through s_m. After $e_0^{(0)}$ is estimated, it is subtracted (added modulo-2) from each syndrome equation it affects to form a *modified syndrome* set $s'_0, s'_1, \cdots, s'_m$. The modified syndrome bits $s'_1, s'_2, \cdots, s'_m$ along with the newly calculated syndrome bit s_{m+1} are then used to estimate $e_1^{(0)}$. Assuming $e_0^{(0)}$ was correctly estimated, that is, $\hat{e}_0^{(0)} = e_0^{(0)}$, a set of orthogonal check sums can be formed on $e_1^{(0)}$ that is identical to those used to estimate $e_0^{(0)}$, and the same decoding rule can therefore be applied. Each successive information error bit is estimated in the same way. The most recent estimate is used to modify the syndrome, one new syndrome bit is calculated, the set of orthogonal check-sums is formed, and the majority-logic decoding rule is applied.

EXAMPLE 13.10 (Continued)

Assume $e_0^{(0)}$ has been correctly estimated; that is, $\hat{e}_0^{(0)} = e_0^{(0)}$. If it is subtracted from each syndrome equation it affects, then the modified syndrome equations for the self-orthogonal code of Example 13.10 become

$$\begin{array}{lllllllllllllllll}
s'_0 & = & s_0 - e_0^{(0)} & = & & & & & & & & & & & & & e_0^{(1)} \\
s'_1 & = & s_1 - e_0^{(0)} & = & e_1^{(0)} & & & & & & & & & & & + & e_1^{(1)} \\
s'_2 & = & s_2 & = & e_1^{(0)} & + & e_2^{(0)} & & & & & & & & & + & e_2^{(1)} \\
s'_3 & = & s_3 & = & & & e_2^{(0)} & + & e_3^{(0)} & & & & & & & + & e_3^{(1)} \\
s'_4 & = & s_4 - e_0^{(0)} & = & & & & & e_3^{(0)} & + & e_4^{(0)} & & & & & + & e_4^{(1)} \\
s'_5 & = & s_5 & = & e_1^{(0)} & & & & & + & e_4^{(0)} & + & e_5^{(0)} & & & + & e_5^{(1)} \\
s'_6 & = & s_6 - e_0^{(0)} & = & & & e_2^{(0)} & & & & & + & e_5^{(0)} & + & e_6^{(0)} & + & e_6^{(1)} .
\end{array} \qquad (13.52)$$

Because s_0' no longer checks any information error bits, it is of no use in estimating $e_1^{(0)}$. The syndrome bit s_7, however, checks $e_1^{(0)}$. Hence, the modified syndrome bits s_1' through s_6' along with s_7 can be used to estimate $e_1^{(0)}$. The equations are

$$\begin{bmatrix} s_1' \\ s_2' \\ s_3' \\ s_4' \\ s_5' \\ s_6' \\ s_7 \end{bmatrix} = \begin{bmatrix} 1 & & & & & & \\ 1 & 1 & & & & & \\ 0 & 1 & 1 & & & & \\ 0 & 0 & 1 & 1 & & & \\ 1 & 0 & 0 & 1 & 1 & & \\ 0 & 1 & 0 & 0 & 1 & 1 & \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} e_1^{(0)} \\ e_2^{(0)} \\ e_3^{(0)} \\ e_4^{(0)} \\ e_5^{(0)} \\ e_6^{(0)} \\ e_7^{(0)} \end{bmatrix} + \begin{bmatrix} e_1^{(1)} \\ e_2^{(1)} \\ e_3^{(1)} \\ e_4^{(1)} \\ e_5^{(1)} \\ e_6^{(1)} \\ e_7^{(1)} \end{bmatrix}. \tag{13.53}$$

Because the parity triangle is unchanged from (13.50), the syndrome bits s_1', s_2', s_5', and s_7 form a set of $J = 4$ orthogonal check-sums on $e_1^{(0)}$ as follows:

$$\begin{aligned} s_1' &= e_1^{(0)} & & & & +e_1^{(1)} \\ s_2' &= e_1^{(0)} + e_2^{(0)} & & & & +e_2^{(1)} \\ s_5' &= e_1^{(0)} & & +e_4^{(0)} + e_5^{(0)} & & +e_5^{(1)} \\ s_7 &= e_1^{(0)} & +e_3^{(0)} & & +e_6^{(0)} + e_7^{(0)} & +e_7^{(1)}. \end{aligned} \tag{13.54}$$

From this orthogonal set, $e_1^{(0)}$ will be correctly estimated if there are $t_{ML} = 2$ or fewer errors among the $n_E = 11$ error bits checked by (13.54). In other words, the majority-logic error-correcting capability of the code is the same for $e_1^{(0)}$ as for $e_0^{(0)}$, assuming that $e_0^{(0)}$ was estimated correctly. Moreover, exactly the same decoding rule can be used to estimate $e_1^{(0)}$ as was used for $e_0^{(0)}$, since the check-sum equations (13.54) are identical to (13.51), except that different error bits are checked. This means that after estimating $e_0^{(0)}$, the implementation of the decoding circuitry need not be changed to estimate $e_1^{(0)}$.

In general, after an information error bit is estimated, it is subtracted from each syndrome equation it affects. Assuming the estimate is correct, the syndrome equations

$$\begin{aligned} s_l' &= e_l^{(0)} & & & & +e_l^{(1)} \\ s_{l+1}' &= e_l^{(0)} \quad +e_{l+1}^{(0)} & & & & +e_{l+1}^{(1)} \\ s_{l+4}' &= e_l^{(0)} & & +e_{l+3}^{(0)} + e_{l+4}^{(0)} & & +e_{l+4}^{(1)} \\ s_{l+6} &= e_l^{(0)} & +e_{l+2}^{(0)} & & +e_{l+5}^{(0)} + e_{l+6}^{(0)} & +e_{l+6}^{(1)} \end{aligned} \tag{13.55}$$

form a set of $J = 4$ orthogonal check-sums on $e_l^{(0)}$, and the same decoding rule can be used to estimate $e_l^{(0)}, l = 0, 1, \cdots$. Hence, $e_l^{(0)}$ will be correctly estimated if there are $t_{ML} = 2$ or fewer errors among the $n_E = 11$ error bits checked by (13.55), $l = 0, 1, \cdots$.

A complete encoder/decoder block diagram for the $(2, 1, 6)$ self-orthogonal code of Example 13.10 with majority-logic decoding is shown in Figure 13.21. The decoder operates as follows:

Step 1. The first $(m+1)$ syndrome bits $s_0, s_1, \cdots, s_6$ are calculated.

Step 2. A set of $J = 4$ orthogonal check-sums on $e_0^{(0)}$ is formed from the syndrome bits calculated in step 1.

Step 3. The four check-sums are fed into a majority gate that produces an output of 1 if and only if three or four (more than half) of its inputs are 1's. If its output is 0 ($\hat{e}_0^{(0)} = 0$), $r_0^{(0)}$ is assumed to be correct. If its output is 1 ($\hat{e}_0^{(0)} = 1$), $r_0^{(0)}$ is assumed to be incorrect, and hence it must be corrected. The correction is performed by adding the output of the majority gate ($\hat{e}_0^{(0)}$) to $r_0^{(0)}$. The output of the threshold gate ($\hat{e}_0^{(0)}$) is also fed back and subtracted from each syndrome bit it affects. (It is not necessary to subtract $\hat{e}_0^{(0)}$ from s_0, since this syndrome bit is not used in any future estimates.)

Step 4. The estimated information bit $\hat{u}_0 = r_0^{(0)} + \hat{e}_0^{(0)}$ is shifted out of the decoder. The syndrome register is shifted once to the right, the next block of received bits ($r_7^{(0)}$ and $r_7^{(1)}$) is shifted into the decoder, and the next syndrome bit s_7 is calculated and shifted into the leftmost stage of the syndrome register.

Step 5. The syndrome register now contains the modified syndrome bits $s_1', s_2', \cdots, s_6'$ along with s_7. The decoder repeats steps 2, 3, and 4 and estimates $e_1^{(0)}$. All successive information error bits are then estimated in the same way.

Because each estimate must be fed back to modify the syndrome register before the next estimate can be made, this is called a *feedback decoder*. The process is analogous to the feedback of each estimate in a Meggitt decoder for cyclic codes. Note that each estimate made by a feedback majority-logic decoder depends on only $(m+1)$ error blocks, the effect of previously estimated error bits having been removed by the feedback. As will be seen in Section 13.6, this fact leads to inferior performance compared with optimum decoding methods. The small ($m+1$ time units) decoding delay and the simplicity of the decoder, however, make majority-logic decoding preferable to Viterbi, BCJR, or sequential decoding in some applications.

In the general case of an (n, k, m) systematic feedforward encoder, the (time-domain) generator matrix $\mathbb{G}$ is given by (11.42) and (11.43), and the (time-domain) parity-check matrix $\mathbb{H}$ is given by (11.45). The $(n-k)$ syndrome sequences, one corresponding to each parity sequence, are given by

$$\mathbb{s} = (s_0^{(k)} \cdots s_0^{(n-1)}, s_1^{(k)} \cdots s_1^{(n-1)}, s_2^{(k)} \cdots s_2^{(n-1)}, \cdots) = \mathbb{r}\mathbb{H}^T = \mathbb{e}\mathbb{H}^T. \qquad (13.56)$$

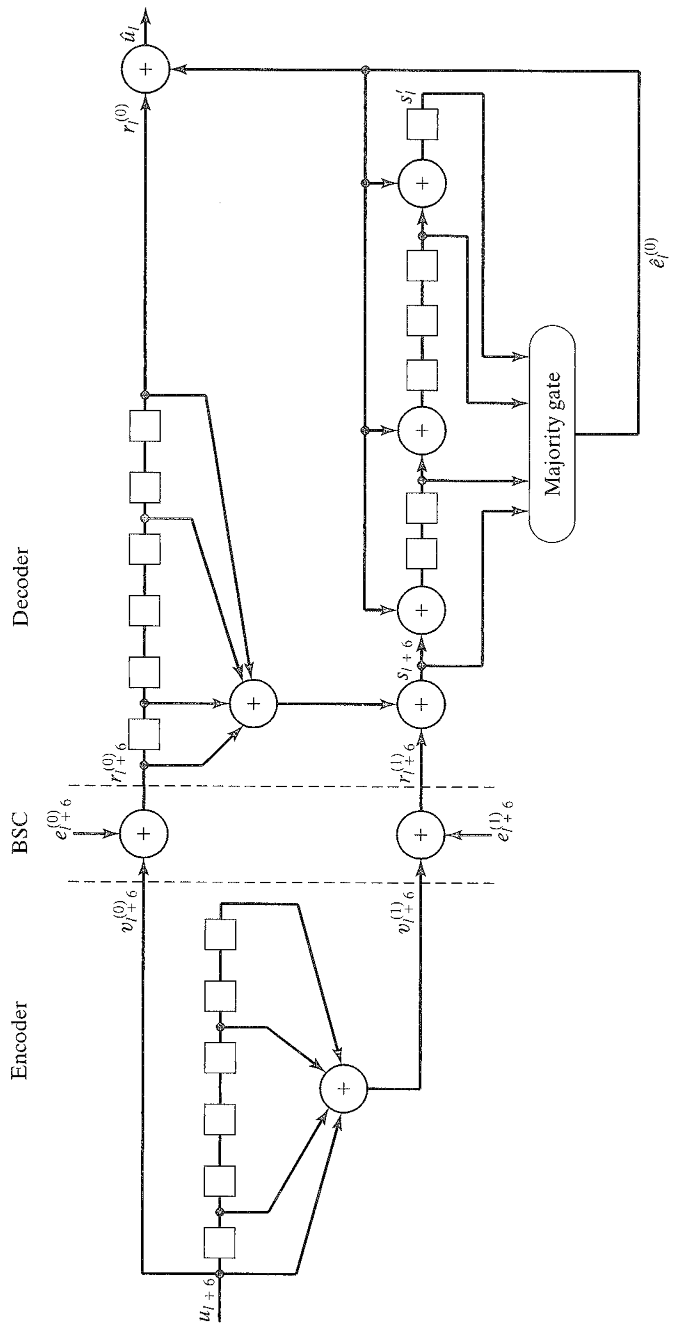


FIGURE 13.21: Complete system block diagram for a (2, 1, 6) self-orthogonal code with majority-logic decoding.

In polynomial notation, for $j = k, \cdots, n-1$,

$$\begin{aligned} \mathbb{s}^{(j)}(D) &= \sum_{i=1}^{k} \mathbb{r}^{(i-1)}(D)\mathbb{g}_i^{(j)}(D) + \mathbb{r}^{(j)}(D) \\ &= \sum_{i=1}^{k} \mathbb{e}^{(i-1)}(D)\mathbb{g}_i^{(j)}(D) + \mathbb{e}^{(j)}(D), \end{aligned} \tag{13.57}$$

and forming the syndrome vector $\mathbb{s}(D) = [\mathbb{s}^{(k)}(D), \cdots, \mathbb{s}^{(n-1)}(D)]$ is equivalent to "encoding" the received information sequences and then adding each resulting parity sequence to the corresponding received parity sequence. A block diagram of a syndrome former for an (n, k, m) systematic feedforward encoder is shown in Figure 13.22. We can put (13.57) in matrix form as follows:

$$\mathbb{s}(D) = \mathbb{r}(D)\mathbb{H}^T(D) = \mathbb{e}(D)\mathbb{H}^T(D), \tag{13.58}$$

where $\mathbb{r}(D) = [\mathbb{r}^{(0)}(D), \mathbb{r}^{(1)}(D), \cdots, \mathbb{r}^{(n-1)}(D)]$ is the n-tuple of received sequences, $\mathbb{e}(D) = [\mathbb{e}^{(0)}(D), \mathbb{e}^{(1)}(D), \cdots, \mathbb{e}^{(n-1)}(D)]$ is the n-tuple of error sequences, and the $(n-k) \times n$ (transform domain) parity-check matrix is given by (11.46a).

In this case, there are $n-k$ new syndrome bits to be formed and k information error bits to be estimated at each time unit, and there are a total of $k(n-k)$ parity

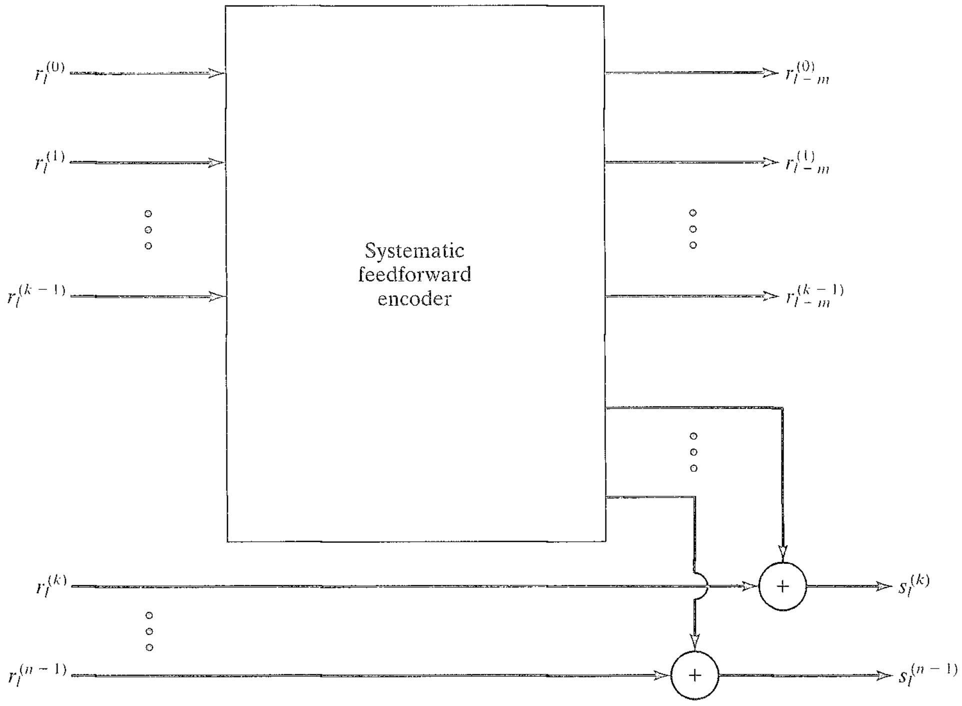


FIGURE 13.22: Syndrome forming circuit for an (n, k, m) systematic feedforward encoder.

triangles, one corresponding to each generator polynomial. The general structure of the parity triangles used to form orthogonal parity checks is as follows:

$$\begin{bmatrix} s_0^{(k)} \\ \vdots \\ s_m^{(k)} \\ s_0^{(k+1)} \\ \vdots \\ s_m^{(k+1)} \\ \vdots \\ s_0^{(n-1)} \\ \vdots \\ s_m^{(n-1)} \end{bmatrix} = \begin{bmatrix} g_{1,0}^{(k)} & & & g_{2,0}^{(k)} & & & & g_{k,0}^{(k)} & & \\ \vdots & \ddots & & \vdots & \ddots & & & \vdots & \ddots & \\ g_{1,m}^{(k)} & \cdots & g_{1,0}^{(k)} & g_{2,m}^{(k)} & \cdots & g_{2,0}^{(k)} & \cdots & g_{k,m}^{(k)} & \cdots & g_{k,0}^{(k)} \\ g_{1,0}^{(k+1)} & & & g_{2,0}^{(k+1)} & & & & g_{k,0}^{(k+1)} & & \\ \vdots & \ddots & & \vdots & \ddots & & & \vdots & \ddots & \\ g_{1,m}^{(k+1)} & \cdots & g_{1,0}^{(k+1)} & g_{2,m}^{(k+1)} & \cdots & g_{2,0}^{(k+1)} & \cdots & g_{k,m}^{(k+1)} & \cdots & g_{k,0}^{(k+1)} \\ \vdots & & & \vdots & & & & \vdots & & \\ g_{1,0}^{(n-1)} & & & g_{2,0}^{(n-1)} & & & & g_{k,0}^{(n-1)} & & \\ \vdots & \ddots & & \vdots & \ddots & & & \vdots & \ddots & \\ g_{1,m}^{(n-1)} & \cdots & g_{1,0}^{(n-1)} & g_{2,m}^{(n-1)} & \cdots & g_{2,0}^{(n-1)} & \cdots & g_{k,m}^{(n-1)} & \cdots & g_{k,0}^{(n-1)} \end{bmatrix} \begin{bmatrix} e_0^{(0)} \\ \vdots \\ e_m^{(0)} \\ e_0^{(1)} \\ \vdots \\ e_m^{(1)} \\ \vdots \\ e_0^{(k-1)} \\ \vdots \\ e_m^{(k-1)} \end{bmatrix} + \begin{bmatrix} e_0^{(k)} \\ \vdots \\ e_m^{(k)} \\ e_0^{(k+1)} \\ \vdots \\ e_m^{(k+1)} \\ \vdots \\ e_0^{(n-1)} \\ \vdots \\ e_m^{(n-1)} \end{bmatrix}. \tag{13.59}$$

The first set of $m+1$ rows corresponds to syndrome sequence $\mathbf{s}^{(k)}$, the second set of $m+1$ rows corresponds to syndrome sequence $\mathbf{s}^{(k+1)}$, and so on. Syndrome bits, or sums of syndrome bits, are then used to form orthogonal check-sums on the information error bits $e_0^{(0)}, e_0^{(1)}, \cdots, e_0^{(k-1)}$. If at least J orthogonal check-sums can be formed on each of these information error bits, then $t_{ML} = \lfloor J/2 \rfloor$ is the majority-logic error-correcting capability of the code; that is, any pattern of t_{ML} or fewer errors within the n_E error bits checked by the k sets of orthogonal check-sums will be corrected by the majority-logic decoding rule. A feedback majority-logic decoder for a t_{ML}-error-correcting (n, k, m) systematic code operates as follows:

Step 1. The first $(m+1)(n-k)$ syndrome bits are calculated.

Step 2. A set of J orthogonal check-sums are formed on each of the k information error bits from the syndrome bits calculated in step 1.

Step 3. Each set of J check-sums is fed into a majority gate that produces an output of 1 if and only if more than half of its inputs are 1's. If its output is 0 ($\hat{e}_0^{(j)} = 0$), $r_0^{(j)}$ is assumed to be correct. If the output of the jth gate is 1 ($\hat{e}_0^{(j)} = 1$), $r_0^{(j)}$ is assumed to be incorrect, and hence it must be corrected. The corrections are performed by adding the output of each majority gate to the corresponding received bit. The output of each majority gate is also fed back and subtracted from each syndrome it affects. (It is not necessary to modify the time unit 0 syndrome bits.)

Step 4. The estimated information bits $\hat{u}_0^{(j+1)} = r_0^{(j)} + \hat{e}_0^{(j)}$, $j = 0, 1, \cdots, k-1$, are shifted out of the decoder. The syndrome registers are shifted once to the right, the next block of n received bits is shifted into the decoder, and the next set of $n-k$ syndrome bits is calculated and shifted into the leftmost stages of the $n-k$ syndrome registers.

Step 5. The syndrome registers now contain the modified syndrome bits along with the new set of syndrome bits. The decoder repeats steps

2, 3, and 4 and estimates the next block of information error bits $e_1^{(0)}, e_1^{(1)}, \cdots, e_1^{(k-1)}$. All successive blocks of information error bits are then estimated in the same way.

As noted earlier, if all previous sets of k estimates are correct, their effect is removed by the feedback, and the next set of k estimates depends on only $(m+1)$ error blocks. A block diagram of a general majority-logic feedback decoder for an (n, k, m) systematic code is shown in Figure 13.23.

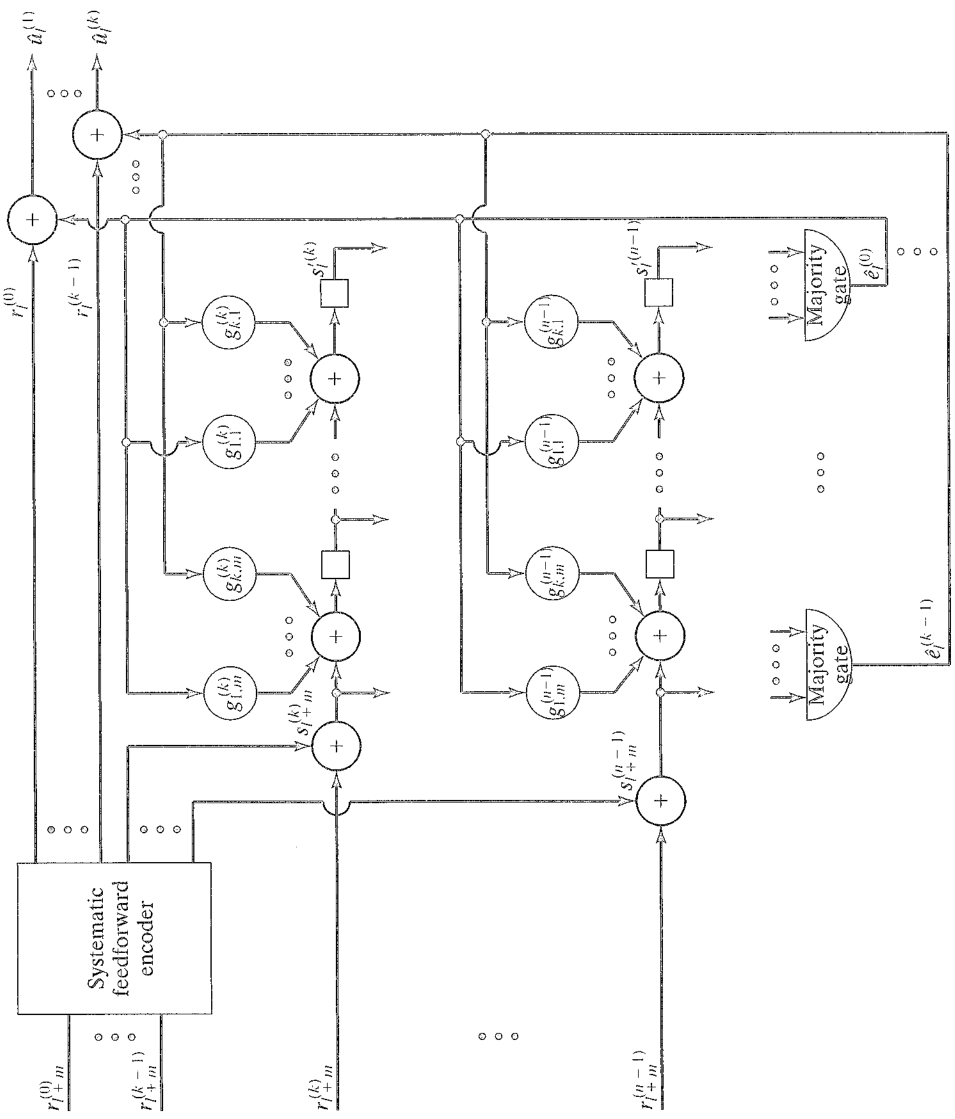


FIGURE 13.23: Complete majority-logic decoder for an (n, k, m) systematic code.

EXAMPLE 13.12 A Rate $R = 2/3$ Self-Orthogonal Code

Consider the $(3, 2, 13)$ systematic code with $g_1^{(2)}(D) = 1 + D^8 + D^9 + D^{12}$ and $g_2^{(2)}(D) = 1 + D^6 + D^{11} + D^{13}$. Following the procedure of Example 13.10, we can write the first $(m + 1) = 14$ syndrome bits as

$$[\mathbf{s}^T]_{13} = \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \\ s_7 \\ s_8 \\ s_9 \\ s_{10} \\ s_{11} \\ s_{12} \\ s_{13} \end{bmatrix} = \begin{bmatrix}
1 &&&&&&&&&&&&& \\
0 & 1 &&&&&&&&&&&& \\
0 & 0 & 1 &&&&&&&&&&& \\
0 & 0 & 0 & 1 &&&&&&&&&& \\
0 & 0 & 0 & 0 & 1 &&&&&&&&& \\
0 & 0 & 0 & 0 & 0 & 1 &&&&&&&& \\
0 & 0 & 0 & 0 & 0 & 0 & 1 &&&&&&& \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 &&&&&& \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 &&&&& \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 &&&& \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 &&& \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 && \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \\
0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix} e_0^{(0)} \\ e_1^{(0)} \\ e_2^{(0)} \\ e_3^{(0)} \\ e_4^{(0)} \\ e_5^{(0)} \\ e_6^{(0)} \\ e_7^{(0)} \\ e_8^{(0)} \\ e_9^{(0)} \\ e_{10}^{(0)} \\ e_{11}^{(0)} \\ e_{12}^{0)} \\ e_{13}^{(0)} \end{bmatrix}$$

$$+ \begin{bmatrix}
1 &&&&&&&&&&&&& \\
0 & 1 &&&&&&&&&&&& \\
0 & 0 & 1 &&&&&&&&&&& \\
0 & 0 & 0 & 1 &&&&&&&&&& \\
0 & 0 & 0 & 0 & 1 &&&&&&&&& \\
0 & 0 & 0 & 0 & 0 & 1 &&&&&&&& \\
1 & 0 & 0 & 0 & 0 & 0 & 1 &&&&&&& \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 &&&&&& \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 &&&&& \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 &&&& \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 &&& \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 && \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix} e_0^{(1)} \\ e_1^{(1)} \\ e_2^{(1)} \\ e_3^{(1)} \\ e_4^{(1)} \\ e_5^{(1)} \\ e_6^{(1)} \\ e_7^{(1)} \\ e_8^{(1)} \\ e_9^{(1)} \\ e_{10}^{(1)} \\ e_{11}^{(1)} \\ e_{12}^{(1)} \\ e_{13}^{(1)} \end{bmatrix} + \begin{bmatrix} e_0^{(2)} \\ e_1^{(2)} \\ e_2^{(2)} \\ e_3^{(2)} \\ e_4^{(2)} \\ e_5^{(2)} \\ e_6^{(2)} \\ e_7^{(2)} \\ e_8^{(2)} \\ e_9^{(2)} \\ e_{10}^{(2)} \\ e_{11}^{(2)} \\ e_{12}^{(2)} \\ e_{13}^{(2)} \end{bmatrix}. \tag{13.60}$$

There are two parity triangles in this case, one corresponding to each generator polynomial. We can use these to form a set of $J = 4$ orthogonal check-sums on the information error bit $e_0^{(0)}$ as follows:

$$\begin{array}{*{29}{c}}
\rightarrow & 1 & & & & & & & & & & & & & & \boxed{1} \\
 & 0 & 1 & & & & & & & & & & & & & 0 & 1 \\
 & 0 & 0 & 1 & & & & & & & & & & & & 0 & 0 & 1 \\
 & 0 & 0 & 0 & 1 & & & & & & & & & & & 0 & 0 & 0 & 1 \\
 & 0 & 0 & 0 & 0 & 1 & & & & & & & & & & 0 & 0 & 0 & 0 & 1 \\
 & 0 & 0 & 0 & 0 & 0 & 1 & & & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & & & & & & & & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & & & & & & & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
\rightarrow & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \boxed{1} & & & & & & 0 & 0 & \boxed{1} & 0 & 0 & 0 & 0 & 0 & \boxed{1} \\
\rightarrow & 1 & \boxed{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \boxed{1} & & & & & 0 & 0 & 0 & \boxed{1} & 0 & 0 & 0 & 0 & 0 & \boxed{1} \\
 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & & & & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & & & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
\rightarrow & 1 & 0 & 0 & \boxed{1} & \boxed{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \boxed{1} & & 0 & \boxed{1} & 0 & 0 & 0 & 0 & \boxed{1} & 0 & 0 & 0 & 0 & 0 & \boxed{1} \\
 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 .
\end{array}$$

Similarly, we can form a set of $J = 4$ orthogonal check-sums on $e_0^{(1)}$ as follows:

$$\begin{array}{*{29}{c}}
\rightarrow & \boxed{1} & & & & & & & & & & & & & & 1 \\
 & 0 & 1 & & & & & & & & & & & & & 0 & 1 \\
 & 0 & 0 & 1 & & & & & & & & & & & & 0 & 0 & 1 \\
 & 0 & 0 & 0 & 1 & & & & & & & & & & & 0 & 0 & 0 & 1 \\
 & 0 & 0 & 0 & 0 & 1 & & & & & & & & & & 0 & 0 & 0 & 0 & 1 \\
 & 0 & 0 & 0 & 0 & 0 & 1 & & & & & & & & & 0 & 0 & 0 & 0 & 0 & 1 \\
\rightarrow & 0 & 0 & 0 & 0 & 0 & 0 & \boxed{1} & & & & & & & & 1 & 0 & 0 & 0 & 0 & 0 & \boxed{1} \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & & & & & & & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & & & & & & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & & & & & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & & & & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
\rightarrow & 0 & 0 & \boxed{1} & \boxed{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \boxed{1} & & & 1 & 0 & 0 & 0 & 0 & \boxed{1} & 0 & 0 & 0 & 0 & 0 & \boxed{1} \\
 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
\rightarrow & 0 & \boxed{1} & 0 & 0 & \boxed{1} & \boxed{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \boxed{1} & 1 & 0 & \boxed{1} & 0 & 0 & 0 & 0 & \boxed{1} & 0 & 0 & 0 & 0 & 0 & \boxed{1} .
\end{array}$$

Because all check-sums are formed from syndrome bits alone, and not sums of syndrome bits, this is a self-orthogonal code. A total of 31 different channel error bits (24 information error bits and 7 parity error bits) are checked by the two orthogonal sets, and the effective decoding length is $n_E = 31$. Hence, the majority-logic decoding rule will correctly estimate both $e_0^{(0)}$ and $e_0^{(1)}$ whenever $t_{ML} = 2$ or fewer of these $n_E = 31$ error bits are 1's. A block diagram of the decoder is shown in Figure 13.24.[6]

[6] In the remainder of this chapter the labeling of Figure 11.5 is used to describe rate $R = (n-1)/n$ systematic feedforward encoders.

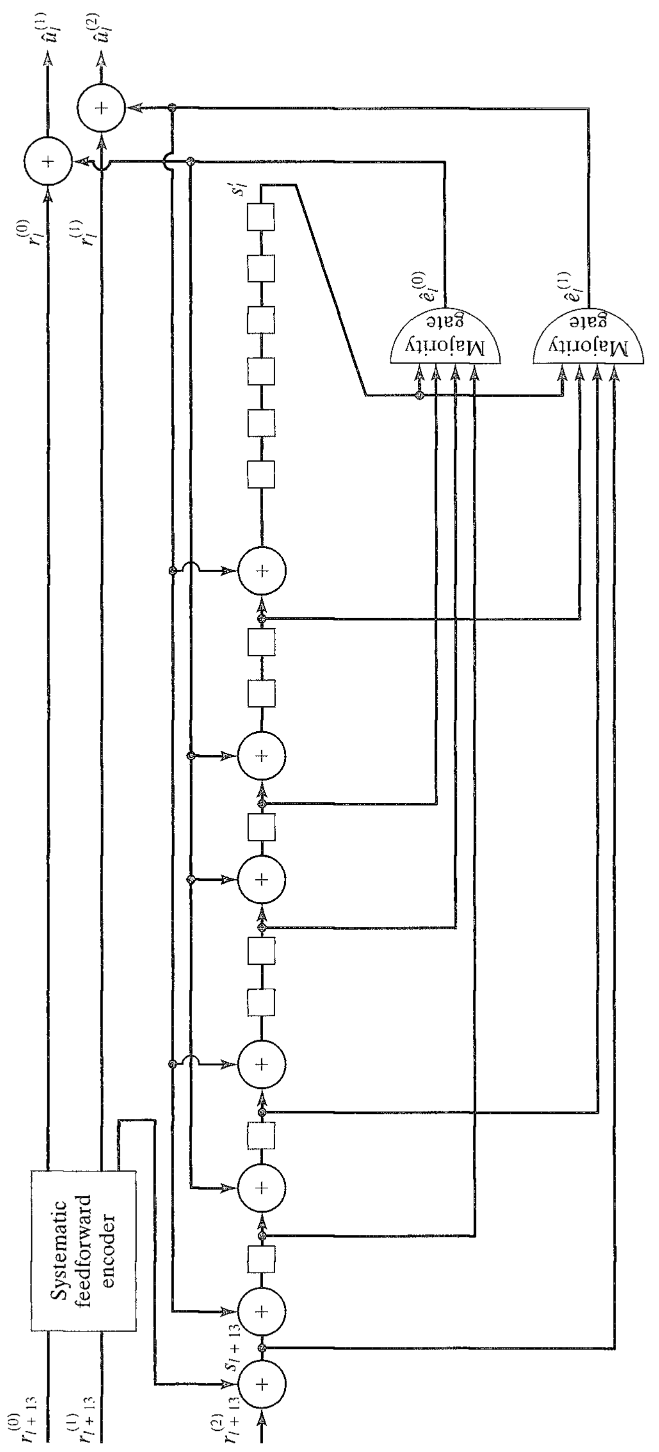


FIGURE 13.24: Complete majority-logic decoder for a (3, 2, 13) self-orthogonal code.

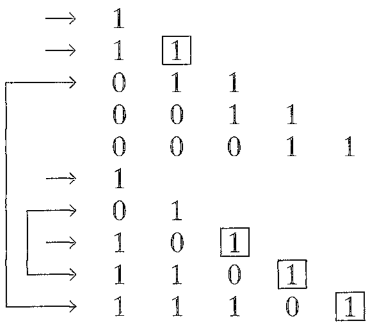
The total number of channel error bits that appear in the syndrome equations (13.59) is $n_A \triangleq n(m+1)$, called the *actual decoding length*. In Example 13.12, $n_A = 3(14) = 42$ channel error bits appear in the syndrome equations (13.60). Hence, 11 channel error bits have no effect on the estimates of $e_0^{(0)}$ and $e_0^{(1)}$. As decoding proceeds, however, these 11 error bits will affect the estimates of successive information error bits. Also note that there are many patterns of more than $t_{ML} = 2$ channel errors that are corrected by this code; however, there are some patterns of 3 channel errors that cannot be corrected.

EXAMPLE 13.13 A Rate $R = 1/3$ Orthogonalizable Code

Now, consider a (3, 1, 4) systematic code with $\mathbf{g}^{(1)}(D) = 1+D$ and $\mathbf{g}^{(2)}(D) = 1+D^2+D^3+D^4$. In this case, there are two syndrome sequences, $\mathbf{s}^{(1)} = (s_0^{(1)}, s_1^{(1)}, s_2^{(1)}, \cdots)$ and $\mathbf{s}^{(2)} = (s_0^{(2)}, s_1^{(2)}, s_2^{(2)}, \cdots)$, and we can write the first $(m+1) = 5$ blocks of syndrome bits as

$$[\mathbf{s}^T]_4 = \begin{bmatrix} s_0^{(1)} \\ s_1^{(1)} \\ s_2^{(1)} \\ s_3^{(1)} \\ s_4^{(1)} \\ s_0^{(2)} \\ s_1^{(2)} \\ s_2^{(2)} \\ s_3^{(2)} \\ s_4^{(2)} \end{bmatrix} = \begin{bmatrix} 1 & & & & \\ 1 & 1 & & & \\ 0 & 1 & 1 & & \\ 0 & 0 & 1 & 1 & \\ 0 & 0 & 0 & 1 & 1 \\ 1 & & & & \\ 0 & 1 & & & \\ 1 & 0 & 1 & & \\ 1 & 1 & 0 & 1 & \\ 1 & 1 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} e_0^{(0)} \\ e_1^{(0)} \\ e_2^{(0)} \\ e_3^{(0)} \\ e_4^{(0)} \end{bmatrix} + \begin{bmatrix} e_0^{(1)} \\ e_1^{(1)} \\ e_2^{(1)} \\ e_3^{(1)} \\ e_4^{(1)} \\ e_0^{(2)} \\ e_1^{(2)} \\ e_2^{(2)} \\ e_3^{(2)} \\ e_4^{(2)} \end{bmatrix}. \tag{13.61}$$

As in the case of the (3, 2, 13) code of Example 13.12, there are two parity triangles. In Example 13.12 we used the two parity triangles to form two separate sets of orthogonal check-sums on two different information error bits. In this example, there is only one information error bit per unit time, and we use the two parity triangles to form a single set of orthogonal check-sums on the information error bit $e_0^{(0)}$ as follows:



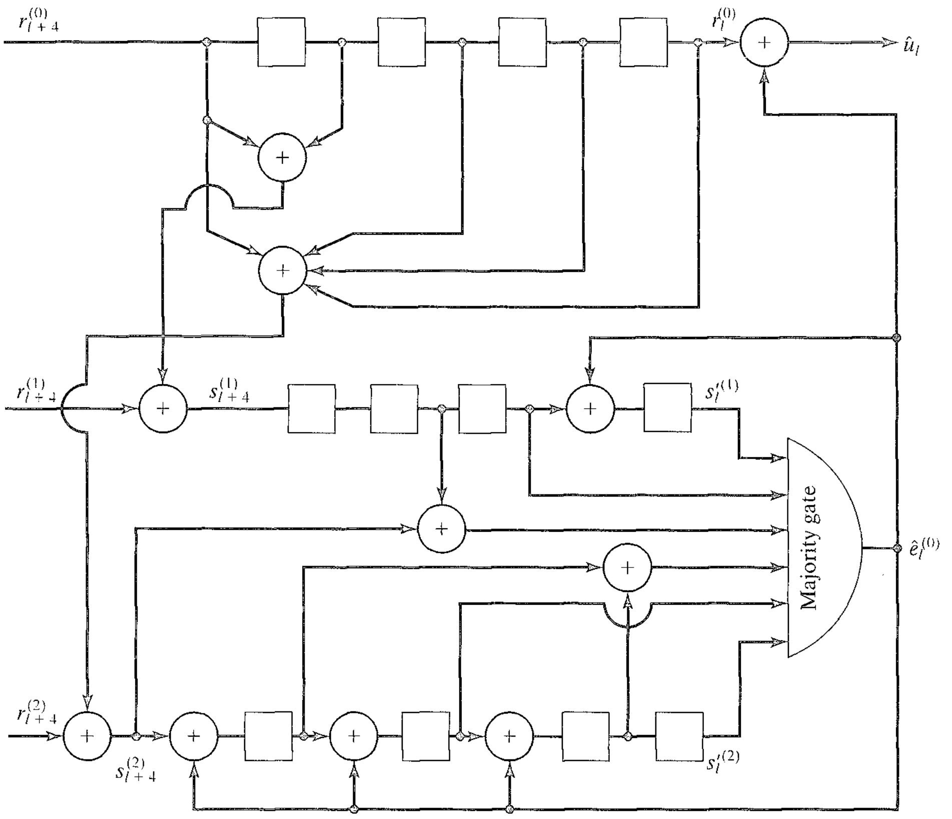


FIGURE 13.25: Complete majority-logic decoder for a (3, 1, 4) orthogonalizable code.

The check-sums $s_0^{(1)}, s_0^{(2)}, s_1^{(1)}, s_2^{(2)}, s_1^{(2)} + s_3^{(2)}$, and $s_2^{(1)} + s_4^{(2)}$ form a set of $J = 6$ orthogonal check-sums on $e_0^{(0)}$. The effective decoding length $n_E = 13$ for this orthogonalizable code, and the majority-logic decoding rule correctly estimates $e_0^{(0)}$ whenever $t_{ML} = 3$ or fewer of these 13 error bits are 1's. A block diagram of the decoder is shown in Figure 13.25.

In the foregoing discussion of feedback majority-logic decoding, it was assumed that the past estimates subtracted (added modulo-2) from the syndrome were all correct. This is of course not always true. When an incorrect estimate is fed back to the syndrome, it has the same effect as an additional transmission error and can cause further decoding errors that would not occur otherwise. This difficulty is called the *error propagation effect* of feedback decoders [33].

EXAMPLE 13.10 (Continued)

For the (2, 1, 6) self-orthogonal code of Example 13.10, let $\tilde{e}_l^{(0)} = e_l^{(0)} + \hat{e}_l^{(0)}$ be the result of adding the estimate $\hat{e}_l^{(0)}$ to a syndrome equation containing $e_l^{(0)}$. Because

$\tilde{e}_l^{(0)}$ is formed after $e_l^{(0)}$ has been decoded, it is called a *postdecoding error*. If all past estimates are assumed to be correct, the postdecoding errors are all equal to 0, and the modified syndrome equations are given by (13.55). If this is not the case, however, the unmodified syndrome equations are given by

$$\begin{aligned}
s'_l &= \tilde{e}_{l-6}^{(0)} + \tilde{e}_{l-4}^{(0)} + \tilde{e}_{l-1}^{(0)} + e_l^{(0)} + e_l^{(1)} \\
s'_{l+1} &= \tilde{e}_{l-5}^{(0)} + \tilde{e}_{l-3}^{(0)} + e_l^{(0)} + e_{l+1}^{(0)} + e_{l+1}^{(1)} \\
s'_{l+4} &= \tilde{e}_{l-2}^{(0)} + e_l^{(0)} + e_{l+3}^{(0)} + e_{l+4}^{(0)} + e_{l+4}^{(1)} \\
s_{l+6} &= e_l^{(0)} + e_{l+2}^{(0)} + e_{l+5}^{(0)} + e_{l+6}^{(0)} + e_{l+6}^{(1)}.
\end{aligned} \tag{13.62}$$

Clearly, (13.62) reduces to (13.55) if $\tilde{e}_\lambda^{(0)} = 0, \lambda = l - 6, \cdots, l - 1$. If any $\tilde{e}_\lambda^{(0)} = 1$, however, it has the same effect as a transmission error in the preceding equations and can cause error propagation.

Several approaches have been taken to reducing the effects of error propagation. One is to periodically resynchronize the decoder by inserting a string of km zeros into the information sequence after every kh information bits. When the resynchronization sequence is received, the decoder is instructed to decode a string of km consecutive zeros. During this m time unit span of correct decoding, all postdecoding errors must be zero, and the effect of the past is removed from the syndrome. This periodic resynchronization of the decoder limits error propagation to at most $h + m$ time units; however, for the fractional rate loss $m/(h + m)$ to be small, h must be much larger than m, and the effects of error propagation can still be quite long.

Another approach to limiting error propagation is to select codes with an automatic resynchronization property. Certain codes have the property that if the channel is error-free over a limited span of time units, the effect of past errors on the syndrome is automatically removed, thus halting error propagation. For example, it will be shown in Section 13.7 that self-orthogonal codes possess this property. Also, *uniform codes*, an orthogonalizable class of low-rate codes with large majority-logic error-correcting capability (similar to maximum-length block codes) introduced by Massey [34], have been shown by Sullivan [35] to possess an automatic resynchronization property.

Error propagation can be completely eliminated simply by not using feedback in the decoder; that is, past estimates are not fed back to modify the syndrome. This approach, first suggested by Robinson [36], is called *definite decoding*. Because the effects of previously estimated error bits are not removed from the syndrome in a definite decoder, however, these error bits can continue to influence future decoding estimates, thus possibly causing decoding errors that would not be made by a feedback decoder. On the other hand, error propagation due to erroneous decoding estimates is eliminated in definite decoding. An analysis by Morrissey [37] comparing the effect of error propagation with feedback to the reduced error-correcting capability without feedback concludes that feedback decoders outperform definite decoders unless the channel is very noisy.

Systematic feedforward encoders are preferred over nonsystematic feedforward encoders for majority-logic decoding applications because, with systematic encoders, orthogonal parity checks must be formed only on the information error bits. With nonsystematic encoders, orthogonal checks must be formed on all the error bits, and the estimated information sequence must be recovered from the estimated codeword using an encoder inverse. This recovery process can cause error amplification, as noted previously in Section 12.3. (Recall that for catastrophic encoders, the lack of a feedforward encoder inverse results in an infinite error amplification factor.) In addition, syndrome bits typically check more error bits for a nonsystematic encoder than for a systematic encoder with the same memory order, thus making it more difficult to find the same number of orthogonal parity checks. For example, we can view the construction of orthogonal parity checks for rate $R = 1/2$ nonsystematic feedforward encoders as being essentially equivalent to the construction of orthogonal parity checks for rate $R = 2/3$ systematic feedforward encoders.

As an illustration, consider the rate $R = 2/3$ systematic code of Example 13.12. If the two generator polynomials are used instead to construct a rate $R = 1/2$ nonsystematic code, exactly the same procedure can be used to form sets of $J = 4$ orthogonal parity checks on the error bits $e_0^{(0)}$ and $e_0^{(1)}$. (The effective decoding length of the rate $R = 1/2$ nonsystematic code is only $n_E = 24$ in this case, compared with $n_E = 31$ for the rate $R = 2/3$ systematic code.) It follows that for a given memory order m, we can form the same number of orthogonal parity checks on a rate $R = 2/3$ systematic code as on a rate $R = 1/2$ nonsystematic code, although the nonsystematic code will have a smaller effective decoding length. Also, this nonsystematic, rate $R = 1/2$, $J = 4$ self-orthogonal code has memory order $m = 13$ and effective decoding length $n_E = 24$, whereas the systematic, rate $R = 1/2$, $J = 4$ self-orthogonal code of Example 13.10 has $m = 6$ and $n_E = 11$. Thus, the systematic feedforward encoder results in a more efficient realization of a rate $R = 1/2$ code with majority-logic error-correcting capability $t_{ML} = 2$ than the nonsystematic feedforward encoder. In other words, although nonsystematic feedforward encoders have larger free distances than systematic feedforward encoders with the same memory order, this does not imply that they have larger majority-logic error-correcting capabilities (see Problem 13.23).

We now give an example showing that it is also possible to use systematic feedback encoders in majority-logic decoding applications.

EXAMPLE 13.14 A Rate $R = 1/2$ Self-Orthogonal Code (Feedback Encoder)

Consider the (2,1,4) systematic feedback encoder with generator matrix

$$G(D) = \begin{bmatrix} 1 & (1 + D^4)/(1 + D + D^2 + D^3 + D^4) \end{bmatrix}. \tag{13.63}$$

The parity generator sequence of this encoder is given by $\mathbf{g}^{(1)} = (1100101001\ 010010\cdots)$, where the 5-bit sequence (10010) repeats periodically after an initial 1. This semi-infinite parity generator sequence results in the following semi-infinite parity triangle:

$$\begin{array}{ccccccccccccccccc}
\rightarrow & 1 \\
\rightarrow & 1 & \boxed{1} \\
 & 0 & 1 & 1 \\
 & 0 & 0 & 1 & 1 \\
\rightarrow & 1 & 0 & 0 & \boxed{1} & \boxed{1} \\
 & 0 & 1 & 0 & 0 & 1 & 1 \\
\rightarrow & 1 & 0 & \boxed{1} & 0 & 0 & \boxed{1} & \boxed{1} \\
 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
 & \vdots & & & & & & & & & & & & & & & & \ddots
\end{array}$$

We see that it is possible to form $J = 4$ orthogonal parity checks on the information error bit $e_0^{(0)}$ and that the effective decoding length is $n_E = 11$. Because each orthogonal check-sum is a single syndrome bit, the code is self-orthogonal. This (2, 1, 4) self-orthogonal code with $J = 4$, $t_{ML} = 2$, and $n_E = 11$ is obtained with a memory order $m = 4$ systematic feedback encoder. In Example 13.10, the same parameters were obtained with an $m = 6$ systematic feedforward encoder. Thus, in this case, a feedback encoder gives a more efficient realization of a self-orthogonal code with majority-logic error-correcting capability $t_{ML} = 2$ than a feedforward encoder. It is interesting to note that if the preceding semi-infinite generator sequence $\mathbf{g}^{(1)} = (1100101001010010\cdots)$ is truncated after 7 bits, we obtain the same finite sequence $\mathbf{g}^{(1)} = (1100101)$ used to generate the $m = 6$ self-orthogonal code in Example 13.10.

For systematic codes with J orthogonal check-sums on each information error bit, a generalized form of the majority-logic decoding rule can be applied to DMCs and the unquantized AWGN channel. In these cases, the decoding rule takes the form of estimating an error bit $\hat{e}_l = 1$ if and only if a weighted sum of the orthogonal parity checks (formed, as on a BSC, from the hard-decision channel outputs) exceeds some (real-number) threshold. Thus, this generalized form of majority-logic decoding is referred to as *threshold decoding*.

For an unquantized AWGN channel, the (real-number) received sequence $\mathbf{r} = (r_0^{(0)} r_0^{(1)} \cdots r_0^{(n-1)}, r_1^{(0)} r_1^{(1)} \cdots r_1^{(n-1)}, r_2^{(0)} r_2^{(1)} \cdots r_2^{(n-1)}, \cdots)$ corresponding to the (normalized by $\sqrt{E_s}$) transmitter mapping $1 \rightarrow +1$ and $0 \rightarrow -1$ can be quantized into a hard-output sequence by using the mapping $r_l^{(j)} \rightarrow 1$ if $r_l^{(j)} > 0$; otherwise, $r_l^{(j)} \rightarrow 0$ (see Figure 13.26). Then, these hard-decision outputs can be used to form the (binary) syndrome sequence $\mathbf{s}$. Now, let $\{A_i\}$ represent a set of orthogonal check-sums (syndrome bits or sums of syndrome bits) on an error bit e_l, $i = 1, 2, \cdots, J$.

Threshold decoding is designed to produce an estimate $\hat{e}_l$ of the error bit e_l such that the a posteriori probability $P(e_l = \hat{e}_l|\{A_i\})$ is maximized. On a DMC, hard-decision outputs and orthogonal check-sums are formed in an analogous manner. For this reason, on DMCs or an unquantized AWGN channel, the term *a posteriori probability threshold decoding*, or *APP threshold decoding*, is used to refer to the generalized majority-logic decoding rule.

We now proceed to develop the APP threshold decoding rule for an unquantized AWGN channel. First, we let p_i be the probability that check-sum A_i contains an odd number of 1's, excluding error bit e_l , and we let $q_i = 1 - p_i$. (Clearly, q_i is the probability that check-sum A_i contains an even number of 1's, excluding error bit e_l.) Then, if n_i is the number of error bits included in check-sum A_i, excluding error bit e_l, we let e_{ij} be the jth error bit in check-sum A_i, r_{ij} be the corresponding soft received value, and γ_{ij} be the conditional probability that $e_{ij} = 1$ given r_{ij}, $j = 1, 2, \cdots, n_i$. In other words,

$$\gamma_{ij} = P(e_{ij} = 1|r_{ij}), \quad i = 1, 2, \cdots, J, \quad j = 1, 2, \cdots, n_i. \tag{13.64}$$

Assuming BPSK modulation, soft received values $\sqrt{E_s}r_{ij}$, $i = 1, 2, \cdots, J$, $j = 1, 2, \cdots, n_i$, and an unquantized AWGN channel with one-sided noise power spectral density N_0, we can show that (see Problem 13.24)

$$\gamma_{ij} = \frac{e^{-L_c|r_{ij}|}}{1 + e^{-L_c|r_{ij}|}}, \tag{13.65}$$

where $L_c = 4E_s/N_0$ is the channel reliability factor. For DMCs, the APP threshold decoding rule also quantizes the received sequence $\mathbf{r}$ into hard decisions and then uses the channel transition probabilities to calculate the conditional probabilities γ_{ij} in (13.64).

We now establish the relationship between p_i and the γ_{ij}'s by making use of a well-known result from discrete probability theory, expressed here as a lemma (see Problem 13.25).

Lemma 13.1 Let $\{e_j\}$ be a set of n independent binary random variables, let $\gamma_j = P(e_j = 1)$, $j = 1, 2, \cdots, n$, and let p be the probability that an odd number of the e_j's are 1's. Then,

$$p = \frac{1}{2}\left[1 - \prod_{j=1}^{n}(1 - 2\gamma_j)\right]. \tag{13.66}$$

Using (13.66) we can write

$$p_i = \frac{1}{2}\left[1 - \prod_{j=1}^{n_i}(1 - 2\gamma_{ij})\right], \quad i = 1, 2, \cdots, J. \tag{13.67}$$

Finally, we let $p_0 = P(e_l = 1|r_l)$ (computed using (13.65)) and $q_0 = 1 - p_0$ and define the *weighting factors* $w_i \equiv \log(q_i/p_i)$, $i = 0, 1, \cdots, J$. We can now state the APP threshold decoding rule for an unquantized AWGN channel.

APP Threshold Decoding Rule (AWGN Channel) We define the threshold $T \equiv \sum_{(0 \le i \le J)} w_i$, and we choose the estimate $\hat{e}_l = 1$ if and only if

$$\sum_{i=1}^{J} A_i w_i \geq T/2. \tag{13.68}$$

Note that if we choose all the weighting factors $w_i = 1, \ i = 0, 1, \cdots, J$, the APP threshold decoding rule becomes

$$\sum_{i=1}^{J} A_i \geq (J+1)/2, \tag{13.69}$$

which is equivalent to the majority-logic decoding rule for the BSC. It is interesting to note, however, that if we calculate the weighting factor using (13.64) and (13.67) for a BSC with crossover probability p, the APP threshold decoding rule is equivalent to the majority-logic decoding rule only if all J orthogonal check-sums include the same number of error bits, that is, only if $n_1 = n_2 = \cdots = n_J$ (see Problem 13.26).

THEOREM 13.2 The APP threshold decoding rule maximizes the APP value $P(e_l = \hat{e}_l | \{A_i\})$ of error bit e_l.

Proof. Using Bayes' rule, we can express the APP value $P(e_l = \hat{e}_l | \{A_i\})$ of error bit e_l as

$$P(e_l = \hat{e}_l | \{A_i\}) = P(\{A_i\} | e_l = \hat{e}_l) \frac{P(e_l = \hat{e}_l)}{P(\{A_i\})}, \tag{13.70}$$

where $P(\{A_i\} | e_l = \hat{e}_l)$ represents the joint probability of the J check-sum values, given that $e_l = \hat{e}_l$. A decoding rule that maximizes the APP value will therefore choose $\hat{e}_l = 1$ if and only if

$$P(\{A_i\} | e_l = 1) P(e_l = 1) \geq P(\{A_i\} | e_l = 0) P(e_l = 0). \tag{13.71}$$

Now, note that the check-sums $A_i, \ i = 1, 2, \cdots, J$, given a particular value of the error bit e_l, are independent random variables. This observation follows from the orthogonality property of the check-sums. Hence, we can write

$$P(\{A_i\} | e_l = \hat{e}_l) = \prod_{i=1}^{J} P(A_i | e_l = \hat{e}_l), \tag{13.72}$$

and a decoding rule that maximizes the APP value will choose $\hat{e}_l = 1$ if and only if

$$P(e_l = 1) \prod_{i=1}^{J} P(A_i | e_l = 1) \geq P(e_l = 0) \prod_{i=1}^{J} P(A_i | e_l = 0). \tag{13.73}$$

Taking logs of both sides and rearranging terms, we can rewrite (13.73) as

$$\sum_{i=1}^{J} \log \frac{P(A_i | e_l = 1)}{P(A_i | e_l = 0)} \geq \log \frac{P(e_l = 0)}{P(e_l = 1)}. \tag{13.74}$$

Next, we note that

$$P(A_i = 0 | e_l = 0) = P(A_i = 1 | e_l = 1) = q_i \tag{13.75a}$$

and

$$P(A_i = 1 | e_l = 0) = P(A_i = 0 | e_l = 1) = p_i. \tag{13.75b}$$

We can now write inequality (13.74) as

$$\sum_{i=1}^{J}(1-2A_i)\log\frac{p_i}{q_i} \geq \log\frac{q_0}{p_0}. \tag{13.76}$$

Hence, a decoding rule that maximizes the APP value will choose $\hat{e}_l = 1$ if and only if

$$-2\sum_{i=1}^{J} A_i \log\frac{p_i}{q_i} \geq -\sum_{i=0}^{J}\log\frac{p_i}{q_i}, \tag{13.77a}$$

$$\sum_{i=1}^{J} A_i \log\frac{q_i}{p_i} \geq \frac{1}{2}\sum_{i=0}^{J}\log\frac{q_i}{p_i}, \tag{13.77b}$$

or

$$\sum_{i=1}^{J} A_i w_i \geq T/2. \tag{13.77c}$$

Q.E.D.

It is interesting to compare the APP threshold decoding rule with the MAP decoding algorithm developed in Section 12.6. In APP threshold decoding, the a posteriori probability of an error bit e_l is maximized given the values of a set of J orthogonal check-sums on e_l, whereas in MAP decoding the a posteriori probability of an information bit u_l is maximized given the entire received sequence $\mathbf{r}$. Thus, each decision made by a MAP decoder is optimum, since it is based on the entire received sequence, whereas APP threshold decoding is suboptimum, because each decision is based on only a portion of the received sequence. Also, APP threshold decoding produces its estimates of the information bits indirectly by first estimating whether a hard-decision received bit was "correct" or "incorrect," whereas MAP decoding directly estimates the information bits based on their APP values. The APP threshold decoding rule is considerably simpler to implement than the MAP decoding algorithm, however.

To implement the APP threshold decoding rule for an unquantized AWGN channel, hard decisions must be made on the received symbols to compute the syndrome and the orthogonal check-sums as described earlier in this section. The soft received symbols must also be fed into a computation module, where, using (13.65) and (13.67), the weighting factors w_i, $i = 0, 1, \cdots, J$, and the threshold T are computed for each information error bit $e_l^{(j)}$ to be estimated, $l = 0, 1, 2, \cdots$, $j = 0, 1, \cdots, k-1$. (Note that using (13.65) requires that the receiver maintain an estimate of the channel SNR E_s/N_0, just as in MAP decoding.) Because the threshold T is, in general, different for each error bit to be estimated, decisions are made by comparing the (variable) weighted statistic $\sum_{(1\leq i\leq J)} A_i w_i - T/2$ with the fixed value 0; that is, we choose the estimate $e_l^{(j)} = 1$ if and only if

$$\sum_{i=1}^{J} A_i w_i - T/2 \geq 0\,. \tag{13.78}$$

A block diagram of an APP threshold decoder for the (2, 1, 6) self-orthogonal code of Example 13.10 and an unquantized AWGN channel is shown in Figure 13.26.

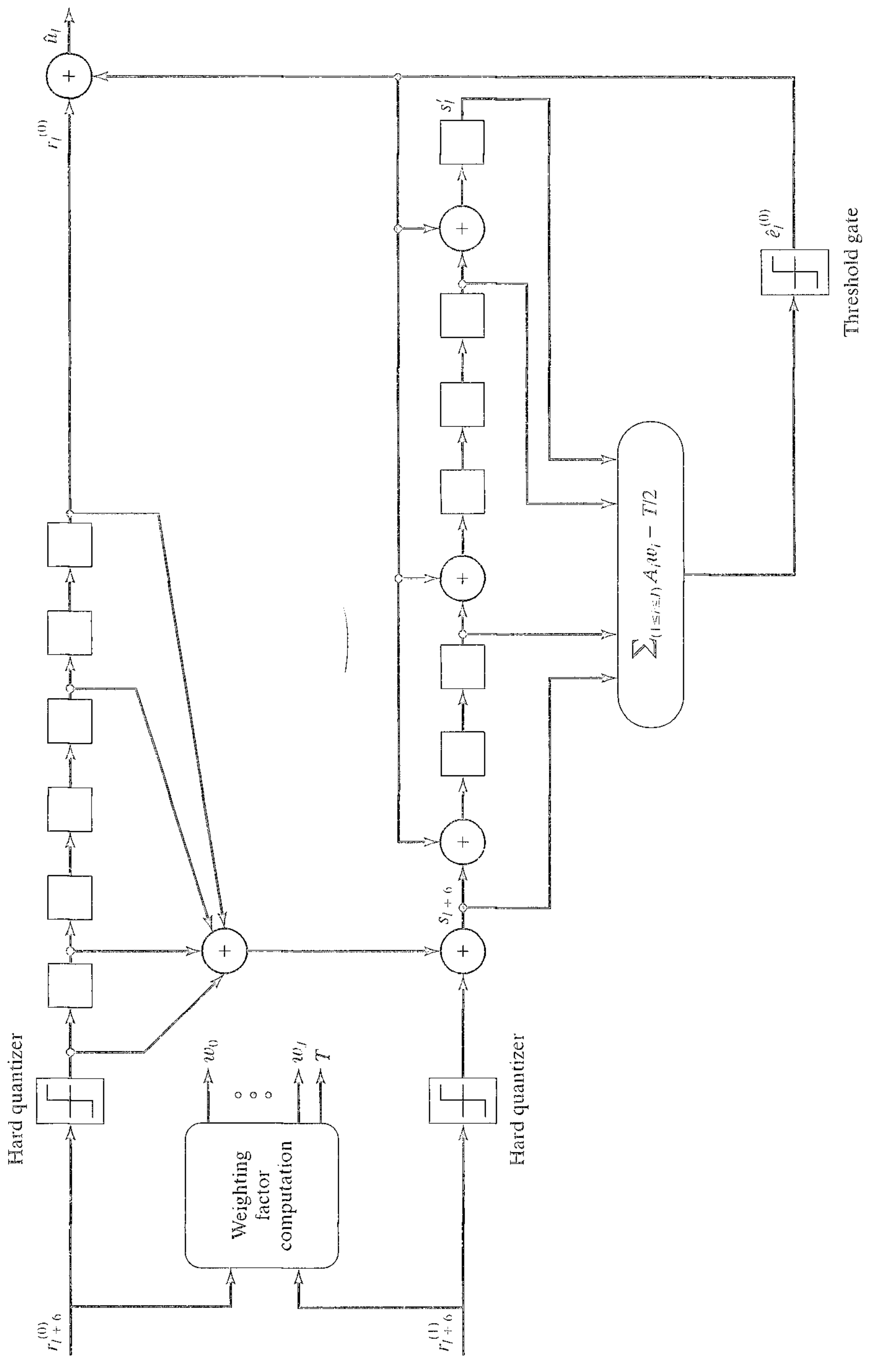


FIGURE 13.26: Complete APP threshold decoder for a (2, 1, 6) self-orthogonal code and an AWGN channel.

13.6 PERFORMANCE CHARACTERISTICS OF MAJORITY-LOGIC DECODING

In Section 11.3 the minimum distance of a convolutional code was defined as

$$\begin{aligned} d_{min} &= \min_{[\mathbf{u}']_m, [\mathbf{u}'']_m} \{d([\mathbf{v}']_m, [\mathbf{v}'']_m) : [\mathbf{u}']_0 \neq [\mathbf{u}'']_0\} \\ &= \min_{[\mathbf{u}]_m} \{w[\mathbf{v}]_m : \mathbf{u}_0 \neq \mathbf{0}\}, \end{aligned} \tag{13.79}$$

where $\mathbf{v}$, $\mathbf{v}'$, and $\mathbf{v}''$ are the codewords corresponding to the information sequences $\mathbf{u}$, $\mathbf{u}'$, and $\mathbf{u}''$, respectively. Note that in the computation of d_{min}, only the first $(m+1)$ time units are considered, and only codewords that differ in the first information block are compared. Now, consider a BSC with received sequence $\mathbf{r}$ and define a feedback decoder as one that produces an estimate $\hat{\mathbf{u}}_0 = \mathbf{u}_0$ of the first information block if and only if the codeword $\mathbf{v} = \mathbf{u}\mathbf{G}$ minimizes $d([\mathbf{r}]_m, [\mathbf{v}]_m)$, where $\mathbf{u}_0$ is the first block in the information sequence $\mathbf{u}$. The following theorem demonstrates that the minimum distance of a convolutional code guarantees a certain error-correcting capability for a feedback decoder.

THEOREM 13.3 For an (n, k, m) convolutional code with minimum distance d_{min}, $\mathbf{u}_0$ is correctly decoded by a feedback decoder if $\lfloor (d_{min} - 1)/2 \rfloor$ or fewer channel errors occur in the first $(m+1)$ blocks $[\mathbf{r}]_m$ of the received sequence.

Proof. Assume that the codeword $\mathbf{v}$ corresponding to the information sequence $\mathbf{u}$ is transmitted over a BSC and that $\mathbf{r} = \mathbf{v} + \mathbf{e}$ is received. Then,

$$d([\mathbf{r}]_m, [\mathbf{v}]_m) = w([\mathbf{r}]_m - [\mathbf{v}]_m) = w([\mathbf{e}]_m). \tag{13.80}$$

Now, let $\mathbf{v}'$ be a codeword corresponding to an information sequence $\mathbf{u}'$ with $\mathbf{u}'_0 \neq \mathbf{u}_0$. Then, the triangle inequality yields

$$\begin{aligned} d([\mathbf{r}]_m, [\mathbf{v}']_m) &\geq d([\mathbf{v}]_m, [\mathbf{v}']_m) - d([\mathbf{r}]_m, [\mathbf{v}]_m) \\ &= d([\mathbf{v}]_m, [\mathbf{v}']_m) - w([\mathbf{e}]_m) \\ &\geq d_{min} - w([\mathbf{e}]_m). \end{aligned} \tag{13.81}$$

By assumption, the number of channel errors $w([\mathbf{e}]_m) \leq (d_{min} - 1)/2$ and hence,

$$d([\mathbf{r}]_m, [\mathbf{v}']_m) \geq \frac{d_{min}+1}{2} > \frac{d_{min}-1}{2} \geq d([\mathbf{r}]_m, [\mathbf{v}]_m). \tag{13.82}$$

Therefore, no codeword $\mathbf{v}'$ with $\mathbf{u}'_0 \neq \mathbf{u}_0$ can be closer to $\mathbf{r}$ over the first $(m+1)$ time units than the transmitted codeword $\mathbf{v}$, and a feedback decoder will correctly decode the first information block $\mathbf{u}_0$. (Note that the codeword, say $\mathbf{v}'$, that minimizes $d([\mathbf{r}]_m, [\mathbf{v}']_m)$ may not equal $\mathbf{v}$, but it must agree with $\mathbf{v}$ in the first block.) Q.E.D.

In Theorem 13.3 we see that a feedback decoder that finds the codeword that is closest to the received sequence $\mathbf{r}$ over the first $(m+1)$ time units, and then chooses the first information block in this codeword as its estimate of $\mathbf{u}_0$, guarantees correct decoding of $\mathbf{u}_0$ if there are $\lfloor (d_{min} - 1)/2 \rfloor$ or fewer channel errors in the first $(m + 1)$ blocks of $\mathbf{r}$. Theorem 13.3 also has a converse; that is, a feedback decoder that correctly estimates $\mathbf{u}_0$ from $[\mathbf{r}]_m$ whenever $[\mathbf{r}]_m$ contains $\lfloor (d_{min} - 1)/2 \rfloor$ or fewer channel errors cannot correctly estimate $\mathbf{u}_0$ from $[\mathbf{r}]_m$ for all $[\mathbf{r}]_m$ containing $\lfloor (d_{min} - 1)/2 \rfloor + 1$ channel errors. In other words, there is at least one received sequence containing $\lfloor (d_{min} - 1)/2 \rfloor + 1$ errors in its first $(m+1)$ blocks that will result in incorrect decoding of $\mathbf{u}_0$ (see Problem 13.27). Hence, $t_{FB} \triangleq \lfloor (d_{min} - 1)/2 \rfloor$ is called the *feedback decoding error-correcting capability* of a code. If the decoding decisions that are fed back do not cause any postdecoding errors, the same error-correcting capability applies to the decoding of each successive information block $\mathbf{u}_l$ based on the $(m + 1)$ received blocks $(\mathbf{r}_l, \mathbf{r}_{l+1}, \cdots, \mathbf{r}_{l+m})$.

Any feedback decoder that bases its estimate of $\mathbf{u}_0$ on only the first $(m + 1)$ blocks of the received sequence thus has feedback decoding error-correcting capability $t_{FB} = \lfloor (d_{min} - 1)/2 \rfloor$. In the case of a feedback majority-logic decoder, since the J orthogonal parity checks guarantee correct decoding of $\mathbf{u}_0$ whenever $[\mathbf{r}]_m$ contains $t_{ML} = \lfloor J/2 \rfloor$ or fewer channel errors, it follows that

$$J \leq d_{min} - 1. \tag{13.83}$$

If $d_{min} - 1$ orthogonal parity checks can be formed on each information error bit, then the code is said to be *completely orthogonalizable*; that is, the feedback decoding error-correcting capability can be achieved with majority-logic decoding (see Example 13.15 and Problem 13.30). If the code is not completely orthogonalizable, however, majority-logic decoding cannot achieve the feedback decoding error-correcting capability (see Example 13.16 and Problem 13.31). Hence, it is desirable when using majority-logic decoding to select codes that are completely orthogonalizable. This restricts the choice of codes that can be used with majority-logic decoding, since most codes are not completely orthogonalizable; however, as will be seen in Section 13.7, several classes of completely orthogonalizable convolutional codes have been found.

Note that the result of Theorem 13.3 does not apply to Viterbi, BCJR, or sequential decoding, since these decoding methods process the entire received sequence $\mathbf{r}$ before making a final decision on $\mathbf{u}_0$.[7] The longer decoding delay of these decoding methods is one reason for their superior performance when compared with majority-logic decoding; however, majority-logic decoders are much simpler to implement, since they must store only $(m + 1)$ blocks of the received sequence at any time.

The minimum distance of a convolutional code can be found by computing the CDF d_i and then letting $i = m$ (since $d_{min} = d_m$). A more direct way of finding d_{min} makes use of the parity-check matrix $\mathbb{H}$. Because $\mathbf{v}$ is a codeword if and only if $\mathbf{v}\mathbb{H}^T = \mathbf{0}$, the minimum number of rows of $\mathbb{H}^T$, or columns of $\mathbb{H}$, that add to $\mathbf{0}$ corresponds to the minimum-weight nonzero codeword. Because d_{min} is the weight of the codeword with $\mathbf{u}_0 \neq 0$ that has minimum weight over its first $(m + 1)$ blocks, it can be computed by finding the minimum number of rows of $\mathbb{H}^T$, or columns of

[7]Even a truncated Viterbi decoder, or a backsearch-limited sequential decoder, must process at least $4m$ or $5m$ blocks of the received sequence before making any final decisions.

$\mathbb{H}$, including at least one of the first k, that add to $\mathbb{0}$ over the first $(m+1)$ time units. In other words, we must find the minimum-weight codeword $[\mathbb{v}]_m$ with $\mathbb{u}_0 \neq \mathbb{0}$ for which $[\mathbb{v}]_m[\mathbb{H}^T]_m = \mathbb{0}$, where $[\mathbb{H}^T]_m$ includes only the first $(m+1)(n-k)$ columns of $\mathbb{H}^T$. This is equivalent to forming the first $(m+1)(n-k)$ rows of $\mathbb{H}$ and then finding the minimum number of columns of this matrix, including at least one of the first k columns, that add to $\mathbb{0}$.

EXAMPLE 13.15 Finding d_{min} Using the $\mathbb{H}$ Matrix

Consider the (2, 1, 6) systematic code of Example 13.10. The first $(m+1)(n-k) = (7)(1) = 7$ rows of $\mathbb{H}$ are

$$[\mathbb{H}]_m = [\mathbb{H}]_6 = \begin{bmatrix} 1 & 1 & & & & & & & & & & & & \\ 1 & 0 & 1 & 1 & & & & & & & & & & \\ 0 & 0 & 1 & 0 & 1 & 1 & & & & & & & & \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & & & & & & \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & & & & \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & & \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}. \tag{13.84}$$

The minimum distance d_{min} is the minimum number of columns of this matrix, including the first column, that add to zero. Because $J = 4$ orthogonal check-sums can be found for this code, we know that $d_{min} \geq J + 1 = 5$. But columns 1, 2, 4, 10, and 14 add to zero, implying that $d_{min} \leq 5$. Hence, d_{min} must equal 5 for this code, and the code is completely orthogonalizable. The minimum-weight codeword with $\mathbb{u}_0 \neq \mathbb{0}$ in this case is given by

$$[\mathbb{v}]_m = (1\,1, 0\,1, 0\,0, 0\,0, 0\,1, 0\,0, 0\,1).$$

We can also obtain d_{min} by finding the minimum-weight linear combination of rows of the generator matrix $[\mathbb{G}]_m$ that includes at least one of the first k rows, that is, the codeword with $\mathbb{u}_0 \neq \mathbb{0}$ that has minimum weight over the first $(m+1)$ time units.

EXAMPLE 13.16 Finding d_{min} Using the $\mathbb{G}$ Matrix

Consider the (2, 1, 5) systematic code with $\mathrm{g}^{(1)}(D) = 1 + D + D^3 + D^5$. Then,

$$[\mathbb{G}]_m = [\mathbb{G}]_5 = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ & & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ & & & & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ & & & & & & 1 & 1 & 0 & 1 & 0 & 0 \\ & & & & & & & & 1 & 1 & 0 & 1 \\ & & & & & & & & & & 1 & 1 \end{bmatrix}. \tag{13.85}$$

There are several information sequences with $u_0 = 1$ that produce codewords of weight 5. For example, the information sequence $[\mathbb{u}]_m = (1\,1\,1\,0\,0\,0)$ produces the codeword $[\mathbb{v}]_m = (11, 10, 10, 00, 01, 00)$; however, there are no linear combinations of rows of $[\mathbb{G}]_m$ including the first row that have weight 4. Hence, $d_{min} = 5$ for this code. The parity triangle is given by

$$\begin{array}{llllll}
1 \\
1 & 1 \\
0 & 1 & 1 \\
1 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 1 & 1.
\end{array}$$

The maximum number of orthogonal check-sums that can be formed in this case is $J = 3$. For example, $\{s_0, s_1, s_3\}$ and $\{s_0, s_2 + s_3, s_5\}$ are both sets of $J = 3$ orthogonal check-sums on $e_0^{(0)}$. Thus, since $J = 3 < d_{min} - 1 = 4$, this code is not completely orthogonalizable; that is, its feedback decoding error-correcting capability is 2, but its majority-logic error-correcting capability is only 1.

Examples 13.14 and 13.15 both consider systematic feedforward encoders. It was shown in Section 11.1 that any (n, k, m) nonsystematic feedforward encoder, by means of a linear transformation on the rows of its generator matrix, can be converted to an equivalent (n, k, m') systematic feedback encoder. (Note that for $k > 1$, m' may be different from m.) Truncating the (in general, semi-infinite) generator sequences of the systematic feedback encoder to length $m + 1$ then results in an (n, k, m) systematic feedforward encoder with the same value of d_{min} as the original nonsystematic feedforward encoder. Hence, the feedback decoding error-correcting capability cannot be improved by considering nonsystematic feedforward encoders. This result implies that in the selection of encoders with memory order m and large d_{min} for possible use with majority-logic decoding, it suffices to consider only systematic feedforward encoders. The situation is markedly different for d_{free}, where nonsystematic feedforward encoders offer substantially larger values of d_{free} than systematic feedforward encoders with the same constraint length ν, as was noted previously in Sections 11.3 and 12.3. This advantage in d_{free} of nonsystematic feedforward encoders (or equivalent systematic feedback encoders) compared with systematic feedforward encoders with the same value of ν accounts for their almost exclusive use in applications involving Viterbi or BCJR decoding, where decoding complexity increases exponentially with ν.

EXAMPLE 13.14 (Continued)

Consider the (2, 1, 4) systematic feedback encoder of Example 13.14 and its equivalent (2, 1, 4) nonsystematic feedforward encoder with $\mathbb{G}(D) = [1 + D + D^2 + D^3 + D^4 \quad 1 + D^4]$. Then, for the nonsystematic encoder,

$$[\mathbb{G}]_4 = \begin{bmatrix} 11 & 10 & 10 & 10 & 11 \\ & 11 & 10 & 10 & 10 \\ & & 11 & 10 & 10 \\ & & & 11 & 10 \\ & & & & 11 \end{bmatrix}, \qquad (13.86)$$

and d_{min} is the minimum-weight linear combination of rows of $[\mathbb{G}]_4$ that includes the first row. Clearly, the sum of rows 1 and 2 is a weight-4 codeword, and $d_{min} = 4$ for this code. Now, consider the semi-infinite generator matrix

$$\mathbb{G} = \left[\begin{array}{cccccccccc} 11 & 01 & 00 & 00 & 01 & 00 & 01 & \cdots & & \\ & 11 & 01 & 00 & 00 & 01 & 00 & 01 & \cdots & \\ & & 11 & 01 & 00 & 00 & 01 & 00 & 01 & \cdots \\ & & & \ddots & & & & & & \ddots \end{array}\right] \tag{13.87}$$

of the systematic feedback encoder. Truncating the semi-infinite generator sequence $\mathbf{g}^{(1)} = (11001010010\cdots)$ to length $m + 1 = 5$, we obtain a systematic feedforward encoder with generator matrix $\mathbb{G}'(D) = [1 \quad 1 + D + D^4]$, and

$$[\mathbb{G}']_4 = \left[\begin{array}{ccccc} 11 & 01 & 00 & 00 & 01 \\ & 11 & 01 & 00 & 00 \\ & & 11 & 01 & 00 \\ & & & 11 & 01 \\ & & & & 11 \end{array}\right]. \tag{13.88}$$

The minimum distance d'_{min} of this code is the minimum-weight linear combination of rows of $[\mathbb{G}']_4$ that includes the first row. In this case, row 1 by itself is a weight-4 codeword, and $d'_{min} = 4 = d_{min}$. Hence, the systematic feedforward encoder with generator matrix $\mathbb{G}'(D)$ has the same minimum distance as the original nonsystematic feedforward encoder with generator matrix $\mathbb{G}(D)$. The free distance of the nonsystematic encoder is 6, however, whereas the free distance of the systematic encoder is only 4.

The performance of an (n, k, m) convolutional code with majority-logic decoding on a BSC can be estimated from the distance properties of the code. First, for a feedback decoder with error-correcting capability $t_{FB} = \lfloor (d_{min} - 1)/2 \rfloor$, it follows from Theorem 13.3 that a decoding error can occur in estimating $\mathbf{u}_0$ only if more than t_{FB} channel errors occur in the first $(m + 1)$ blocks of the received sequence. Hence, recalling that the total number of channel error bits that appear in the syndrome equations (13.59) is the actual decoding length $n_A = n(m + 1)$, we see that the bit-error probability in decoding the first information block, $P_{b1}(E)$, can be upper bounded by

$$P_{b1}(E) \leq \frac{1}{k} \sum_{i=t_{FB}+1}^{n_A} \binom{n_A}{i} p^i (1 - p)^{n_A - i}, \tag{13.89}$$

where p is the channel transition probability. For small p, this bound is dominated by its first term, so that

$$P_{b1}(E) \approx \frac{1}{k} \binom{n_A}{t_{FB} + 1} p^{t_{FB}+1} (1 - p)^{n_A - t_{FB} - 1} \approx \frac{1}{k} \binom{n_A}{t_{FB} + 1} p^{t_{FB}+1}. \tag{13.90}$$

For a majority-logic decoder with error-correcting capability $t_{ML} = \lfloor J/2 \rfloor$ and effective decoding length n_E, an analogous argument yields

$$P_{b1}(E) \triangleq \frac{1}{k} \sum_{i=t_{ML}+1}^{n_E} \binom{n_E}{i} p^i (1-p)^{n_E - i} \tag{13.91}$$

and

$$P_{b1}(E) \approx \frac{1}{k} \binom{n_E}{t_{ML}+1} p^{t_{ML}+1}. \tag{13.92}$$

These results strictly apply only to decoding the first information block; however, if each estimated information error block $\hat{e}_l$ is subtracted from the syndrome equations it affects, and if these estimates are correct, the modified syndrome equations used to estimate $\mathbf{e}_{l+1}$ are identical to those used to estimate $\mathbf{e}_l$, except that different error bits are checked. This point was illustrated in Section 13.5 in connection with Example 13.10. Hence, for a feedback decoder, the bit-error probability in decoding any information block, $P_b(E)$, equals $P_{b1}(E)$, assuming that previous estimates have been correct. Under this assumption, a feedback decoder has bit-error probability upper bounded by

$$P_b(E) \leq \frac{1}{k} \sum_{i=t_{FB}+1}^{n_A} \binom{n_A}{i} p^i (1-p)^{n_A - i}, \tag{13.93}$$

and, for small p, approximated by

$$P_b(E) \approx \frac{1}{k} \binom{n_A}{t_{FB}+1} p^{t_{FB}+1}. \tag{13.94}$$

Similarly, for a majority-logic decoder that uses feedback to modify the syndrome equations,

$$P_b(E) \leq \frac{1}{k} \sum_{i=t_{ML}+1}^{n_E} \binom{n_E}{i} p^i (1-p)^{n_E - i} \tag{13.95}$$

and

$$P_b(E) \approx \frac{1}{k} \binom{n_E}{t_{ML}+1} p^{t_{ML}+1}. \tag{13.96}$$

As noted in Section 13.5, if previous decoding estimates are not all correct, postdecoding errors can appear in the modified syndrome equations, thereby causing error propagation and degrading the performance of the decoder. If the channel transition probability p is not too large, however, and if the code contains good resynchronization properties, such as the self-orthogonal codes to be discussed in the next section, the effect on bit-error probability is small, and (13.93) through (13.96) remain valid [37]. Finally, for the same code, soft-decision APP threshold decoding will generally perform about 2 dB better than hard-decision majority-logic decoding.

We now do a rudimentary comparison of majority-logic decoding and Viterbi decoding on the basis of their performance, decoding speed, decoding delay, and implementation complexity.

Performance. Viterbi decoding is an optimum decoding procedure for convolutional codes, and its bit-error probability on the BSC was shown in (12.36) to be approximated by

$$P_b(E) \approx C_V e^{-(Rd_{free}/2)(E_b/N_0)} \tag{13.97}$$

when E_b/N_0 is large, where C_V is a constant associated with the code structure. For a (suboptimum) feedback decoder, it was shown previously that for a BSC,

$$P_b(E) \approx \frac{1}{k}\binom{n_A}{t_{FB}+1} p^{t_{FB}+1} \tag{13.98}$$

for small p. Using the approximation (see (12.33) and (12.35))

$$p \approx \tfrac{1}{2} e^{-E_b R/N_0} \tag{13.99}$$

we obtain

$$\begin{aligned} P_b(E) &\approx \frac{1}{k}\binom{n_A}{t_{FB}+1}\left(\frac{1}{2}\right) e^{-E_b R(t_{FB}+1)/N_0} \\ &\approx C_{FB} e^{-R(d_{min}/2)(E_b/N_0)} \end{aligned} \tag{13.100}$$

when E_b/N_0 is large, where C_{FB} is a constant associated with the code structure. If the code is completely orthogonalizable, (13.100) also represents the performance of majority-logic decoding. The approximations of (13.97) and (13.100) differ in that $P_b(E)$ decreases exponentially with d_{free} for Viterbi decoding, whereas $P_b(E)$ decreases exponentially with d_{min} for feedback (majority-logic) decoding. Because for most codes

$$d_{min} = d_m < \lim_{l\to\infty} d_l = d_{free}, \tag{13.101}$$

the asymptotic performance of feedback (majority-logic) decoding is inferior to that of Viterbi decoding, since a Viterbi decoder delays making decisions until the entire received sequence is processed or, in the case of a truncated decoder, until at least $4m$ or $5m$ blocks are received. It thus bases its decisions on the total (or free) distance between sequences. A feedback (majority-logic) decoder, on the other hand, bases its decisions on the minimum distance over only $(m + 1)$ time units.

Random coding bounds indicate that for a given rate and memory order, the free distance of the best nonsystematic feedforward encoders (or systematic feedback encoders) is about twice the minimum distance of the best systematic feedforward encoders [38]. (Recall that nonsystematic encoders cannot achieve larger values of d_{min} than systematic encoders, however.) This gives Viterbi decoding a roughly 3-dB advantage in performance over feedback decoding for the same rate and memory order. In other words, the memory order for feedback decoding must be about twice that for Viterbi decoding to achieve comparable performance. If majority-logic decoding is being used, an even longer memory order must be employed. This is because optimum minimum distance codes are usually not completely orthogonalizable; that is, to achieve a given d_{min} for a completely orthogonalizable code requires a longer memory order than would otherwise be necessary. If the code must be self-orthogonal (say, to protect against error propagation), the memory order must be longer yet. For example, the best rate

$R = 1/2$ self-orthogonal code with $d_{min} = 7$ has memory order $m = 17$, the best $R = 1/2$ orthogonalizable code with $d_{min} = 7$ has memory order $m = 11$, the best $R = 1/2$ systematic code with $d_{min} = 7$ has memory order $m = 10$, and the best $R = 1/2$ nonsystematic code with $d_{free} = 7$ has memory order $m = 4$.

Decoding Speed. A Viterbi decoder requires 2^{ν} computations (ACS operations) per decoded information bit, whereas a majority-logic decoder requires only one computation (majority gate decision) per decoded bit. Although the time required to perform a computation is somewhat different in each case, this comparison indicates that majority-logic decoders are capable of much higher speed operation than Viterbi decoders. (As noted in Chapter 12, the speed of Viterbi decoding can be increased by a factor of 2^{ν}, making it attractive for high-speed applications, by using a parallel implementation. This greatly increases the implementation complexity of the decoder, however.)

Decoding Delay. A majority-logic decoder has a decoding delay of m time units; that is, bits received at time unit l are decoded at time unit $l + m$. Viterbi decoding, on the other hand, has a decoding delay equal to the entire frame length $h + m$; that is, no decoding decisions are made until all $h + m$ encoded blocks have been received. Because typically $h >> m$, the decoding delay is substantial in these cases.[8] (If truncated Viterbi decoding is used, the decoding delay is reduced to $4m$ or $5m$ time units, with a minor penalty in performance.)

Implementation Complexity. Majority-logic decoders are simpler to implement than Viterbi decoders. Besides a replica of the encoder and a buffer register for storing the received information bits, all that is needed is a syndrome register, some modulo-2 adders (EXCLUSIVE-OR gates), and k majority gates. These relatively modest implementation requirements make majority-logic decoding particularly attractive in low-cost applications; however, if large minimum distances are needed to achieve high reliabilities at low SNRs, very large memory orders are required, and the implementation complexity and decoding delay increases. In these cases, Viterbi decoding provides a more efficient trade-off between performance and complexity.

Finally, we saw in Chapter 12 that Viterbi decoders can easily be adapted to take advantage of soft demodulator decisions. Soft-decision majority-logic decoders have also been developed, but at a considerable increase in implementation complexity. Massey's [6] APP decoding, discussed in the previous section, and Rudolph's [39] generalized majority-logic decoding are examples of such schemes.

13.7 CODE CONSTRUCTION FOR MAJORITY-LOGIC DECODING

In this section we discuss two classes of completely orthogonalizable codes for use with majority-logic decoding: self-orthogonal codes and orthogonalizable codes.

[8]Recall that for any convolutional code, when data is sent in frames, an additional m time units of "known" input bits must be fed into the encoder and the resulting encoder outputs transmitted over the channel to allow proper decoding of the last m blocks of information bits. Thus, when m is large, short data frames result in considerable rate loss. In the case of Viterbi decoding, the "known" bits are used to drive the encoder back to the all-zero state. In majority-logic decoding, the "known" bits are arbitrary, and knowledge of these bits can be used by the decoder to improve the reliability of the last m blocks of information bits.

The resynchronization properties of each of these classes when used with feedback decoding is also discussed.

13.7.1 Self-Orthogonal Codes

An (n, k, m) code is said to be *self-orthogonal* if, for each information error bit in block $\mathbf{u}_0$, the set of all syndrome bits that check it forms an orthogonal check-set on that bit. In other words, sums of syndrome are not used to form orthogonal check-sets in a self-orthogonal code.

Self-orthogonal codes were first constructed by Massey [6]. A more efficient construction, based on the notion of difference sets, was introduced by Robinson and Bernstein [40]. The *positive difference set* Δ associated with a set of nonnegative integers $\{l_1, l_2, \cdots, l_J\}$, where $l_1 < l_2 < \cdots < l_J$, is defined as the set of $J(J-1)/2$ positive differences $l_b - l_a$, where $l_b > l_a$. A positive difference set Δ is said to be *full* if all the differences in Δ are distinct, and two positive difference sets Δ_i and Δ_j are said to be *disjoint* if they do not contain any differences in common. Now, consider an $(n, n-1, m)$ systematic convolutional code with generator polynomials

$$\mathbf{g}_{j+1}^{(n-1)}(D) = g_{j+1,0}^{(n-1)} + g_{j+1,1}^{(n-1)}D + \cdots + g_{j+1,\nu_{j+1}}^{(n-1)}D^{\nu_{j+1}}, \quad j = 0, 1, \cdots, n-2, \tag{13.102}$$

where the memory order $m = \max_{(0 \le j \le n-2)} \nu_{j+1}$. Let $g_{j+1,l_1}^{(n-1)}, g_{j+1,l_2}^{(n-1)}, \cdots, g_{j+1,l_{J_j}}^{(n-1)}$ be the nonzero components of $\mathbf{g}_{j+1}^{(n-1)}(D)$, where $l_1 < l_2 < \cdots < l_{J_j}$, $j = 0, 1, \cdots, n-2$, and let Δ_j be the positive difference set associated with the set of integers of $\{l_1, l_2, \cdots, l_{J_j}\}$. The following theorem forms the basis for the construction of self-orthogonal convolutional codes.

Theorem 13.4 An $(n, n-1, m)$ systematic code is self-orthogonal if and only if the positive difference sets $\Delta_0, \Delta_1, \cdots, \Delta_{n-2}$ associated with the code generator polynomials are full and mutually disjoint.

Proof. The proof of this theorem consists of two parts.

First, assume the code is self-orthogonal, and suppose that a positive difference set Δ_j exists that is not full. Then, at least two differences in Δ_j are equal, say $l_b - l_a = l_d - l_c$, where $l_b > l_a$ and $l_d > l_c$. Using (13.57), we can write the syndrome sequence as

$$\mathbf{s}^{(n-1)}(D) = \sum_{k=0}^{n-2} \mathbf{e}^{(k)}(D)\mathbf{g}_{k+1}^{(n-1)}(D) + \mathbf{e}^{(n-1)}(D), \tag{13.103}$$

which we also can express as

$$\begin{aligned}\mathbf{s}^{(n-1)}(D) &= \mathbf{e}^{(j)}(D)\mathbf{g}_{j+1}^{(n-1)}(D) + \sum_{\substack{k=0\\k\neq j}}^{n-2} \mathbf{e}^{(k)}(D)\mathbf{g}_{k+1}^{(n-1)}(D) + \mathbf{e}^{(n-1)}(D)\\ &= (D^{l_1} + D^{l_2} + \cdots + D^{l_{J_j}})\mathbf{e}^{(j)}(D) + \sum_{\substack{k=0\\k\neq j}}^{n-2} \mathbf{e}^{(k)}(D)\mathbf{g}_{k+1}^{(n-1)}(D) + \mathbf{e}^{(n-1)}(D).\end{aligned} \tag{13.104}$$

Because l_a, l_b, l_c, and l_d all belong to the set $\{l_1, l_2, \cdots, l_{J_j}\}$, the syndrome bits $s_{l_a}^{(n-1)}, s_{l_b}^{(n-1)}, s_{l_c}^{(n-1)}$, and $s_{l_d}^{(n-1)}$ all check the information error bit $e_0^{(j)}$. In particular,

$$s_{l_b}^{(n-1)} = e_0^{(j)} + e_{l_b-l_a}^{(j)} + \text{ other terms} \tag{13.105a}$$

and

$$s_{l_d}^{(n-1)} = e_0^{(j)} + e_{l_d-l_c}^{(j)} + \text{ other terms.} \tag{13.105b}$$

Because $l_b - l_a = l_d - l_c$, $e_{l_b-l_a}^{(j)} = e_{l_d-l_c}^{(j)}$, and the set of syndrome bits that check the information error bit $e_0^{(j)}$ is not orthogonal, thus contradicting the assumption that the code is self-orthogonal.

Now, suppose that two difference sets Δ_i and Δ_j exist that are not disjoint. Then, they must have at least one difference in common. Let $l_b - l_a$ and $f_d - f_c$ be the common difference in Δ_i and Δ_j, respectively. Then, $l_b - l_a = f_d - f_c$, and we can write the syndrome sequence as

$$\begin{aligned}\mathbb{s}^{(n-1)}(D) &= \mathbb{e}^{(i)}(D)\mathbb{g}_{i+1}^{(n-1)}(D) + \mathbb{e}^{(j)}(D)\mathbb{g}_{j+1}^{(n-1)}(D)\\ &\quad + \sum_{\substack{k=0\\k\neq i,j}}^{n-2} \mathbb{e}^{(k)}(D)\mathbb{g}_{k+1}^{(n-1)}(D) + \mathbb{e}^{(n-1)}(D)\\ &= (D^{l_1}+D^{l_2}+\cdots+D^{l_{J_i}})\mathbb{e}^{(i)}(D) + (D^{f_1}+D^{f_2}+\cdots+D^{f_{J_j}})\mathbb{e}^{(j)}(D)\\ &\quad + \sum_{\substack{k=0\\k\neq i,j}}^{n-2} \mathbb{e}^{(k)}(D)\mathbb{g}_{k+1}^{(n-1)}(D) + \mathbb{e}^{(n-1)}(D).\end{aligned} \tag{13.106}$$

Assume, without loss of generality, that $l_b \geq f_d$. Then, $l_a \geq f_c$, and since l_a and l_b belong to set $l_1, l_2, \cdots, l_{J_i}$, the syndrome bits $s_{l_a}^{(n-1)}$ and $s_{l_b}^{(n-1)}$ both check the information error bit $e_0^{(i)}$; that is,

$$s_{l_a}^{(n-1)} = e_0^{(i)} + e_{l_a-f_c}^{(j)} + \text{ other terms} \tag{13.107a}$$

and

$$s_{l_b}^{(n-1)} = e_0^{(i)} + e_{l_b-f_d}^{(j)} + \text{ other terms.} \tag{13.107b}$$

But $l_b - l_a = f_d - f_c$ implies that $l_b - f_d = l_a - f_c$, and hence $e_{l_a-f_c}^{(j)} = e_{l_b-f_d}^{(j)}$. Therefore, the set of syndrome bits that check the information error bit $e_0^{(i)}$ is not orthogonal, which again contradicts the assumption that the code is self-orthogonal. We conclude that all $n-1$ positive difference sets must be full and disjoint.

Second, assume that the positive difference sets $\Delta_0, \Delta_1, \cdots, \Delta_{n-2}$ are full and disjoint, and suppose that the code is not self-orthogonal. Then, there must exist at least one pair of syndrome bits for which

$$s_{l_a}^{(n-1)} = e_0^{(i)} + e_{l_a-f_c}^{(j)} + \text{ other terms} \tag{13.108a}$$

and

$$s_{l_b}^{(n-1)} = e_0^{(i)} + e_{l_b - f_d}^{(j)} + \text{other terms}, \tag{13.108b}$$

where $l_a - f_c = l_b - f_d$. If $i = j$, the difference $l_a - f_c$ and $l_b - f_d$ are both in Δ_i, and hence Δ_i cannot be full. This contradicts the assumption that Δ_i is full. If $i \neq j$, then the difference $l_b - l_a$ is in Δ_i, and the difference $f_d - f_c$ is in Δ_j. Because $l_a - f_c = l_b - f_d$ implies that $l_b - l_a = f_d - f_c$, in this case the positive difference sets Δ_i and Δ_j cannot be disjoint. This contradicts the assumption that Δ_i and Δ_j are disjoint. We conclude that the code must be self-orthogonal. **Q.E.D.**

Because each of the J_j nonzero components of the generator polynomial $\mathbf{g}_{j+1}^{(n-1)}(D)$ is used to form one of a set of orthogonal check-sums on $e_0^{(j)}$, J_j orthogonal check-sums are formed on $e_0^{(j)}$. If $J_j = J$, $j = 0, 1, \cdots, n-2$, then each generator polynomial has the same weight J, and J orthogonal parity checks are formed on each information error bit. Therefore, if $\mathbf{u}^{(1)}(D) = 1$ and $\mathbf{u}^{(2)}(D) = \cdots = \mathbf{u}^{(n-1)}(D) = 0$, the codeword $\mathbf{v}(D) = [1, 0, 0, \cdots, 0, \mathbf{g}_1^{(n-1)}(D)]$ has weight $J+1$, and $d_{min} \leq J+1$. On the other hand, $d_{min} \geq J+1$, since otherwise the majority-logic error-correcting capability $t_{ML} = \lfloor J/2 \rfloor$ would exceed the feedback decoding correcting-capability $t = \lfloor (d_{min}-1)/2 \rfloor$, which is impossible. Hence, $d_{min} = J+1$, and the $(n, n-1, m)$ self-orthogonal code is completely orthogonalizable. If the J_j's are unequal, it can also be shown that $d_{min} = J+1$, where $J \triangleq \min_{(0 \leq j \leq n-2)} J_j$ (see Problem 13.33). Hence, in this case also, the code is completely orthogonalizable.

Robinson and Bernstein [40] have developed a procedure based on Theorem 13.4 for constructing $(n, n-1, m)$ self-orthogonal convolutional codes. Each of these codes has $J_j = J = d_{min} - 1$, $j = 0, 1, \cdots, n-2$, and error-correcting capability $t_{ML} = \lfloor J/2 \rfloor = t_{FB} = \lfloor (d_{min}-1)/2 \rfloor$. A list of these codes for $n = 2, 3, 4,$ and 5 and various values of t_{ML} and m is given in Table 13.2. In the table, each generator polynomial $\mathbf{g}_{j+1}^{(n-1)}(D)$ is identified by the set of integers $l_1, l_2, \cdots, l_{J_j}$ that specifies the positions of its nonzero components. (Note that, in this case, the octal notation used in previous tables to specify polynomials would be quite cumbersome, since the polynomials are sparse, and m can be very large.)

EXAMPLE 13.17 Difference Set Construction of a Rate $R = 1/2$ Self-Orthogonal Code

Consider the $(2, 1, 6)$ self-orthogonal code from Table 13.2(a) whose generator polynomial $\mathbf{g}^{(1)}(D)$ is identified by the set of integers $\{0, 2, 5, 6\}$; that is, $\mathbf{g}^{(1)}(D) = 1 + D^2 + D^5 + D^6$. This code has $J = 4$ and will correctly estimate the information error bit $e_0^{(0)}$ whenever the first $n_A = n(m+1) = 14$ received bits (more precisely, the $n_E = 11$ received bits checked by the $J = 4$ orthogonal check-sums) contain $t_{FB} = t_{ML} = \lfloor J/2 \rfloor = 2$ or fewer errors. The positive difference set associated with this generator polynomial is $\Delta = \{2, 5, 6, 3, 4, 1\}$. Because this positive difference set contains all positive integers from 1 to $J(J-1)/2 = 6$, it follows that the memory order $m = 6$ of this code is as small as possible[9] for any rate $R = 1/2$ self-orthogonal code with $t_{ML} = 2$. (Note that the $(2, 1, 6)$ self-orthogonal code of Example 13.10, whose generator polynomial is the reciprocal of the generator polynomial in this

[9]For feedback encoders, however, a smaller memory order is possible, as shown in Example 13.14.

TABLE 13.2: Self-orthogonal codes.

t_{ML}	m	$g^{(1)}$
1	1	{0,1}
2	6	{0,2,5,6}
3	17	{0,2,7,13,16,17}
4	35	{0,7,10,16,18,30,31,35}
5	55	{0,2,14,21,29,32,45,49,54,55}
6	85	{0,2,6,24,29,40,43,55,68,75,76,85}
7	127	{0,5,28,38,41,49,50,68,75,92,107,121,123,127}
8	179	{0,6,19,40,58,67,78,83,109,132,133,162,165,169,177,179}
9	216	{0,2,10,22,53,56,82,83,89,98,130,148,152,167,188,192,205,216}
10	283	{0,24,30,43,55,71,75,89,104,125,127.162,167,189,206,215,272,275,282,283}
11	358	{0,3,16,45,50.51,65,104,125,142,182,206,210.218,228,237,289,300,326,333,356,358}
12	425	{0,22,41,57,72,93,99,139,147,153,197,200,214,253,263,265,276,283,308,367,368,372,396,425}

(a) R = 1/2 CODES

t_{ML}	m	$g_1^{(2)}$	$g_2^{(2)}$
1	2	{0,1}	{0,2}
2	13	{0,8,9,12}	{0,6,11,13}
3	40	{0,2,6,24,29,40}	{0,3,15,28,35,36}
4	86	{0,1,27,30,61,73,81,83}	{0,18,23,37,58,62,75,86}
5	130	{0,1,6,35,32,72,100,108,120,130}	{0,23,39,57,60,74,101,103,112,116}
6	195	{0,17,46,50,52,66,88,125,150,165,168,195}	{0,26,34,47,57,58,112,121,140,181,188,193}
7	288	{0,2,7,42,45,117,163,185,195,216,229,246,255,279}	{0,8,12,27,28,64,113,131,154,160,208,219, 233,288}

(b) R = 2/3 CODES

t_{ML}	m	$g_1^{(3)}$	$g_2^{(3)}$	$g_3^{(3)}$
1	3	{0,1}	{0,2}	{0,3}
2	19	{0,3,15,19}	{0,8,17,18}	{0,6,11,13}
3	67	{0,5,15,34,35,42}	{0,31,33,44,47,56}	{0,17,21,43,49,67}
4	129	{0,9,33,37,38,97,122,129}	{0,11,13,23,62,76,79,123}	{0,19,35,50,71,77,117,125}
5	202	{0,7,27,76,113.137,155,156, 170,202}	{0,8,38.48,59,82,111,146, 150,152}	{0,12,25,26,76.81,98,107, 143,197}

(c) R = 3/4 CODES

t_{ML}	m	$g_1^{(4)}$	$g_2^{(4)}$	$g_3^{(4)}$	$g_4^{(4)}$
1	4	{0,1}	{0,2}	{0,3}	{0,4}
2	26	{0,16,20,21}	{0,2,10,15}	{0,14,17,26}	{0,11,18,24}
3	78	{0,5,26,51,55,69}	{0,6,7,41,60,72}	{0,8,11,24,44,78}	{0,10,32,47,49,77}
4	178	{0,19,59,68,85,88, 103,141}	{0,39,87,117,138,148, 154,162}	{0,2,13,25,96,118, 168,172}	{0,7,65,70,97,98, 144,178}

(d) R = 4/5 CODES

Adapted from [40].

example, also has $J = 4$ orthogonal parity checks and $t_{ML} = 2$. This illustrates that taking the reciprocal of a generator polynomial does not change its associated positive difference set.)

EXAMPLE 13.18 Difference Set Construction of a Rate $R = 2/3$ Self-Orthogonal Code

Consider the $(3, 2, 13)$ self-orthogonal code from Table 13.2(b) whose generator polynomials $\mathbf{g}_1^{(2)}(D)$ and $\mathbf{g}_2^{(2)}(D)$ are identified by the sets of integers $\{0, 8, 9, 12\}$ and $\{0, 6, 11, 13\}$, respectively. This code was previously shown to be self-orthogonal with $J = 4$ in Example 13.12. The positive difference sets associated with $\mathbf{g}_1^{(2)}(D)$ and $\mathbf{g}_2^{(2)}(D)$ are $\Delta_0 = \{8, 9, 12, 1, 4, 3\}$ and $\Delta_1 = \{6, 11, 13, 5, 7, 2\}$, respectively. These positive difference sets are full and disjoint, as required by Theorem 13.4 for any self-orthogonal code.

We can obtain a self-orthogonal $(n, 1, m)$ code from a self-orthogonal $(n, n-1, m)$ code in the following way. Consider an $(n, n-1, m)$ self-orthogonal code with generator polynomials $\mathbf{g}_1^{(n-1)}, \mathbf{g}_2^{(n-1)}, \cdots, \mathbf{g}_{n-1}^{(n-1)}$ and let J_j be the weight of $\mathbf{g}_{j+1}^{(n-1)}(D)$, $j = 0, 1, \cdots, n-2$. Then, we obtain an $(n, 1, m)$ code with generator polynomials $\mathbb{h}^{(1)}(D), \mathbb{h}^{(2)}(D), \cdots, \mathbb{h}^{(n-1)}(D)$ from this $(n, n-1, m)$ self-orthogonal code by setting

$$\mathbb{h}^{(j+1)}(D) = \mathbf{g}_{j+1}^{(n-1)}(D), \quad j = 0, 1, \cdots, n-2. \tag{13.109}$$

Because the positive difference sets associated with the generator polynomials $\mathbf{g}_{j+1}^{(n-1)}(D)$, $j = 0, 1, \cdots, n-2$, of the $(n, n-1, m)$ self-orthogonal code are full and disjoint, the positive difference sets associated with the generator polynomial $\mathbb{h}^{(j+1)}(D)$, $j = 0, 1, \cdots, n-2$ of the $(n, 1, m)$ code are also full and disjoint. It is shown in Problem 13.35 that this condition is necessary and sufficient for the $(n, 1, m)$ code to be self-orthogonal. Since J_j is the weight of $\mathbb{h}^{(j+1)}(D)$, $j = 0, 1, \cdots, n-2$, there are a total of $J \triangleq J_0 + J_1 + \cdots + J_{n-2}$ orthogonal check-sums on the information error bit $e_0^{(0)}$, and the $(n, 1, m)$ code has a majority-logic error-correcting capability of $t_{ML} = \lfloor J/2 \rfloor$. Because the information sequence $\mathbf{u}(D) = 1$ results in the codeword $\mathbf{v}(D) = [1, \mathbb{h}^{(1)}(D), \cdots, \mathbb{h}^{(n-1)}(D)]$, which has weight $J + 1$, it follows that $d_{min} \leq J + 1$; but $t_{FB} = \lfloor (d_{min} - 1)/2 \rfloor \geq t_{ML} = \lfloor J/2 \rfloor$ implies that $d_{min} \geq J + 1$. Hence, $d_{min} = J + 1$, and the $(n, 1, m)$ self-orthogonal code must be completely orthogonalizable.

EXAMPLE 13.19 Difference Set Construction of a Rate $R = 1/3$ Self-Orthogonal Code

Consider the $(3, 1, 13)$ self-orthogonal code with generator polynomials $\mathbb{h}^{(1)}(D) = 1 + D^8 + D^9 + D^{12}$ and $\mathbb{h}^{(2)}(D) = 1 + D^6 + D^{11} + D^{13}$ derived from the $(3, 2, 13)$ self-orthogonal code of Example 13.18. This code has $J = J_0 + J_1 = 4 + 4 = 8$ orthogonal check-sums on the information error bit $e_0^{(0)}$, and hence it has an error-correcting capability of $t_{FB} = t_{ML} = \lfloor J/2 \rfloor = 4$; that is, it correctly estimates $e_0^{(0)}$ whenever there are four or fewer errors in the first $n_A - n(m+1) = 42$ received bits (more precisely, the n_E received bits (see Problem 13.36) checked by the $J = 8$ orthogonal check-sums). In a similar fashion, every $(n, n-1, m)$ self-orthogonal code in Table 13.2 can be converted to an $(n, 1, m)$ self-orthogonal code.

If a self-orthogonal code is used with feedback decoding, each successive block of information error bits is estimated using the same decoding rule, and the error-correcting capability is the same as for block zero, assuming that the previous estimates have all been correct. This property was illustrated in equations (13.52) through (13.55) for the self-orthogonal code of Example 13.10. On the other hand, if all previous estimates have not been correct, postdecoding errors appear in the syndrome equations, as illustrated in (13.62) for the same self-orthogonal code. In this case, the modified syndrome equations are still orthogonal, but postdecoding errors as well as transmission errors can cause further decoding errors. This results in the error propagation effect of feedback decoders discussed in Section 13.5.

We now demonstrate that self-orthogonal codes possess the automatic resynchronization property with respect to error propagation. First, note that whenever a decoding error is made, that is, $\mathbb{e}_l^{(j)} = 1$, it is fed back and reduces the total number of 1's stored in syndrome registers. This observation follows from the fact that more than half the syndrome bits that are used to estimate $\mathbb{e}_l^{(j)}$ must be 1's in order to cause the decoding error, and hence when $\mathbb{e}_l^{(j)} = 1$ is fed back it changes more 1's to 0's than 0's to 1's, thereby reducing the total weight in the syndrome registers. Now, assume that there are no more transmission errors after time unit l. Then, beginning at time unit $l + m$, only 0's can enter the leftmost stages of the syndrome registers. No matter what the contents of the registers are at time unit $l + m$, they must soon clear to all 0's since only 0's are shifted in, and each estimate of 1 reduces the total number of 1's in the registers. (Clearly, estimates of 0 have no effect on the register weights.) Once the syndrome registers have resynchronized to all 0's, no further decoding errors are possible unless there are additional transmission errors; that is, error propagation has been terminated.

13.7.2 Orthogonalizable Codes

Completely orthogonalizable convolutional codes can also be constructed using a trial-and-error approach. In this case, some of the orthogonal parity checks are formed from sums of syndrome bits, and these codes are not self-orthogonal. The $(2, 1, 5)$ code of Example 13.11 contains $J = 4$ orthogonal parity checks constructed by trial and error, and hence $d_{min} \geq 5$. Because the two generator polynomials have a total weight of 5, d_{min} must equal 5, and therefore the code is completely orthogonalizable with error-correcting capability $t_{FB} = t_{ML} = \lfloor J/2 \rfloor = 2$. Similarly, the $(3, 1, 4)$ code of Example 13.13 contains $J = 6$ orthogonal parity checks constructed by trial and error, is completely orthogonalizable, and has error-correcting capability $t_{FB} = t_{ML} = \lfloor J/2 \rfloor = 3$. A list of rate $R = 1/2$ and 1/3 orthogonalizable codes constructed by Massey [6] is given in Table 13.3. (The generator polynomials are specified using the notation of Table 13.2. The notation used to describe the rules for forming the orthogonal check-sums in the rate $R = 1/3$ case is explained in the following example.)

TABLE 13.3: Orthogonalizable codes.

t_{ML}	m	$\mathbf{g}^{(1)}$	**Orthogonalization Rules**†
2	5	{0,3,4,5}	(1, 5)
3	11	{0,6,7,9,10,11}	(1, 3, 10)(4, 8, 11)
4	21	{0,11,13,16,17,19,20,21}	(2, 3, 6, 19)(4, 14, 20)(1, 5, 8, 15, 21)
5	35	{0,18,19,27,28,29,30,32,33,35}	(1, 9, 28)(10, 20, 29)(11, 30, 31) (13, 21, 23, 32)(14, 33, 34)(2, 3, 16, 24, 26, 35)
6	51	{0,26,27,39,40,41,42,44,45,47,48,51}	(1, 13, 40)(14, 28, 41)(15, 42, 43) (17, 29, 31, 44)(18, 45, 46)(2, 3, 20, 32, 34, 47) (21, 35, 48, 49, 50)(24, 30, 33, 36, 38, 51)

(a) R = 1/2 CODES

t_{ML}	m	$\mathbf{g}^{(1)}$	$\mathbf{g}^{(2)}$	**Orthogonalization Rules**
3	4	{0,1}	{0,2,3,4}	$(0^1)(0^2)(1^1)(2^2)(1^2 3^2)(2^1 4^2)$
4	7	{0,1,7}	{0,2,3,4,6}	$(0^1)(0^2)(1^1)(2^2)(1^2 3^2)(2^1 4^2)(7^1)(3^1 5^1 6^1 6^2)$
5	10*	{0,1,9}	{0,1,2,3,5,8,9}	$(0^1)(0^2)(1^1)(2^1 2^2)(9^1)(3^2 4^2)(3^1 5^1 5^2)$ $(1^2 4^1 6^1 6^2)(8^1 8^2)(7^2 9^2 10^2)$
6	17*	{0,4,5,6,7,9,12,13,16}	{0,1,14,15,16}	$(0^1)(0^2)(1^1 1^2)(4^1)(5^1)(2^2 6^1)(14^2)(7^1 10^1 11^1 11^2)$ $(3^2 5^2 9^1)(6^2 8^2 12^1)(3^2 16^2 17^2)(4^2 10^2 12^2 16^1)$
7	22	{0,4,5,6,7,9,12,13,16, 19,20,21}	{0,1,20,22}	$(0^1)(0^2)(1^1 1^2)(4^1)(5^1)(2^2 6^1)(7^1 10^1 11^1 11^2)$ $(3^2 5^2 9^1)(19^2 20^2)(22^2)(6^2 8^2 12^1)(4^2 10^2 12^2 16^1)$ $(3^1 7^2 13^2 15^2 19^1)(9^2 13^1 14^2 18^1 20^1 21^1 21^2)$
8	35	{0,4,5,6,7,9,12,16,17, 30,31}	{0,1,22,25,35}	$(0^1)(0^2)(1^1 1^2)(4^1)(5^1)(2^2 6^1)(22^2)(7^1 10^1 11^1 11^2)$ $(3^1 25^2)(3^2 5^2 9^1)(6^2 8^2 12^1)(7^2 14^1 17^1 18^1 18^2)$ $(9^2 16^1 19^1 20^1 20^2)(14^2 15^2 35^2)(12^2 21^2 28^1 31^1 32^1)$ $(10^2 13^2 19^2 26^2 29^2 30^1)$

(b) R = 1/3 CODES

Adapted from [6].

*The actual value of m for these codes is less than shown, and the error-correcting capability t_{ML} listed is achieved by adding 0's to the generator sequences.

†$(x, y, \cdots)$ indicates that the sum $s_x + s_y + \cdots$ forms an orthogonal check sum on $e_0^{(0)}$. Only those orthogonal equations that require a sum of syndrome bits are listed.

EXAMPLE 13.20 Trial-and-Error Construction of a Rate $R = 1/3$ Orthogonalizable Code

Consider the (3, 1, 7) code listed in Table 11.3(b) whose generator polynomials $\mathbf{g}^{(1)}(D)$ and $\mathbf{g}^{(2)}(D)$ are specified by sets of integers {0, 1, 7} and {0, 2, 3, 4, 6}; that is, $\mathbf{g}^{(1)}(D) = 1 + D + D^7$ and $\mathbf{g}^{(2)}(D) = 1 + D^2 + D^3 + D^4 + D^6$. The rules for forming $J = 8$ orthogonal parity checks are given by the set

$$\left\{(0^1), (0^2), (1^1), (2^2), (1^2 3^2), (2^1 4^2), (7^1), (3^1 5^1 6^1 6^2)\right\},$$

where the notation (k^i) indicates that the syndrome bit $s_k^{(i)}$ forms an orthogonal check-sum on $e_0^{(0)}$, and $(k^i l^j)$ indicates that the sum $s_k^{(i)} + s_l^{(j)}$ forms an orthogonal check-sum on $e_0^{(0)}$. The $J = 8$ orthogonal parity checks on $e_0^{(0)}$ are thus given by the set

$$\left\{s_0^{(1)}, s_0^{(2)}, s_1^{(1)}, s_2^{(2)}, s_1^{(2)} + s_3^{(2)}, s_2^{(1)} + s_4^{(2)}, s_7^{(1)}, s_3^{(1)} + s_5^{(1)} + s_6^{(1)} + s_6^{(2)}\right\}.$$

This code has $d_{min} = 9$, is completely orthogonalizable, and has error-correcting capability $t_{FB} = t_{ML} = \lfloor J/2 \rfloor = 4$.

Note that (n, k, m) orthogonalizable codes can achieve a given majority-logic error-correcting capability t_{ML} with a smaller memory order m than self-orthogonal codes, owing to the added flexibility available in using sums of syndrome bits to form orthogonal parity checks for orthogonalizable codes. The major disadvantage of orthogonalizable codes is that they do not possess the automatic resynchronization property that limits error propagation when used with feedback decoding.

PROBLEMS

13.1 Consider the (2, 1, 3) encoder with

$$\mathbf{G}(D) = [1 + D^2 + D^3 \quad 1 + D + D^2 + D^3].$$

a. Draw the code tree for an information sequence of length $h = 4$.
b. Find the codeword corresponding to the information sequence $\mathbf{u} = (1\,0\,0\,1)$.

13.2 For a binary-input, Q-ary output symmetric DMC with equally likely input symbols, show that the output symbol probabilities satisfy (13.7).

13.3 Consider the (2, 1, 3) encoder of Problem 13.1.
a. For a BSC with $p = .045$, find an integer metric table for the Fano metric.
b. Decode the received sequence

$$\mathbf{r} = (1\,1,\ 0\,0,\ 1\,1,\ 0\,0,\ 0\,1,\ 1\,0,\ 1\,1)$$

using the stack algorithm. Compare the number of decoding steps with the number required by the Viterbi algorithm.
c. Repeat (b) for the received sequence

$$\mathbf{r} = (1\,1,\ 1\,0,\ 0\,0,\ 0\,1,\ 1\,0,\ 0\,1,\ 0\,0).$$

Compare the final decoded path with the results of Problem 12.6, where the same received sequence is decoded using the Viterbi algorithm.

13.4 Consider the (2, 1, 3) encoder of Problem 13.1.
a. For the binary-input, 8-ary output DMC of Problem 12.4, find an integer metric table for the Fano metric. (*Hint*: Scale each metric by an appropriate factor and round to the nearest integer.)
b. Decode the received sequence

$$\mathbf{r} = (1_2 1_1,\ 1_2 0_1,\ 0_3 0_1,\ 0_1 1_3,\ 1_2 0_2,\ 0_3 1_1,\ 0_3 0_2)$$

using the stack algorithm. Compare the final decoded path with the result of Problem 12.5(b), where the same received sequence is decoded using the Viterbi algorithm.

13.5 Consider the (2, 1, 3) encoder of Problem 13.1. For a binary-input, continuous-output AWGN channel with $E_s/N_0 = 1$, use the stack algorithm and the AWGN channel Fano metric from (13.16) to decode the received sequence $\mathbf{r} = (+1.72, +0.93, +2.34, -3.42, -0.14, -2.84, -1.92, +0.23, +0.78, -0.63, -0.05, +2.95, -0.11, -0.55)$. Compare the final decoded path with the result of Problem 12.7, where the same received sequence is decoded using the Viterbi algorithm.

13.6 Repeat parts (b) and (c) of Problem 13.3 with the size of the stack limited to 10 entries. When the stack is full, each additional entry causes the path on the bottom of the stack to be discarded. What is the effect on the final decoded path?

13.7 **a.** Repeat Example 13.5 using the stack-bucket algorithm with a bucket quantization interval of 5. Assume the bucket intervals are $\cdots +4$ to 0, -1 to -5, -6 to -10, $\cdots$.

b. Repeat part (a) for a quantization interval of 9, where the bucket intervals are $\cdots +8$ to 0, -1 to -9, -10 to -18, $\cdots$.

13.8 Repeat Example 13.7 for the Fano algorithm with threshold increments of $\Delta = 5$ and $\Delta = 10$. Compare the final decoded path and the number of computations to the results of Examples 13.7 and 13.8. Also compare the final decoded path with the results of the stack-bucket algorithm in Problem 13.7.

13.9 Using a computer program, verify the results of Figure 13.13, and plot ρ as a function of E_b/N_0 (dB) for $R = 1/5$ and $R = 4/5$.

13.10 Show that the Pareto exponent ρ satisfies $\lim_{R\to 0} \rho = \infty$ and $\lim_{R\to C} \rho = 0$ for fixed channel transition probabilities. Also show that $\partial R/\partial \rho < 0$.

13.11 **a.** For a BSC with crossover probability p, plot both the channel capacity C and the cut-off rate R_0 as functions of p. (*Note*: $C = 1 + p\log_2 p + (1 - p)\log_2(1 - p)$.)

b. Repeat part (a) by plotting C and R_0 as functions of the SNR E_b/N_0. What is the SNR difference required to make $C = R_0 = 1/2$?

13.12 **a.** Calculate R_0 for the binary-input, 8-ary output DMC of Problem 12.4.

b. Repeat Example 13.9 for the DMC of part (a) and a code rate of $R = 1/2$. (*Hint:* Use a computer program to find ρ from (13.23) and (13.24).)

c. For the value of ρ calculated in part (b), find the buffer size B needed to guarantee an erasure probability of 10^{-3} using the values of L, A, and μ given in Example 13.9.

13.13 **a.** Sketch $P_{erasure}$ versus E_b/N_0 for a rate $R = 1/2$ code on a BSC using the values of L, A, μ, and B given in Example 13.9.

b. Sketch the required buffer size B to guarantee an erasure probability of 10^{-3} as a function of E_b/N_0 for a rate $R = 1/2$ code on a BSC using the values of L, A, and μ given in Example 13.9. (*Hint:* ρ can be found as a function of E_b/N_0 using the results of Problem 13.9.)

13.14 Repeat Problem 13.4 using the integer metric tables of Figures 13.14(a) and (b). Note any changes in the final decoded path or the number of decoding steps.

13.15 For a BSC with crossover probability p, plot both the computational cutoff rate R_0 from (13.27) and R_{max} from (13.35) as functions of p.

13.16 Find the complete CDFs of the $(2, 1, 3)$ optimum free distance code in Table 12.1(c) and the $(2, 1, 3)$ quick-look-in code in Table 12.2. Which code has the superior distance profile?

13.17 Show that for the BSC, the received sequence $\mathbf{r}$ is a codeword if and only if the error sequence $\mathbf{e}$ is a codeword.

13.18 Using the definition of the $\mathbb{H}$ matrix for rate $R = 1/2$ systematic codes given by (13.42), show that (13.41) and (13.46) are equivalent.

13.19 Draw the complete encoder/decoder block diagram for Example 13.11.

13.20 Consider the $(2, 1, 11)$ systematic code with

$$\mathbf{g}^{(1)}(D) = 1 + D + D^3 + D^5 + D^8 + D^9 + D^{10} + D^{11}.$$

a. Find the parity-check matrix $\mathbb{H}$.

b. Write equations for the syndrome bits $s_0, s_1, \cdots, s_{11}$ in terms of the channel error bits.

c. Write equations for the modified syndrome bits $s'_l, s'_{l+1}, \cdots, s'_{l+11}$, assuming that the effect of error bits prior to time unit l has been removed by feedback.

13.21 Consider the $(3, 2, 13)$ code of Example 13.12 and the $(3, 1, 4)$ code of Example 13.13.

a. Find the generator matrix $\mathbb{G}(D)$.

b. Find the parity-check matrix $\mathbb{H}(D)$.

c. Show that in each case $\mathbb{G}(D)\mathbb{H}^T(D) = \mathbb{0}$.

13.22 Consider the $(3, 2, 13)$ code of Example 13.12.

a. Write equations for the unmodified syndrome bits $s_l, s_{l+1}, \cdots, s_{l+13}$ that include the effect of error bits prior to time unit l (assume $l \geq 13$).

b. Find a set of orthogonal parity checks for both $\mathbb{e}_l^{(0)}$ and $\mathbb{e}_l^{(1)}$ from the unmodified syndrome equations.

c. Determine the resulting majority-logic error-correcting capability t_{ML} and the effective decoding length n_E and compare these values with those in Example 13.12.

d. Draw the block diagram of the decoder. (Note that in this case the decoding estimates are not fed back to modify the syndrome. This alternative to feedback decoding is called *definite decoding*.)

13.23 Find a rate $R = 1/2$ nonsystematic feedforward encoder with the smallest possible value of m such that $J = 4$ orthogonal parity checks can be formed on the error bits $e_0^{(0)}$ and $e_0^{(1)}$.

13.24 Prove (13.65).

13.25 Prove (13.66).

13.26 Prove that if the weighting factors w_i, $i = 0, 1, \cdots, J$, are calculated using (13.64) and (13.67) for a BSC with crossover probability p, the APP threshold decoding rule is equivalent to the majority-logic decoding rule only if all J orthogonal check-sums include the same number of bits, that is, only if $n_1 = n_2 = \cdots = n_J$.

13.27 Consider an (n, k, m) convolutional code with minimum distance $d_{min} = 2t_{FB} + 1$. Prove that there is at least one error sequence $\mathbb{e}$ with weight $t_{FB} + 1$ in its first $(m + 1)$ blocks for which a feedback decoder will decode $\mathbb{u}_0$ incorrectly.

13.28 Consider the $(2, 1, 11)$ code of Problem 13.20.

a. Find the minimum distance d_{min}.

b. Is this code self-orthogonal?

c. Find the maximum number of orthogonal parity checks that can be formed on $e_0^{(0)}$.

d. Is this code completely orthogonalizable?

13.29 Consider the $(3, 1, 3)$ nonsystematic feedforward encoder with $\mathbb{G}_{ns}(D) = [1 + D + D^3 \quad 1 + D^3 \quad 1 + D + D^2]$.

a. Following the procedure in Example 13.14, convert this code to a $(3, 1, 3)$ systematic feedforward encoder with the same d_{min}.

b. Find the generator matrix $\mathbb{G}_s(D)$ of the systematic feedforward encoder.

c. Find the minimum distance d_{min}.

13.30 Consider the $(2, 1, 6)$ code of Example 13.15.

a. Estimate the bit-error probability $P_b(E)$ of a feedback decoder with error-correcting capability t_{FB} on a BSC with small crossover probability p.

b. Repeat (a) for a feedback majority-logic decoder with error-correcting capability t_{ML}.

c. Compare the results of (a) and (b) for $p = 10^{-2}$.

13.31 Repeat Problem 13.30 for the $(2, 1, 5)$ code of Example 13.16.

13.32 Find and compare the memory orders of the following codes:

a. the best rate $R = 1/2$ self-orthogonal code with $d_{min} = 9$.
b. the best rate $R = 1/2$ orthogonalizable code with $d_{min} = 9$.
c. the best rate $R = 1/2$ systematic code with $d_{min} = 9$.
d. the best rate $R = 1/2$ nonsystematic code with $d_{free} = 9$.

13.33 Consider an $(n, n-1, m)$ self-orthogonal code with J_j orthogonal check-sums on $e_0^{(j)}$, $j = 0, 1, \cdots, n-2$. Show that $d_{min} = J + 1$, where $J \triangleq \min_{(0 \le j \le n-2)} J_j$.

13.34 Consider the (2, 1, 17) self-orthogonal code in Table 13.2(a).
a. Form the orthogonal check-sums on information error bit $e_0^{(0)}$.
b. Draw the block diagram of the feedback majority-logic decoder for this code.

13.35 Consider an $(n, 1, m)$ systematic code with generator polynomials $\mathbf{g}^{(j)}(D)$, $j = 1, 2, \cdots, n-1$. Show that the code is self-orthogonal if and only if the positive difference sets associated with each generator polynomial are full and disjoint.

13.36 Find the effective decoding length n_E for the (3, 1, 13) code of Example 13.19.

13.37 Consider the (2, 1, 11) orthogonalizable code in Table 13.3(a).
a. Form the orthogonal check-sums on information error bit $e_0^{(0)}$.
b. Draw the block diagram of the feedback majority-logic decoder for this code.

BIBLIOGRAPHY

1. J. M. Wozencraft, "Sequential Decoding for Reliable Communication," *IRE Conv. Rec.* 5 (pt. 2): 11–25, 1957.

2. J. M. Wozencraft and B. Reiffen, *Sequential Decoding*. MIT Press, Cambridge, 1961.

3. R. M. Fano, "A Heuristic Discussion of Probabilistic Decoding," *IEEE Trans. Inform. Theory*, IT-9: 64–74, April 1963.

4. K. Zigangirov, "Some Sequential Decoding Procedures," *Prob. Peredachi Inform.*, 2: 13–25, 1966.

5. F. Jelinek, "A Fast Sequential Decoding Algorithm Using a Stack," *IBM J. Res. Dev.*, 13: 675–85, November 1969.

6. J. L. Massey, *Threshold Decoding*. MIT Press, Cambridge, 1963.

7. J. L. Massey, "Variable-Length Codes and the Fano Metric," *IEEE Trans. Inform. Theory*, IT-18: 196–98, January 1972.

8. J. M. Geist, "Search Properties of Some Sequential Decoding Algorithms," *IEEE Trans. Inform. Theory*, IT-19: 519–26, July 1973.

9. J. M. Geist, "An Empirical Comparison of Two Sequential Decoding Algorithms," *IEEE Trans. Commun. Technol.*, COM-19: 415–19, August 1971.

10. J. E. Savage, "Sequential Decoding—The Computation Problem," *Bell Syst. Tech. J.*, 45: 149–75, January 1966.

11. I. M. Jacobs and E. R. Berlekamp, "A Lower Bound to the Distribution of Computation for Sequential Decoding," *IEEE Trans. Inform. Theory*, IT-13: 167–74, April 1967.

12. F. Jelinek, "An Upper Bound on Moments of Sequential Decoding Effort," *IEEE Trans. Inform. Theory*, IT-15: 140–49, January 1969.

13. G. D. Forney, Jr., "Convolutional Codes III: Sequential Decoding," *Inform. Control*, 25: 267–97, July 1974.

14. G. D. Forney, Jr., and E. K. Bower, "A High-Speed Sequential Decoder: Prototype Design and Test," *IEEE Trans. Commun. Technol.*, COM-19: 821–35, October 1971.

15. I. Richer, "Sequential Decoding with a Small Digital Computer," *Tech. Rep.* 491, MIT Lincoln Laboratory, January 1972.

16. P. R. Chevillat and D. J. Costello, Jr., "An Analysis of Sequential Decoding for Specific Time-Invariant Convolutional Codes," *IEEE Trans. Inform. Theory*, IT-24: 443–51, July 1978.

17. P. R. Chevillat and D. J. Costello, Jr., "Distance and Computation in Sequential Decoding," *IEEE Trans. Commun.*, COM-24: 440–47, April 1976.

18. J. A. Heller and I. M. Jacobs, "Viterbi Decoding for Satellite and Space Communication," *IEEE Trans. Commun. Technol.*, COM-19: 835–48, October 1971.

19. P. R. Chevillat and D. J. Costello, Jr., "A Multiple Stack Algorithm for Erasurefree Decoding of Convolutional Codes," *IEEE Trans. Commun.*, COM-25: 1460–70, December 1977.

20. D. D. Falconer, "A Hybrid Sequential and Algebraic Decoding Scheme," Sc.D. thesis, MIT, Cambridge, 1967.

21. F. Jelinek and J. Cocke, "Bootstrap Hybrid Decoding for Symmetrical Binary Input Channels," *Inform. Control*, 18: 261–98, April 1971.

22. D. Haccoun and M. J. Ferguson, "Generalized Stack Algorithms for Decoding Convolutional Codes," *IEEE Trans. Inform. Theory*, IT-21: 638–51, November 1975.

23. A. J. P. de Paepe, A. J. Vinck, and J. P. M. Schalkwijk, "A Class of Binary Rate 1/2 Convolutional Codes That Allows an Improved Stack Decoder," *IEEE Trans. Inform. Theory*, IT-26: 389–92, July 1980.

24. R. Johannesson, "Robustly Optimal Rate One-Half Binary Convolutional Codes," *IEEE Trans. Inform. Theory*, IT-21: 464–68, July 1975.

25. R. Johannesson, "Some Long Rate One-Half Binary Convolutional Codes with an Optimum Distance Profile," *IEEE Trans. Inform. Theory*, IT-22: 629–31, September 1976.

26. R. Johannesson, "Some Rate 1/3 and 1/4 Binary Convolutional Codes with an Optimum Distance Profile," *IEEE Trans. Inform. Theory*, IT-23: 281–83, March 1977.

27. R. Johannesson and E. Paaske, "Further Results on Binary Convolutional Codes with an Optimum Distance Profile," *IEEE Trans. Inform. Theory*, IT-24: 264–68, March 1978.

28. M. Cedervall and R. Johannesson, "A Fast Algorithm for Computing Distance Spectrum of Convolutional Codes," *IEEE Trans. Inform. Theory*, IT-35: 1146–59, 1989.

29. R. Johannesson and P. Stahl, "New Rate 1/2, 1/3, and 1/4 Binary Convolutional Encoders with an Optimum Distance Profile," *IEEE Trans. Inform. Theory*, IT-45: 1653–58, July 1999.

30. R. Johannesson and K. S. Zigangirov, *Fundamentals of Convolutional Coding*. IEEE Press, Piscataway, 1999.

31. G. D. Forney, Jr., "Use of a Sequential Decoder to Analyze Convolutional Code Structure," *IEEE Trans. Inform. Theory*, IT-16: 793–95, November 1970.

32. P. R. Chevillat, "Fast Sequential Decoding and a New Complete Decoding Algorithm," Ph.D. thesis, IIT, Chicago, 1976.

33. J. L. Massey and R. W. Liu, "Application of Lyapunov's Direct Method to the Error-Propagation Effect in Convolutional Codes," *IEEE Trans. Inform. Theory*, IT-10: 248–50, July 1964.

34. J. L. Massey, "Uniform Codes," *IEEE Trans. Inform. Theory*, IT-12: 132–34, April 1966.

35. D. D. Sullivan, "Error-Propagation Properties of Uniform Codes," *IEEE Trans. Inform. Theory*, IT-15: 152–61, January 1969.

36. J. P. Robinson, "Error Propagation and Definite Decoding of Convolutional codes," *IEEE Trans. Inform. Theory*, IT-14: 121–28, January 1968.

37. T. N. Morrissey, Jr., "A Unified Markovian Analysis of Decoders for Convolutional Codes," *Tech. Rep. EE-687*, Department of Electrical Engineering, University of Notre Dame, October 1968.

38. D. J. Costello, Jr., "Free Distance Bounds for Convolutional Codes," *IEEE Trans. Inform. Theory*, IT-20: 356–65, May 1974.

39. L. D. Rudolph, "Generalized Threshold Decoding of Convolutional Codes," *IEEE Trans. Inform. Theory*, IT-16: 739–45, November 1970.

40. J. P. Robinson and A. J. Bernstein, "A Class of Binary Recurrent Codes with Limited Error PropagaTion," *IEEE Trans. Inform. Theory*, IT-13: 106–13, January 1967.