Chapter 12

Convolutional Codes

12.1 Introduction and Basic Notation

Convolutional codes are linear codes that have additional structure in the generator matrix so that the encoding operation can be viewed as a filtering — or convolution — operation. Convolutional codes are widely used in practice, with several hardware implementations available for encoding and decoding. A convolutional encoder may be viewed as nothing more than a set of digital filters — linear, time-invariant systems — with the code sequence being the interleaved output of the filter outputs. Convolutional codes are often preferred in practice over block codes, because they provide excellent performance when compared with block codes of comparable encode/decode complexity. Furthermore, they were among the earliest codes for which effective soft-decision decoding algorithms were developed.

Whereas block codes take discrete blocks of k symbols and produce therefrom blocks of n symbols that depend only on the k input symbols, convolutional codes are frequently viewed as *stream codes*, in that they often operate on continuous streams of symbols not partitioned into discrete message blocks. However, they are still rate $R = k/n$ codes, accepting k new symbols at each time step and producing n new symbols. The arithmetic can, of course, be carried out over any field, but throughout this chapter and, in fact, in most of the convolutional coding literature, the field $GF(2)$ is employed.

We represent sequences and transfer functions as power series in the variable x.[1] A sequence $\{\ldots, m_{-2}, m_{-1}, m_0, m_1, m_2, \ldots\}$ with elements from a field $\mathbb{F}$ is represented as a formal **Laurent series** $m(x) = \sum_{l=-\infty}^{\infty} m_l x^l$. The set of all Laurent series over $\mathbb{F}$ is a field, which is usually denoted as $\mathbb{F}((x))$. Thus, $m(x) \in \mathbb{F}((x))$.

For multiple input streams we use a superscript, so $m^{(1)}(x)$ represents the first input stream and $m^{(2)}(x)$ represents the second input stream. For multiple input streams, it is convenient to collect the input streams into a single (row) vector, as in

$$m(x) = \left[m^{(1)}(x) \quad m^{(2)}(x)\right] \in \mathbb{F}((x))^2.$$

A convolutional encoder is typically represented as sets of digital (binary) filters.

Example 12.1 Figure 12.1 shows an example of a convolutional encoder. (Recall that the D blocks represent 1-bit storage devices, or D flip-flops.) The input stream m_k passes through two filters (sharing memory elements) producing two output streams

$$c_k^{(1)} = m_k + m_{k-2} \quad \text{and} \quad c_k^{(2)} = m_k + m_{k-1} + m_{k-2}.$$

These two streams are interleaved together to produce the coded stream c_k. Thus, for every bit of input, there are two coded output bits, resulting in a rate $R = 1/2$ code.

[1] The symbol D is sometimes used instead of x. The Laurent series representation may be called the D-transform in this case.

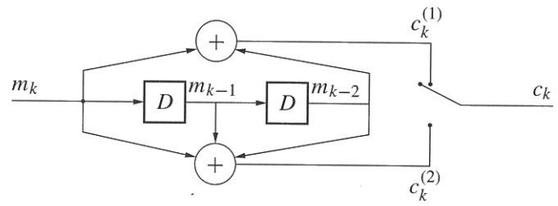


Figure 12.1: A rate $R = 1/2$ convolutional encoder.

It is conventional to assume that the memory elements are initialized with all zeros at the beginning of transmission.

For the input stream $\mathbf{m} = \{1, 1, 0, 0, 1, 0, 1\}$, the outputs are

$$\mathbf{c}^{(1)} = \{1, 1, 1, 1, 1, 0, 0, 0, 1\} \quad \text{and} \quad \mathbf{c}^{(2)} = \{1, 0, 0, 1, 1, 1, 0, 1, 1\}$$

and the interleaved stream is

$$\mathbf{c} = \{11, 10, 10, 11, 11, 01, 00, 01, 11\}$$

(where commas separate the pairs of outputs at a single input time). We can represent the transfer function from input $m(x)$ to output $c^{(1)}(x)$ as $g^{(1)}(x) = 1+x^2$, and the transfer function from $m(x)$ to output $c^{(2)}(x)$ as $g^{(2)}(x) = 1+x+x^2$. The input stream $\mathbf{m} = \{1, 1, 0, 0, 1, 0, 1\}$ can be represented as $m(x) = 1 + x + x^4 + x^6 \in GF(2)[[x]]$. The outputs are

$$c^{(1)}(x) = m(x)g_1(x) = (1 + x + x^4 + x^6)(1 + x^2) = 1 + x + x^2 + x^3 + x^4 + x^8$$
$$c^{(2)}(x) = m(x)g_2(x) = (1 + x + x^4 + x^6)(1 + x + x^2) = 1 + x^3 + x^4 + x^5 + x^7 + x^8.$$

□

A rate $R = k/n$ convolutional code has associated with it an encoder, a $k \times n$ matrix transfer function $G(x)$ called the *transfer function matrix*. For the rate $R = 1/2$ code of this example,

$$G_a(x) = \left[1 + x^2 \quad 1 + x + x^2\right].$$

The transfer function matrix of a convolutional code does not always have only polynomial entries, as the following example illustrates.

Example 12.2 Consider the convolutional transfer function matrix

$$G_b(x) = \left[1 \quad \tfrac{1+x+x^2}{1+x^2}.\right].$$

Since there is a 1 in the first column the input stream appears explicitly in the interleaved output data; this is a *systematic* convolutional encoder.

A realization (in controller form) for this encoder is shown in Figure 12.2. For the input sequence $m(x) = 1 + x + x^2 + x^3 + x^4 + x^8$, the first output is

$$c^{(1)}(x) = m(x) = 1 + x + x^2 + x^3 + x^4 + x^8$$

and the second output is

$$c^{(2)}(x) = \frac{(1 + x + x^2 + x^3 + x^4 + x^8)(1 + x + x^2)}{1 + x^2} = 1 + x^3 + x^4 + x^5 + x^7 + x^8 + x^{10} + \cdots$$

as can be verified by long division. □

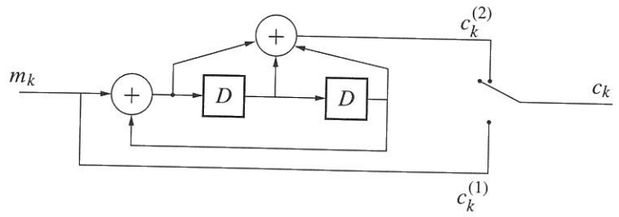


Figure 12.2: A systematic $R = 1/2$ encoder.

An encoder that has only polynomial entries in its transfer function matrix is said to be a **feedforward** encoder or an FIR encoder. An encoder that has rational functions in its transfer function matrix is said to be a **feedback** or IIR encoder.

For a rate $R = k/n$ code with $k > 1$, there are k input message sequences (usually obtained by splitting a single message sequence into k streams). Let

$$\mathbf{m}(x) = [m^{(1)}(x), m^{(2)}(x), \ldots, m^{(k)}(x)]$$

and

$$G(x) = \begin{bmatrix} g^{(1,1)}(x) & g^{(1,2)}(x) & \cdots & g^{(1,n)}(x) \\ g^{(2,1)}(x) & g^{(2,2)}(x) & \cdots & g^{(2,n)}(x) \\ \vdots & & & \\ g^{(k,1)}(x) & g^{(k,2)}(x) & \cdots & g^{(k,n)}(x) \end{bmatrix}. \tag{12.1}$$

The output sequences are represented as

$$\mathbf{c}(x) = [c^{(1)}(x), c^{(2)}(x), \ldots, c^{(n)}(x)] = \mathbf{m}(x)G(x).$$

A transfer function matrix $G(x)$ is said to be systematic if an identity matrix can be identified among the elements of $G(x)$. (That is, if by row and/or column permutations of $G(x)$, an identity matrix can be obtained.)

Example 12.3 For a rate $R = 2/3$ code, a systematic transfer function matrix might be

$$G_1(x) = \begin{bmatrix} 1 & 0 & \frac{x}{1+x^3} \\ 0 & 1 & \frac{x^2}{1+x^3} \end{bmatrix}, \tag{12.2}$$

with a possible realization as shown in Figure 12.3. This is based on the controller form of Figure 4.7. Another more efficient realization based on the observability form from Figure 4.8, is shown in figure 12.4. In this case, only a single set of memory elements is used, employing linearity. With $m(x) = [1 + x^2 + x^4 + x^5 + x^7 + \cdots, x^2 + x^5 + x^6 + x^7 + \cdots]$, the output is

$$\mathbf{c}(x) = [1 + x^2 + x^4 + x^5 + x^7 + \cdots, x^2 + x^5 + x^6 + x^7 + \cdots, x + x^3 + x^5 + \cdots].$$

The corresponding bit sequences are

$$[\{1, 0, 1, 0, 1, 1, 0, 1, \ldots\}, \{0, 0, 1, 0, 0, 1, 1, 1, \ldots\}, \{0, 1, 0, 1, 0, 1, 0, 0, \ldots, \}\}$$

which, when interleaved, produce the output sequence

$$\{100, 001, 110, 001, 100, 111, 010, 110\}.$$

□

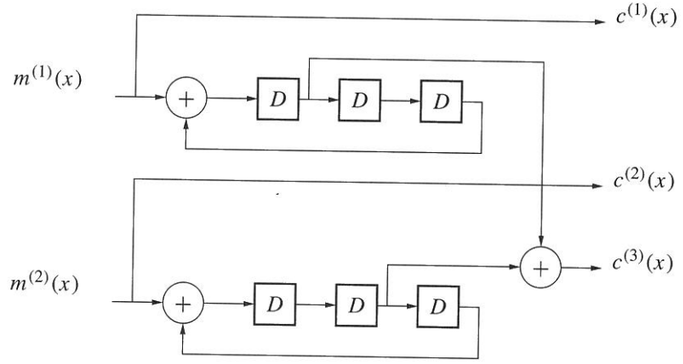


Figure 12.3: A systematic $R = 2/3$ encoder.

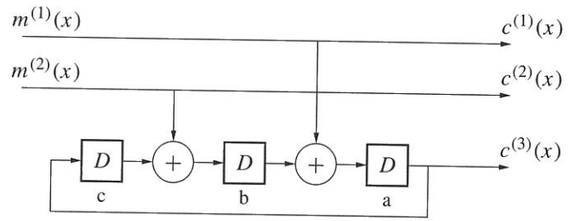


Figure 12.4: A systematic $R = 2/3$ encoder with more efficient hardware.

For feedforward encoders, it is common to indicate the connection polynomials as vectors of numbers representing the *impulse response* of the encoder, rather than polynomials. The transfer function matrix $G(x) = [1 + x^2, 1 + x + x^2]$ is represented by the vectors

$$\mathbf{g}^{(1)} = [101] \quad \text{and} \quad \mathbf{g}^{(2)} = [111].$$

These are often expressed compactly (e.g., in tables of codes) in octal form, where triples of bits are represented using the integers from 0 to 7. In this form, the encoder is represented using $g^{(1)} = 5$, $g^{(2)} = 7$.

For an impulse response $\mathbf{g}^{(j)} = [g_0^{(j)}, g_1^{(j)}, \ldots, g_r^{(j)}]$, the output at time i due to the input sequence m_i is

$$c_i = \sum_{l=0}^{r} m_{i-l} g_l^{(j)},$$

which is, of course, a convolution sum (hence the name of the codes). For an input sequence $\mathbf{m}$, the output sequence can be written as $\mathbf{c}^{(j)} = \mathbf{m} * \mathbf{g}^{(j)}$, where $*$ denotes discrete-time convolution. The operation of convolution can also be represented using matrices. Let $\mathbf{m} = [m_0, m_1, m_2, \ldots]$. Then for $\mathbf{g}^{(j)} = [g_0^{(j)}, g_1^{(j)}, \ldots, g_r^{(j)}]$, the convolution $\mathbf{c} = \mathbf{m}\mathbf{g}^{(j)}$

can be represented as

$$\mathbf{c} = [m_0, m_1, \ldots, m_L] \begin{bmatrix} g_0^{(j)} & g_1^{(j)} & g_2^{(j)} & \cdots & g_L^{(j)} & & \\ & g_0^{(j)} & g_1^{(j)} & \cdots & g_{L-1}^{(j)} & g_L^{(j)} & \\ & & g_0^{(j)} & \cdots & g_{L-2}^{(j)} & g_{L-1}^{(j)} & g_L^{(j)} \\ & & & \ddots & \ddots & \ddots & \ddots \end{bmatrix},$$

where empty entries in the matrix indicate zeros.

For a rate $1/2$ code, the operation of convolution and interleaving the output sequences is represented by the following matrix, where the columns of different matrices are interleaved:

$$G = \begin{bmatrix} g_0^{(1)} g_0^{(2)} & g_1^{(1)} g_1^{(2)} & g_2^{(1)} g_2^{(2)} & \cdots & g_L^{(1)} g_L^{(2)} & & \\ & g_0^{(1)} g_0^{(2)} & g_1^{(1)} g_1^{(2)} & \cdots & g_{L-1}^{(1)} g_{L-1}^{(2)} & g_L^{(1)} g_L^{(2)} & \\ & & g_0^{(1)} g_0^{(2)} & \cdots & g_{L-2}^{(1)} g_{L-2}^{(2)} & g_{L-1}^{(1)} g_{L-1}^{(2)} & g_L^{(1)} g_L^{(2)} \\ & & & \ddots & \ddots & \ddots & \ddots \end{bmatrix}. \tag{12.3}$$

It is the shift (Toeplitz) structure of these generator matrices that gives rise to some of the desirable attributes of convolutional codes.

For a k-input, n-output code with impulse response vectors $\mathbf{g}^{(i,j)}, i = 1, 2, \ldots, k$ and $j = 1, 2, \ldots, n$, where $\mathbf{g}^{(i,j)}$ is the impulse response of the encoder connecting input i with output j, the output can be written as

$$c_t^{(j)} = \sum_{q=1}^{k} \sum_{l=0}^{r} m_{i-l}^{(q)} g_{l-t}^{(q,j)}, \qquad j = 1, 2, \ldots, n.$$

A matrix description of the these codes can also be given, but the transfer function matrix $G(x)$ is usually more convenient.

We always deal with *delay free* transfer function matrices, for which it is not possible to factor out a common multiple x^i from $G(x)$. That is, it is not possible to write $G(x) = x^i \tilde{G}(x)$ for some $i > 0$ and any $\tilde{G}(x)$.

12.1.1 The State

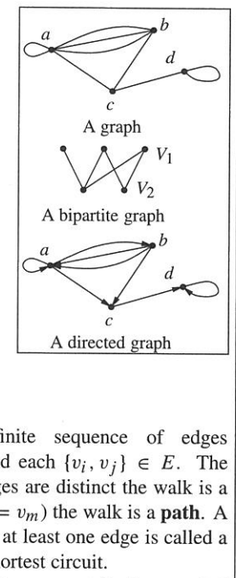
A convolutional encoder is a state machine. For both encoding and decoding purposes, it is frequently helpful to think of the *state diagrams* of the state machines, that is, a representation of the temporal relationships between the states portraying state/next-state relationships as a function of the inputs and the outputs. For an implementation with ν memory elements, there are 2^ν states in the state diagram.

Another representation which is extremely useful is a graph representing the connections from states at one time instant to states at the next time instant. The graph is a bipartite graph, that is, a graph which contains two subsets of nodes, with edges running only between the two subsets. By stacking these bipartite graphs to show several time steps one obtains a graph known as a *trellis*, so-called because of its resemblance to a trellis that might be used for decorative purposes in landscaping.

It may be convenient for some implementations to provide a table indicating the state/next state information explicitly. From the state/next state table the state/previous state information can be extracted.

Box 12.1: Graphs: Basic Definitions

Graph concepts are used throughout the remainder of the book. The following definitions summarize some graph concepts [375]. A **graph** G is a pair (V, E), where V is a nonempty finite set of **vertices** or **nodes** often called the vertex set and E is a finite family of unordered pairs of elements of V called **edges**. (A family is like a set, but some elements may be repeated.) In the graph here, the vertex set is $V = \{a, b, c, d\}$ and the edge family is $E = \{\{a, a\}, \{a, b\}, \{a, b\}, \{a, b\}, \{a, c\}, \{b, c\}, \{c, d\}, \{d, d\}\}$.



A **loop** is an edge joining a vertex to itself. If the edge family is in fact a set (no repeated elements) and there are no loops then the graph is a **simple graph.**

Two vertices of a graph G are **adjacent** if there is an edge joining them. We also say that adjacent nodes are **neighbors**. The two vertices are said to be **incident** to such an edge. Two distinct edges of a graph are adjacent if they have at least one vertex in common. The **degree** of a vertex is the number of edges incident to it.

The vertex set V of a **bipartite** graph can be split into two disjoint sets $V = V_1 \cup V_2$ in such a way that every edge of G joins a vertex of V_1 to a vertex of V_2.

A **walk** through a graph $G = (V, E)$ is a finite sequence of edges $\{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_{m-1}, v_m\}$, where each $v_i \in V$ and each $\{v_i, v_j\} \in E$. The **length** of a walk is the number of edges in it. If all the edges are distinct the walk is a **trail**; if all the vertices are also distinct (except possibly $v_0 = v_m$) the walk is a **path**. A path or trail is **closed** if $v_0 = v_m$. A closed path containing at least one edge is called a **circuit**. The **girth** of a graph is the number of edges in its shortest circuit.

A graph is **connected** if there is a path between any pair of vertices $v, w \in V$. A connected graph which contains no circuits is a **tree**. A node of a tree is a **leaf** if its degree is equal to 1.

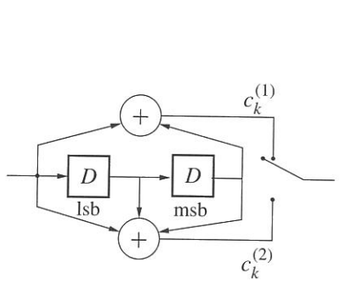
A **directed graph** (digraph) G is a pair (V, E), where E is a finite family of *ordered pairs* of elements of V. Such graphs are frequently represented using arrows to represent edges. In the directed graph here, the edge set is $E = \{(a, a), (a, b), (b, a), (b, a), (a, c), (b, c), (c, d), (d, d)\}$.

Example 12.4 Consider again the convolutional encoder of Example 12.1 with transfer function matrix $G(x) = [1 + x^2, 1 + x + x^2]$. A realization and its corresponding state diagram are shown in Figure 12.5(a) and (b). The state is indicated as a pair of bits, with the first bit representing the least significant bit (lsb). The branches along the state diagram indicate input/output values. One stage of the trellis (corresponding to the transition between two time instants) is shown in Figure 12.5(c). Three trellis stages are shown in Figure 12.5(d). □

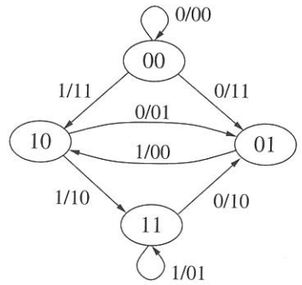
Example 12.5 For the rational systematic encoder with matrix transfer function

$$G(x) = \begin{bmatrix} 1 & 0 & \frac{x}{1+x^3} \\ 0 & 1 & \frac{x^2}{1+x^3} \end{bmatrix}, \tag{12.4}$$

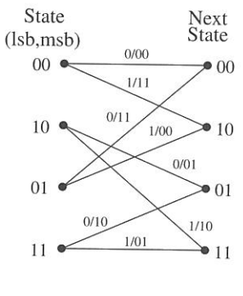
with the circuit realization of Figure 12.4, the state diagram, and trellis are shown in Figure 12.6. In the state diagram, the states are represented as integers from 0 to 7 using the numbers (a, b, c)



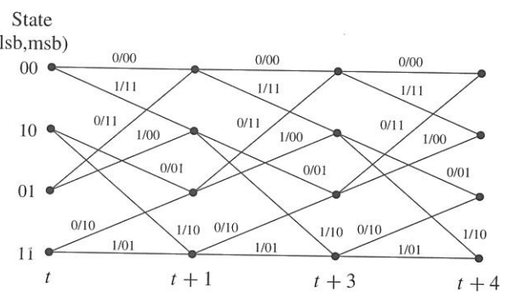
(a) Encoder.



(b) State Diagram.



(c) One stage of trellis.



(d) Three stages of trellis.

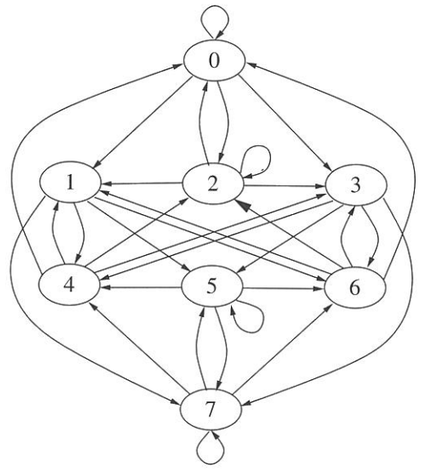
Figure 12.5: Encoder, state diagram, and trellis for $G(x) = [1 + x^2, 1 + x + x^2]$.

corresponding to the registers shown on the circuit. (That is, the lsb is on the right of the diagram this time.) Only the state transitions are shown, not the inputs or outputs, as that would excessively clutter the diagram. The corresponding trellis is also shown, with the branches input/output information listed on the left, with the order of the listing corresponding to the sequence of branches emerging from the corresponding state in top-to-bottom order.

□

12.2 Definition of Codes and Equivalent Codes

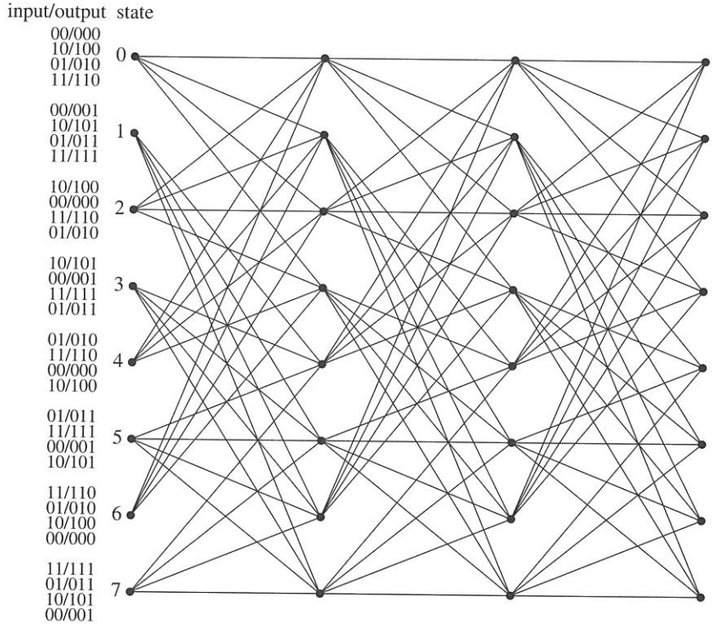
Having now seen several examples of codes, it is now time to formalize the definition and examine some structural properties of the codes. It is no coincidence that the code sequences $(c^{(1)}(x), c^{(2)}(x))$ are the same for Examples 12.1 and 12.2. The sets of sequences that lie in the range of the transfer function matrices $G_a(x)$ and $G_b(x)$ are identical: even though the encoders are different, they encode to the same code. (This is analogous to having different generator matrices to represent the same block code.)



(a) State diagram.

State	Next State			
0	0	1	2	3
1	4	5	6	7
2	1	0	3	2
3	5	4	7	6
4	2	3	0	1
5	6	7	4	5
6	3	2	1	0
7	7	6	5	4

(b) State/Next State Information.



(c) Trellis.

Figure 12.6: State diagram and trellis for a rate $R = 2/3$ systematic encoder.

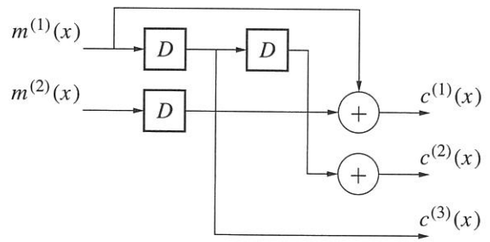


Figure 12.7: A feedforward $R = 2/3$ encoder.

We formally define a convolutional code as follows:

Definition 12.1 [303, p. 94] A rate $R = k/n$ code over the field of rational Laurent series $F[[x]]$ over the field $\mathbb{F}$ is the image of an injective linear mapping of the k-dimensional Laurent series $\mathbf{m}(x) \in \mathbb{F}[[x]]^k$ into the n-dimensional Laurent series $\mathbf{c}(x) \in \mathbb{F}[[x]]^n$. □

In other words, the convolutional code is the set $\{\mathbf{c}(x)\}$ of all possible output sequences as all possible input sequences $\{\mathbf{m}(x)\}$ are applied to the encoder. The code is the image set or (row) range of the linear operator $G(x)$, *not* the linear operator $G(x)$ itself.

Example 12.6 Let

$$G_2(x) = \begin{bmatrix} 1 & x^2 & x \\ x & 1 & 0 \end{bmatrix} \tag{12.5}$$

and note that

$$G_2(x) = \begin{bmatrix} 1 & x^2 \\ x & 1 \end{bmatrix} G_1(x) = T_2(x) G_1(x),$$

(where $G_1(x)$ was defined in (12.2)) and consider the encoding operation

$$m(x)G_2(x) = m(x) \begin{bmatrix} 1 & x^2 \\ x & 1 \end{bmatrix} G_1(x) = m'(x)G_1(x),$$

where

$$m'(x) = m(x) \begin{bmatrix} 1 & x^2 \\ x & 1 \end{bmatrix} = m(x)T_2(x).$$

Corresponding to each $m(x)$ there is a unique $m'(x)$, since $T_2(x)$ is invertible. Hence, as $m'(x)$ varies over all possible input sequences, $m(x)$ also varies over all possible input sequences. The set of output sequences $\{c(x)\}$ produced is the same for $G_2(x)$ as $G_1(x)$: that is, both encoders produce the same code.

Figure 12.7 shows a schematic representation of this encoder. Note that the implementation of both $G_1(x)$ (of Figure 12.4) and $G_2(x)$ have three one-bit memory elements in them. The contents of these memory elements may be thought of as the *state* of the devices. Since these are binary circuits, there are $2^3 = 8$ distinct states in either implementation. □

Example 12.7 Another encoder for the code of Example 12.1 is

$$G_3(x) = \begin{bmatrix} 1+x & 1+x^2 & x \\ x+x^2 & 1+x & 0 \end{bmatrix}, \tag{12.6}$$

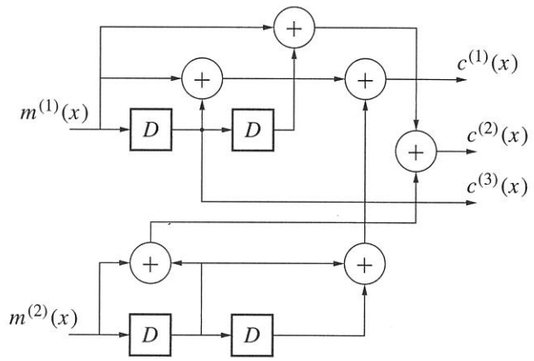


Figure 12.8: A less efficient feedforward $R = 2/3$ encoder.

where, it may be observed,

$$G_3(x) = \begin{bmatrix} 1+x & 1+x^2 \\ x+x^2 & 1+x \end{bmatrix} G_1(x) = T_3(x)G_1(x).$$

The encoding operation

$$m(x)G_3(x) = m(x)T_3(x)G_1(x) = m'(x)G_1(x),$$

with $m'(x) = m(x)T_3(x)$, again results in the same code, since $T_3(x)$ is invertible.

The schematic for $G_3(x)$ in Figure 12.8 would require more storage blocks than either $G_1(x)$ or $G_2(x)$: it is not as efficient in terms of hardware. □

These examples motivate the following definition.

Definition 12.2 Two transfer function matrices $G(x)$ and $G'(x)$ are said to be **equivalent** if they generate the same convolutional code. Two transfer function matrices $G(x)$ and $G'(x)$ are equivalent if $G(x) = T(x)G'(x)$ for an *invertible* matrix $T(x)$. □

These examples also motivate other considerations: For a given a code, is there always a feedforward transfer matrix representation? Is there always a systematic representation? What it the "minimal" representation, requiring the least amount of memory? As the following section reveals, another question is whether the representation is catastrophic.

12.2.1 Catastrophic Encoders

Besides the hardware inefficiency, there is another fundamental problem with the encoder $G_3(x)$ of (12.6). Suppose that the input is

$$\mathbf{m}(x) = \begin{bmatrix} 0 & \frac{1}{1+x} \end{bmatrix},$$

where, expanding the formal series by long division,

$$\frac{1}{1+x} = 1 + x + x^2 + x^3 + \cdots$$

The input sequence thus has infinite Hamming weight. The corresponding output sequence is

$$\mathbf{c}(x) = m(x)G_3(x) = \begin{bmatrix} x & 1 & 0 \end{bmatrix},$$

a sequence with total Hamming weight 2. Suppose now that $\mathbf{c}(x)$ is passed through a channel and that *two* errors occur at precisely the locations of the nonzero code elements. Then the received sequences is exactly zero, which would decode (under any reasonable decoding scheme) to $\hat{\mathbf{m}}(x) = \begin{bmatrix} 0 & 0 \end{bmatrix}$. Thus, a *finite* number of errors in the channel result in an *infinite* number of decoder errors. Such an encoder is called a *catastrophic* encoder. It may be emphasized, however, that the problem is not with the code but the particular encoder, since $G_1(x)$, $G_2(x)$ and $G_3(x)$ all produce the same code but, $G_1(x)$ and $G_2(x)$ do not exhibit catastrophic behavior.

Letting $\mathrm{wt}(c(x))$ denote the weight of the sequence $c(x)$, we have the following definition:

Definition 12.3 [303, p.97] An encoder $G(x)$ for a convolutional code is **catastrophic** if there exists a message sequence $\mathbf{m}(x)$ such that $\mathrm{wt}(\mathbf{m}(x)) = \infty$ and the weight of the coded sequence $\mathrm{wt}(\mathbf{m}(x)G(x)) < \infty$. □

To understand more of the nature of catastrophic codes, we introduce the idea of a right inverse of a matrix.

Definition 12.4 Let $k < n$. A **right inverse** of a $k \times n$ matrix G is a $n \times k$ matrix G^{-1} such that $GG^{-1} = I_{k,k}$, the $k \times k$ identity matrix. (This is not the same as the inverse, which cannot exist when G is not square.) A right inverse of G can exist only if G is full rank. □

Example 12.8 For $G_1(x)$ of (12.2), a right inverse is

$$G_1(x)^{-1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}.$$

For $G_2(x)$ of (12.5), a right inverse is

$$G_2(x)^{-1} = \begin{bmatrix} 1 & x \\ x & 1+x^2 \\ x^2 & 1+x+x^3 \end{bmatrix}.$$

For $G_3(x)$ of (12.6), a right inverse is

$$G_3(x)^{-1} = \frac{1}{1+x+x^3+x^4} \begin{bmatrix} 1+x & 1+x^2 \\ x+x^2 & 1+x \\ 0 & 0 \end{bmatrix}.$$

□

Note that $G_1(x)^{-1}$ and $G_2(x)^{-1}$ are *polynomial* matrices — they have only polynomial entries — while $G_3(x)^{-1}$ has non-polynomial entries — some of its entries involve rational functions.

Example 12.9 It should be observed that right inverses are not necessarily unique. For example, the matrix $\begin{bmatrix} 1+x^2, 1+x+x^2 \end{bmatrix}$ has the right inverses

$$\begin{bmatrix} 1+x & x \end{bmatrix}^T \quad \text{and} \quad \begin{bmatrix} \frac{x}{x+1} & 1 \end{bmatrix}^T.$$

Of these, one has all polynomial elements. □

Definition 12.5 A transfer function matrix with only polynomial entries is said to be a **polynomial encoder** (i.e., it uses FIR filters). More briefly, such an encoder is said to be **polynomial**. A transfer function matrix with rational entries is said to be a **rational encoder** (i.e., it uses IIR filters), or simply **rational**. □

For an encoder $G(x)$ with right inverse $G(x)^{-1}$, the message may be recovered (in a theoretical sense when there is no noise corrupting the code — this is not a decoding algorithm!) by

$$\mathbf{c}(x)G(x)^{-1} = \mathbf{m}(x)G(x)G(x)^{-1} = \mathbf{m}(x). \tag{12.7}$$

Now suppose that $\mathbf{c}(x)$ has finite weight, but $\mathbf{m}(x)$ has infinite weight: from (12.7) this can only happen if one or more elements of the right inverse $G(x)^{-1}$ has an infinite number of coefficients, that is, they are rational functions. It turns out that this is a necessary and sufficient condition:

Theorem 12.1 *A transfer function matrix $G(x)$ is not catastrophic if and only if it has a right inverse $G(x)^{-1}$ having only polynomial entries.*

From the right inverses in Example 12.8, we see that $G_1(x)$ and $G_2(x)$ have polynomial right inverses, while $G_3(x)$ has non-polynomial entries, indicating that $G_3(x)$ is a catastrophic generator.

Definition 12.6 A transfer function matrix $G(x)$ is **basic** if it is polynomial and has a polynomial right inverse. □

$G_2(x)$ is an example of a basic transfer function matrix.

Example 12.10 Another example of a transfer function matrix for the code is

$$G_4(x) = \begin{bmatrix} 1+x+x^2+x^3 & 1+x & x \\ x & 1 & 0 \end{bmatrix}. \tag{12.8}$$

The invariant factor decomposition (presented below) can be used to show that this is basic. However, for sufficiently small matrices finding a right inverse may be done by hand. We seek a polynomial matrix such that

$$\begin{bmatrix} 1+x+x^2+x^3 & 1+x & x \\ x & 1 & 0 \end{bmatrix} \begin{bmatrix} a & d \\ b & e \\ c & f \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Writing out the implied equations we have

$$a(1+x+x^2+x^3) + b(1+x) + cx = 1 \qquad ax + b = 0$$

$$d(1+x+x^2+x^3) + e(1+x) + fx = 0 \qquad dx + e = 1.$$

From the second we obtain $b = ax$; substituting this into the first we find $a(1+x^3) + cx = 1$. By setting $c = x^2$ and $a = 1$ we can solve this using polynomials.

From the fourth equation we obtain $e = 1 + dx$, so that from the third equation

$$d(1+x+x^2+x^3) + (1+dx)(1+x) + fx = 0$$

or $d(1+x^3) + fx = 1+x$. This yields $d = 1$ and $f = x^2+1$. This gives a polynomial right inverse, so $G_4(x)$ is basic. Note that the encoder requires four memory elements in its implementation. □

Two *basic* encoders $G(x)$ and $G'(x)$ are equivalent if and only if $G(x) = T(x)G'(x)$, where

1. $T(x)$ is not only invertible (as required by mere equivalence),
2. But also $\det(T(x)) = 1$,

so that when the right inverse is computed all the elements remain polynomial.

12.2.2 Polynomial and Rational Encoders

We show in this section that every rational encoder has an equivalent basic encoder. The implication is that it is sufficient to use only feedforward (polynomial) encoders to represent every code. There is, however, an important caveat: there may not be an equivalent basic (or even polynomial) *systematic* encoder. Thus, if a systematic coder is desired, it may be necessary to use a rational encoder. This is relevant because the very powerful behavior of turbo codes relies on good *systematic* convolutional codes.

Our results make use of the **invariant factor decomposition** of a matrix [162, section 3.7]. Let $G(x)$ be a $k \times n$ polynomial matrix. Then[2] $G(x)$ can be written as

$$G(x) = A(x)\Gamma(x)B(x),$$

where $A(x)$ is a $k \times k$ polynomial matrix and $B(x)$ is a $n \times n$ polynomial matrix and where $\det(A(x)) = 1$, $\det(B(x)) = 1$ (i.e., they are **unimodular matrices**); and $\Gamma(x)$ is the $k \times n$ diagonal matrix

$$\Gamma(x) = \begin{bmatrix} \gamma_1(x) & & & & \\ & \gamma_2(x) & & & \\ & & \gamma_3(x) & & \mathbf{0}_{k,n-k} \\ & & & \ddots & \\ & & & & \gamma_k(x) \end{bmatrix}.$$

The nonzero elements $\gamma_i(x)$ of $\Gamma(x)$ are polynomials and are called the **invariant factors** of $G(x)$. (If any of the $\gamma_i(x)$ are zero, they are included in the zero block, so k is the number of nonzero elements.) Furthermore, the invariant factors satisfy

$$\gamma_i(x) \,|\, \gamma_{i+1}(x).$$

(Since we are expressing a theoretical result here, we won't pursue the algorithm for actually computing the invariant factor decomposition[3]); it is detailed in [162].

Extending the invariant factor theorem to rational matrices, a rational matrix $G(x)$ can be written as

$$G(x) = A(x)\Gamma(x)B(x),$$

where $A(x)$ and $B(x)$ are again polynomial unimodular matrices and $\Gamma(x)$ is diagonal with rational entries $\alpha_i(x)/\beta_i(x)$, such that $\gamma_i(x) = \alpha_i(x) \,|\, \alpha_{i+1}(x)$ and $\beta_{i+1}(x) \,|\, \beta_i(x)$.

Let $G(x)$ be a rational encoding matrix, with invariant factor decomposition $G(x) = A(x)\Gamma(x)B(x)$. Let us decompose $B(x)$ into the blocks

$$B(x) = \begin{bmatrix} G'(x) \\ B_2(x) \end{bmatrix},$$

[2] The invariant factor decomposition has a technical requirement: The factorization in the ring must be unique, up to ordering and units. This technical requirement *is* met in our case, since the polynomials form a principal ideal domain, which implies unique factorization. See, e.g., [106, Chapter 32].

[3] The invariant factor decomposition can be thought of as a sort of singular value decomposition for modules.

where $G'(x)$ is $k \times n$. Then, since the last k columns of $\Gamma(x)$ are zero, we can write

$$G(x) = A(x) \begin{bmatrix} \frac{\alpha_1(x)}{\beta_1(x)} & & & \\ & \frac{\alpha_2(x)}{\beta_2(x)} & & \\ & & \ddots & \\ & & & \frac{\alpha_k(x)}{\beta_k(x)} \end{bmatrix} G'(x) \triangleq A(x)\Gamma'(x)G'(x).$$

Since $A(x)$ and $\Gamma'(x)$ are nonsingular matrices, $G(x)$ and $G'(x)$ are equivalent encoders: they describe the same convolutional code. But $G'(x)$ is polynomial (since $B(x)$ is polynomial) and since $B(x)$ is unimodular (and thus has a polynomial inverse) it follows that $G'(x)$ has a polynomial right inverse. We have thus proved the following:

Theorem 12.2 *Every rational encoder has an equivalent basic transfer function matrix.*

The proof of the theorem is constructive: To obtain a basic encoding matrix from a rational transfer function $G(x)$, compute the invariant factor decomposition $G(x) = A(x)\Gamma(x)B(x)$ and take the first k rows of $B(x)$.

12.2.3 Constraint Length and Minimal Encoders

Comparing the encoders for the code we have been examining, we have seen that the encoders for $G_1(x)$ or $G_2(x)$ use three memory elements, while the encoder $G_3(x)$ uses four memory elements. We investigate in this section aspects of the question of the smallest amount of memory that a code requires of its encoder.

Let $G(x)$ be a basic encoder (so that the elements of $G(x)$ are polynomials). Let

$$\nu_i = \max_j \deg(g_{ij}(x))$$

denote the maximum degree of the polynomials in row i of $G(x)$. This is the number of memory elements necessary to store the portion of a realization (circuit) of the encoder corresponding to input i. The number

$$\nu = \sum_{i=1}^{k} \nu_i \tag{12.9}$$

represents the total amount of storage required for all inputs. This quantity is called the **constraint length** of the encoder.

Note: In other sources (e.g., [373]), the constraint length is defined as the maximum number of bits in a single output stream that can be affected by any input bit (for a polynomial encoder). This is taken as the highest degree of the encoder plus one: $\nu = 1 + \max_{i,j} \deg(g_{i,j}(x))$. The reader should be aware that different definitions are used. Ours suits the current purposes.

We make the following definition:

Definition 12.7 A **minimal basic** encoder is a basic encoder that has the smallest constraint length among all equivalent basic encoders. □

Typically we are interested in minimal encoders: they require the least amount of hardware to build and they have the fewest evident states. We now explore the question of when an encoder is minimal basic.

The first theorem involves a particular decomposition of the decoder matrix. We demonstrate first with some examples. Let $G(x) = G_2(x)$ from (12.2). Write

$$G_2(x) = \begin{bmatrix} 1 & x^2 & x \\ x & 1 & 0 \end{bmatrix} = \begin{bmatrix} x^2 & & \\ & x & \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 & x \\ 0 & 1 & 0 \end{bmatrix}. \tag{12.10}$$

As another example, when $G(x) = G_4(x)$ from (12.8)

$$\begin{aligned} G_4(x) &= \begin{bmatrix} 1+x+x^2+x^3 & 1+x & x \\ x & 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} x^3 & \\ & x \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 1+x+x^2 & 1+x & x \\ 0 & 1 & 0 \end{bmatrix}. \end{aligned} \tag{12.11}$$

In general, given a basic encoder $G(x)$ we write it as

$$G(x) = \begin{bmatrix} x^{\nu_1} & & & \\ & x^{\nu_2} & & \\ & & \cdots & \\ & & & x^{\nu_k} \end{bmatrix} G_h + \tilde{G}(x) = \Lambda(x) G_h + \tilde{G}(x), \tag{12.12}$$

where G_h is a binary matrix with a 1 indicating the position where the highest degree term x^{ν_i} occurs in row i and each row of $\tilde{G}(x)$ contains all the terms of degree less than ν_i. Using this notation, we have the following:

Theorem 12.3 *[175] Let $G(x)$ be a $k \times n$ basic encoding matrix with constraint length ν. The following statements are equivalent:*

(a) $G(x)$ is a minimal basic encoding matrix.

(b) The maximum degree μ among the $k \times k$ subdeterminants of $G(x)$ is equal to the overall constraint length ν.

(c) G_h is full rank.

To illustrate this theorem, consider the decomposition of $G_4(x)$ in (12.11). The 2×2 subdeterminants of $G_4(x)$ are obtained by taking the determinant of the two 2×2 submatrices of $G_4(x)$,

$$\det \begin{bmatrix} 1+x+x^2+x^3 & 1+x \\ x & 1 \end{bmatrix} \qquad \det \begin{bmatrix} 1+x & x \\ 1 & 0 \end{bmatrix},$$

the maximum degree of which is 3. Also, we note that G_h is not full rank. Hence, we conclude that $G_4(x)$ is not a minimal basic encoding matrix.

Proof To show the equivalence of (b) and (c): Observe that the degree of a subdeterminant of $G(x)$ is determined by the $k \times k$ submatrices of $\Lambda(x)G_h$ (which have the largest degree terms) and not by $\tilde{G}(x)$. The degree of the determinants of the $k \times k$ submatrices of $G(x)$ are then determined by the subdeterminants of $\Lambda(x)$ and the $k \times k$ submatrices of G_h. Since $\det \Lambda(x) \neq 0$, if G_h is full rank, then at least one of its $k \times k$ submatrices has nonzero determinant, so that at least one of the determinants of the $k \times k$ submatrices of $\Lambda(x)G_h$ has degree μ equal to $\deg \Lambda(x) = \nu$. On the other hand, if G_h is rank deficient, then none of the determinants of the submatrices of $\Lambda(x)G_h$ can be equal to the determinant of $\Lambda(x)$.

To show that (a) implies (b): Assume that $G(x)$ is minimal basic. Suppose that $\text{rank}(G_h) < k$. Let the rows of $G(x)$ be denoted by $\mathbf{g}_1, \mathbf{g}_2, \ldots, \mathbf{g}_k$, let the rows of G_h be denoted by $\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_k$, and let the rows of $\tilde{G}(x)$ be denoted by $\tilde{\mathbf{g}}_1, \tilde{\mathbf{g}}_2, \ldots, \tilde{\mathbf{g}}_k$. Then the decomposition (12.12) is

$$\mathbf{g}_i = x^{\nu_i} \mathbf{h}_i + \tilde{\mathbf{g}}_i.$$

By the rank-deficiency there is a linear combination of rows of G_h such that

$$\mathbf{h}_{i_1} + \mathbf{h}_{i_2} + \cdots + \mathbf{h}_{i_d} = 0$$

for some $d \leq k$. Assume (without loss of generality) that the rows of $G(x)$ are ordered such that $\nu_1 \geq \nu_2 \geq \cdots \geq \nu_k$. The ith row of $\Lambda(x)G_h$ is $x^{\nu_i}\mathbf{h}_i$. Adding

$$x^{\nu_{i_1}} [\mathbf{h}_{i_2} + \mathbf{h}_{i_3} + \cdots + \mathbf{h}_{i_d}]$$

to the i_1st row of $\Lambda(x)G_h$ (which is $x^{\nu_i}\mathbf{h}_i$) reduces it to an all-zero row. Note that

$$x^{\nu_{i_1}} [\mathbf{h}_{i_2} + \mathbf{h}_{i_3} + \cdots + \mathbf{h}_{i_d}] = x^{\nu_{i_1} - \nu_{i_2}} x^{\nu_{i_2}} \mathbf{h}_{i_2} + x^{\nu_{i_1} - \nu_{i_3}} x^{\nu_{i_3}} \mathbf{h}_{i_3} + \cdots + x^{\nu_{i_1} - \nu_{i_d}} x^{\nu_{i_d}} \mathbf{h}_{i_d}.$$

Now consider computing $G'(x) = T(x)G(x)$, where $T(x)$ is the invertible matrix

$$T = \begin{array}{c} \\ i_1: \\ \\ \\ \\ \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{cccccccc} & & & & i_2 & & i_3 & \cdots & i_d \end{array} \\ \begin{bmatrix} 1 & & & & & & & & \\ & 1 & & & & & & & \\ & & \ddots & & & & & & \\ & & & 1 & \cdots & x^{\nu_{i_1} - \nu_{i_2}} & & & \\ & & & & & \ddots & & & \\ & & & & & & 1 & & \\ & & & & & & & \ddots & \\ & & & & & & & & 1 \end{bmatrix} \end{array},$$

with an identity on the diagonal. This has the effect of adding

$$x^{\nu_{i_1} - \nu_{i_2}} \mathbf{g}_{i_2} + x^{\nu_{i_1} - \nu_{i_3}} \mathbf{g}_{i_3} + \cdots + x^{\nu_{i_1} - \nu_{i_d}} \mathbf{g}_{i_d}$$

to the i_1st row of $G(x)$, which reduces the highest degree of the i_1st row of $G(x)$ (because the term $x^{\nu_{i_1}}\mathbf{h}_{i_1}$ is eliminated) but leaves other rows of $G(x)$ unchanged. But $G'(x)$ is an equivalent transfer function matrix. We thus obtain a basic encoding matrix $G'(x)$ equivalent to $G(x)$ with an overall constraint length less than that of $G(x)$. This contradicts the assumption that $G(x)$ is minimal basic, which implies that G_h must be full rank. From the equivalence of (b) and (c), $\mu = \nu$.

Conversely, to show (b) implies (a): Let $G'(x)$ be a basic encoding matrix equivalent to $G(x)$. Then $G'(x) = T(x)G(x)$, where $T(x)$ is a $k \times k$ polynomial matrix with $\det T(x) = 1$. The maximum degree among the $k \times k$ subdeterminants of $G'(x)$ is equal to that of $G(x)$ (since $\det T(x) = 1$). Hence, $\text{rank}(G_h)$ is invariant over all equivalent basic encoding matrices. Since $\text{rank}(G_h)$ is less than or equal to the overall constraint length, if $\mu = \nu$, it follows that $G(x)$ is minimal basic. □

The proof is essentially constructive: given a non-minimal basic $G(x)$, a minimal basic encoder can be constructed by finding rows of G_h such that

$$\mathbf{h}_{i_1} + \mathbf{h}_{i_2} + \mathbf{h}_{i_3} + \cdots + \mathbf{h}_{i_d} = 0,$$

where the indices are ordered such that $\nu_{i_d} \geq \nu_{i_j}$, $1 \leq j < d$, then adding

$$x^{\nu_{i_d} - \nu_{i_1}} \mathbf{g}_{i_1} + \cdots x^{\nu_{i_d} - \nu_{i_{d-1}}} \mathbf{g}_{i_{d-1}} \tag{12.13}$$

to the i_dth row of $G(x)$.

Example 12.11 Let $G(x) = G_4(x)$, as before. Then

$$\mathbf{h}_1 = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \quad \mathbf{h}_2 = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$

so that $\mathbf{h}_1 + \mathbf{h}_2 = 0$. We have $i_1 = 2$ and $i_2 = 1$. We thus add

$$x^{3-1}\mathbf{g}_2 = x^2 \begin{bmatrix} x & 1 & 0 \end{bmatrix} = \begin{bmatrix} x^3 & x^2 & 0 \end{bmatrix}$$

to row 1 of $G(x)$ to obtain the transfer function matrix

$$G_5(x) = \begin{bmatrix} 1+x+x^2 & 1+x+x^2 & x \\ x & 1 & 0 \end{bmatrix}$$

to obtain an equivalent minimal basic encoder. □

Comparing $G_5(x)$ with $G_2(x)$, we make the observation that minimal basic encoders are not unique.

As implied by its name, the advantage of a basic minimal encoder is that it is "smallest" in some sense. It may be built in such a way that the number of memory elements in the device is the smallest possible and the number of states of the device is the smallest possible. There is another advantage to minimal encoders: it can be shown that a minimal encoder is not catastrophic.

12.2.4 Systematic Encoders

Given an encoder $G(x)$, it may be turned into a systematic decoder by identifying a full-rank $k \times k$ submatrix $T(x)$. Then form

$$G'(x) = T(x)^{-1}G(x).$$

Then $G'(x)$ is of the form (perhaps after column permutations)

$$G'(x) = \begin{bmatrix} I_{k,k} & P_{k,n-k}(x) \end{bmatrix},$$

where $P_{k,n-k}(x)$ is a (generally) rational matrix. The outputs produced by $P_{k,n-k}$ — that is, the non-systematic part of the generator — are frequently referred to as the parity bits, or check bits, of the coded sequence.

Example 12.12 [175] Suppose

$$G(x) = \begin{bmatrix} 1+x & x & 1 \\ x^2 & 1 & 1+x+x^2 \end{bmatrix}$$

and $T(x)$ is taken as the first two columns:

$$T(x) = \begin{bmatrix} 1+x & x \\ x^2 & 1 \end{bmatrix} \qquad T^{-1}(x) = \frac{1}{1+x+x^3}\begin{bmatrix} 1 & x \\ x^2 & 1+x \end{bmatrix}.$$

Then

$$G'(x) = T^{-1}(x)G(x) = \begin{bmatrix} 1 & 0 & \frac{1+x+x^2+x^3}{1+x+x^3} \\ 0 & 1 & \frac{1+x^2+x^3}{1+x+x^3} \end{bmatrix}.$$

□

Historically, polynomial encoders (i.e., those implemented using FIR filters) have been much more commonly used than systematic encoders (employing IIR filters). However, there are some advantages to using systematic codes. First, it can be shown that every systematic encoding matrix is minimal. Second, systematic codes cannot be catastrophic (since the data appears explicitly in the codeword).

For a given constraint length, the set of systematic codes with polynomial transfer matrices has generally inferior distance properties compared with the set of systematic codes with rational transfer matrices. In fact, it has been observed [357, p. 252] that for large constraint lengths ν, the performance of a polynomial systematic code of constraint length K is approximately the same as that of a nonsystematic code of constraint length $K(1-k/n)$. (See Table 12.2) For example, for a rate $R = 1/2$ code, polynomial systematic codes have about the performance of nonsystematic codes of half the constraint length, while requiring exactly the same optimal decoder complexity. Because of these reasons, recent work in turbo codes has relied almost exclusively on systematic encoders.

12.3 Decoding Convolutional Codes

12.3.1 Introduction and Notation

Several algorithms have been developed for decoding convolutional codes. The one most commonly used is the Viterbi algorithm, which is a maximum likelihood sequence estimator (MLSE). A variation on the Viterbi algorithm, known as the soft-output Viterbi algorithm (SOVA), which provides not only decoded symbols but also an indication of the *reliability* of the decoded values, is presented in Section 14.3.17 in conjunction with turbo codes. Another decoding algorithm is the maximum a posteriori (MAP) decoder frequently referred to as the BCJR algorithm, which computes probabilities of decoded bits. The BCJR algorithm is somewhat more complex than the Viterbi algorithm, without significant performance gains compared to Viterbi codes. It is, however, ideally suited for decoding turbo codes, and so is also detailed in chapter 14. It is also shown there that the BCJR and the Viterbi are fundamentally equivalent at a deeper level.

Suboptimal decoding algorithms are also occasionally of interest, particularly when the constraint length is large. These provide most of the performance of the Viterbi algorithm, but typically have substantially lower computational complexity. In Section 12.8 the stack algorithm (also known as the ZJ algorithm), Fano's algorithm, and the M-algorithm are presented as instances of suboptimal decoders.

To set the stage for the decoding algorithm, we introduce some notation for the stages of processing. Consider the block diagram in Figure 12.9. The time index is denoted by t, which indexes the times at which states are distinguished in the state diagram; there are thus k bits input to the encoder and n bits output from the encoder at each time step t.

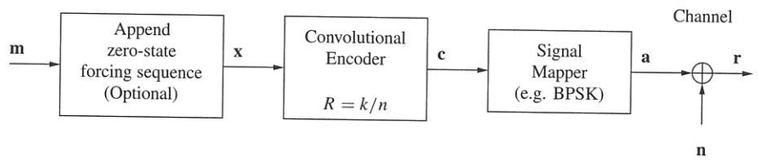


Figure 12.9: Processing stages for a convolutional code.

- The input — message — data may have a sequence appended to drive the state to 0 at the end of some block of input. At time t there are k input bits, denoted as $m_t^{(i)}$ or $x_t^{(i)}$, $i = 1, 2, \ldots, k$. The set of k input bits at time t is denoted as $\mathbf{m}_t = (m_t^{(1)}, m_t^{(2)}, \ldots, m_t^{(k)})$ and those with the (optional) appended sequence are $\mathbf{x}_t$. An input sequence consisting of L blocks is denoted as $\mathbf{x}$:

$$\mathbf{x} = \{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{L-1}\}.$$

- The corresponding coded output bits are denoted as $c_t^{(i)}$, $i = 1, 2, \ldots, n$, or collectively at time t as $\mathbf{c}_t$. The entire coded output sequence is $\mathbf{c} = \{\mathbf{c}_0, \mathbf{c}_1, \ldots, \mathbf{c}_{L-1}\}$.
- The coded output sequence is mapped to a sequence of M symbols selected from a signal constellation with Q points in some signal space, with $Q = 2^p$. We must have 2^{nL} (the number of coded bits in the sequence) equal to 2^{pM}, so that $M = nL/p$. For convenience in notation, we assume that $p = 1$ (e.g., BPSK modulation), so that $M = L$; we use M as identical to L in this development, although it does not have to be.

  The mapped signals at time t are denoted as $a_t^{(i)}$, $i = 1, 2, \ldots, n$. The entire coded sequence is $\mathbf{a} = \{\mathbf{a}_0, \mathbf{a}_1, \ldots, \mathbf{a}_{L-1}\}$
- The symbols $\mathbf{a}_t$ pass through a channel, resulting in a received symbol $r_t^{(i)}$, $i = 1, 2, \ldots, n$, or a block $\mathbf{r}_t$. We consider explicitly two channel models: an AWGN and a BSC. For the AWGN we have

$$r_t^{(i)} = a_t^{(i)} + n_t^{(i)}, \qquad \text{where } n_t^{(i)} \sim \mathcal{N}(0, \sigma^2), \text{ and where } \sigma^2 = \frac{N_0}{2}.$$

  For the AWGN the received data are real- or complex-valued. For the BSC, the mapped signals are equal to the coded data, $\mathbf{a}_t = \mathbf{c}_t$. The received signal is

$$r_t^{(i)} = c_t^{(i)} \oplus n_t^{(i)}, \qquad \text{where } n_t \sim \mathcal{B}(p_c),$$

  where $\oplus$ denotes addition modulo 2 and p_c is the channel crossover probability and $\mathcal{B}(p_c)$ indicates a Bernoulli random variable. For both channels it is assumed that the $n_t^{(i)}$ are mutually independent for all i and t, resulting in a memoryless channel. We denote the likelihood function for these channels as $f(\mathbf{r}_t|\mathbf{a}_t)$. For the AWGN channel,

$$\begin{aligned} f(\mathbf{r}_t|\mathbf{a}_t) &= \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{1}{2\sigma^2}(r_t^{(i)} - a_t^{(i)})^2\right] \\ &= (\sqrt{2\pi}\sigma)^{-n} \exp\left[-\frac{1}{2\sigma^2}\|\mathbf{r}_t - \mathbf{a}_t\|^2\right], \end{aligned}$$

  where $\|\cdot\|^2$ denotes the usual Euclidean distance,

$$\|\mathbf{r}_t - \mathbf{a}_t\|^2 = \sum_{i=1}^{n} (r_t^{(i)} - a_t^{(i)})^2.$$

  For the BSC,

$$\begin{aligned} f(\mathbf{r}_t|\mathbf{a}_t) &= \prod_{i=1}^{n} p_c^{[r_t^{(i)} \neq a_t^{(i)}]} (1 - p_c)^{[r_t^{(i)} = a_t^{(i)}]} = p_c^{d_H(\mathbf{r}_t, \mathbf{a}_t)} (1 - p_c)^{n - d_H(\mathbf{r}_t, \mathbf{a}_t)} \\ &= \left(\frac{p_c}{1 - p_c}\right)^{d_H(\mathbf{r}_t, \mathbf{a}_t)} (1 - p_c)^n, \end{aligned}$$

  where $[r_t^{(i)} \neq a_t^{(i)}]$ returns 1 if the indicated condition is true and $d_H(\mathbf{r}_t, \mathbf{a}_t)$ is the Hamming distance.

  Since the sequence of inputs uniquely determines the sequence of outputs, mapped outputs, and states, the likelihood function can be equivalently expressed as $f(\mathbf{r}|\mathbf{c})$ or $f(\mathbf{r}|\mathbf{a})$ or $f(\mathbf{r}|\{\Psi_0, \Psi_1, \ldots, \Psi_L\})$.
- Maximizing the likelihood is obviously equivalent to minimizing the negative log likelihood. We deal with negative log likelihood functions and throw away terms and/or factors that do not depend upon the conditioning values. For the Gaussian channel, since

$$-\log f(\mathbf{r}_t|\mathbf{a}_t) = +n \log \sqrt{2\pi}\sigma + \frac{1}{2\sigma^2}\|\mathbf{r}_t - \mathbf{a}_t\|^2$$

  we use

$$\|\mathbf{r}_t - \mathbf{a}_t\|^2 \tag{12.14}$$

  as the "negative log likelihood" function. For the BSC, since

$$-\log f(\mathbf{r}_t|\mathbf{a}_t) = -d_H(\mathbf{r}_t, \mathbf{a}_t) \log \frac{p_c}{1 - p_c} - n \log(1 - p_c)$$

  we use

$$d_H(\mathbf{r}_t, \mathbf{a}_t) \tag{12.15}$$

  as the "negative log likelihood" (since $\log(p_c/(1 - p_c)) < 0$).

  More generally, the affine transformation

$$a[-\log f(\mathbf{r}_t|\mathbf{a}) - b] \tag{12.16}$$

  provides a function equivalent for purposes of detection to the log likelihood function for any $a > 0$ and any b. The parameters a and b can be chosen to simplify computations.
- The state at time t in the trellis of the encoder is denoted as Ψ_t. States are represented with integer values in the range $0 \leq \Psi_t < 2^\nu$, where ν is the constraint length for the encoder. (We use 2^ν since we are assuming binary encoders for convenience. For a q-ary code, the number of states is q^ν.) It is always assumed that the initial state is $\Psi_0 = 0$.
- As suggested by Figure 12.10, quantities associated with the transition from state p to state q are denoted with $^{(p,q)}$. For example, the input which causes the transition from state $\Psi_t = p$ to the state $\Psi_{t+1} = q$ is denoted as $\mathbf{x}^{(p,q)}$. (If the trellis had different structure at different times, one might use the notation $\mathbf{x}_t^{(p,q)}$.) The code bits output sequence as a result of this state transition is $\mathbf{c}^{(p,q)}$ and the mapped symbols are $\mathbf{a}^{(p,q)}$.
- A sequence of symbols such as $\{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_l\}$ is denoted as $\mathbf{x}_0^l$.

12.3.2 The Viterbi Algorithm

MLSD

The Viterbi algorithm was originally proposed by Andrew Viterbi [358], but its optimality as a maximum likelihood sequence decoder was not originally appreciated. In [89] it was established that the Viterbi algorithm computes the maximum likelihood code sequence given the received data. The Viterbi algorithm is essentially a shortest path algorithm, roughly

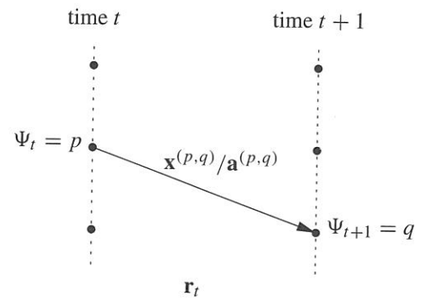


Figure 12.10: Notation associated with a state transition.

analogous to Dijkstra's shortest path algorithm (see, e.g., [305, p. 415]) for computing the shortest path through the trellis associated with the code. The Viterbi algorithm has been applied in a variety of other communications problems, including maximum likelihood sequence estimation in the presence of intersymbol interference [96] and optimal reception of spread-spectrum multiple access communication (see, e.g., [354]). It also appears in many other problems where a "state" can be defined, such as in hidden Markov modeling (see, e.g., [67]). See also [246] for a survey of applications. The decoder takes the input sequence $\mathbf{r} = \{\mathbf{r}_0, \mathbf{r}_1, \ldots\}$ and determines an estimate of the transmitted data $\{\mathbf{a}_0, \mathbf{a}_1, \ldots\}$ and from that an estimate of the sequence of input data $\{\mathbf{x}_0, \mathbf{x}_1, \ldots\}$.

The basic idea behind the Viterbi algorithm is as follows. A coded sequence $\{\mathbf{c}_0, \mathbf{c}_1, \ldots\}$, or its signal-mapped equivalent $\{\mathbf{a}_0, \mathbf{a}_1, \ldots\}$, corresponds to a path through the encoder trellis. Due to noise in the channel, the received sequence $\mathbf{r}$ may not correspond exactly to a path through the trellis. The decoder finds a path through the trellis which is closest to the received sequence, where the measure of "closest" is determined by the likelihood function appropriate for the channel. In light of (12.14), for an AWGN channel the maximum likelihood path corresponds to the path through the trellis which is closest in *Euclidean* distance to $\mathbf{r}$. In light of (12.15), for a BSC the maximum likelihood path corresponds to the path through the trellis which is closest in *Hamming* distance to $\mathbf{r}$. Naively, one could find the maximum likelihood path by computing separately the path lengths of all of the possible paths through the trellis. This, however, is computationally intractable. The Viterbi algorithm organizes the computations in an efficient recursive form.

For an input $\mathbf{x}_t$ the output $\mathbf{c}_t$ depends on the state of the encoder Ψ_t, which in turn depends upon previous inputs. This dependency among inputs means that optimal decisions cannot be made based upon a likelihood function for a single time $f(\mathbf{r}_t|\mathbf{x}_t)$. Instead, optimal decisions are based upon an entire received *sequence* of symbols. The likelihood function to be maximized is thus $f(\mathbf{r}|\mathbf{x})$, where

$$f(\mathbf{r}|\mathbf{x}) = f(\mathbf{r}_0^{L-1}|\mathbf{x}_0^{L-1}) = f(\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{L-1}|\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{L-1}) = \prod_{t=0}^{L-1} f(\mathbf{r}_t|\mathbf{x}_t).$$

The fact that the channel is assumed to be memoryless is used to obtain the last equality. It

is convenient to deal with the log likelihood function,

$$\log f(\mathbf{r}|\mathbf{x}) = \sum_{t=0}^{L-1} \log f(\mathbf{r}_t|\mathbf{x}_t).$$

Consider now a sequence $\hat{\mathbf{x}}_0^{t-1} = \{\hat{\mathbf{x}}_0, \hat{\mathbf{x}}_1, \ldots, \hat{\mathbf{x}}_{t-1}\}$ which leaves the encoder in state $\Psi_t = p$ at time t. This sequence determines a path — or sequence of states — through the trellis for the code, which we denote (abstractly) as Π_t or $\Pi_t(\hat{\mathbf{x}}_0^{t-1})$. Thus

$$\Pi_t = \{\Psi_0, \Psi_1, \ldots, \Psi_t\}.$$

The log likelihood function for this sequence is

$$\log f(\mathbf{r}_0^{t-1}|\hat{\mathbf{x}}_0^{t-1}) = \sum_{i=0}^{t-1} \log f(\mathbf{r}_i|\hat{\mathbf{x}}_i).$$

Let $M_{t-1}(p) = -\log f(\mathbf{r}_0^{t-1}|\hat{\mathbf{x}}_0^{t-1})$ denote the **path metric** for the path Π_t through the trellis defined by the sequence $\hat{\mathbf{x}}_0^{t-1}$ and terminating in state p. (We could write $M_{t-1}(p; \Pi_t)$ or $M_{t-1}(p; \hat{\mathbf{x}}_0^{t-1})$ to indicate that the metric depends on the path but this leads to an awkward notation.) The negative sign in this definition means that we seek to *minimize* the path metric (to maximize the likelihood).

Now let the sequence $\hat{\mathbf{x}}_0^t = \{\hat{\mathbf{x}}_0, \hat{\mathbf{x}}_1, \ldots, \hat{\mathbf{x}}_t\}$ be obtained by appending the input $\hat{\mathbf{x}}_t$ to $\hat{\mathbf{x}}_0^{t-1}$ and suppose the input $\hat{\mathbf{x}}_t$ is such that the state at time $t+1$ is $\Psi_{t+1} = q$. The path metric for this longer sequence is

$$M_t(q) = -\sum_{i=0}^{t} \log f(\mathbf{r}_i|\hat{\mathbf{x}}_i) = -\sum_{i=0}^{t-1} \log f(\mathbf{r}_i|\hat{\mathbf{x}}_i) - \log f(\mathbf{r}_t|\hat{\mathbf{x}}_t) = M_{t-1}(p) - \log f(\mathbf{r}_t|\hat{\mathbf{x}}_t).$$

Let $\mu_t(\mathbf{r}_t, \hat{\mathbf{x}}_t) = -\log f(\mathbf{r}_t|\hat{\mathbf{x}}_t)$ denote the negative log likelihood for this input. As pointed out in (12.16), we could equivalently use

$$\mu_t(\mathbf{r}_t, \hat{\mathbf{x}}_t) = a[-\log f(\mathbf{r}_t|\hat{\mathbf{x}}^{(p,q)}) - b], \tag{12.17}$$

for any $a > 0$. The quantity $\mu_t(\mathbf{r}_t, \hat{\mathbf{x}}_t)$ is called the **branch metric** for the decoder. Since $\hat{\mathbf{x}}_t$ moves the trellis from state p at time t to state q at time $t+1$, we can write $\mu_t(\mathbf{r}_t, \hat{\mathbf{x}}_t)$ as $\mu_t(\mathbf{r}_t, \hat{\mathbf{x}}^{(p,q)})$. Then

$$M_t(q) = \sum_{i=0}^{t-1} \mu_t(\mathbf{r}_i, \hat{\mathbf{x}}_i) + \mu_t(\mathbf{r}_t, \hat{\mathbf{x}}_t) = M_{t-1}(p) + \mu_t(\mathbf{r}_t, \hat{\mathbf{x}}^{(p,q)}).$$

That is, the path metric along a path to state q at time t is obtained by adding the path metric to the state p at time $t-1$ to the branch metric for an input which moves the encoder from state p to state q. (If there is no such input then the branch metric is ∞.)

With this notation, we now come to the crux of the Viterbi algorithm: What do we do when paths merge? Suppose $M_{t-1}(p_1)$ is the path metric of a path ending at state p_1 at time t and $M_{t-1}(p_2)$ is the path metric of a path ending at state p_2 at time t. Suppose further that both of these states are connected to state q at time $t+1$, as suggested in Figure 12.11. The resulting path metrics to state q are

$$M_{t-1}(p_1) + \mu_t(\mathbf{r}_t, \hat{\mathbf{x}}^{(p_1,q)}) \quad \text{and} \quad M_{t-1}(p_2) + \mu_t(\mathbf{r}_t, \hat{\mathbf{x}}^{(p_2,q)}).$$

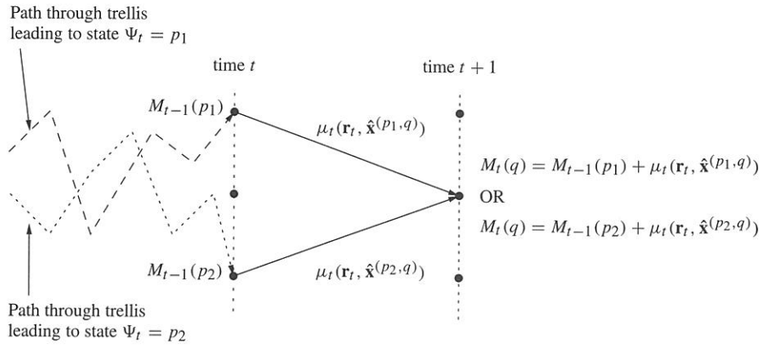


Figure 12.11: The Viterbi step: Select the path with the best metric.

The governing principle of the Viterbi algorithm is this: To obtain the shortest path through the trellis, the path to state q must be the shortest possible. Otherwise, it would be possible to find a shorter path through the trellis by finding a shorter path to state q. (This is Bellman's principle of optimality; see, e.g., [17].) Thus, when the two or more paths merge, the path with the *shortest* path metric is retained and the other path is eliminated from further consideration. That is,

$$M_t(q) = \min\{M_{t-1}(p_1) + \mu_t(\mathbf{r}_t, \hat{\mathbf{x}}^{(p_1,q)}), M_{t-1}(p_2) + \mu_t(\mathbf{r}_t, \hat{\mathbf{x}}^{(p_2,q)})\}$$

and the path with minimal length becomes the path to state q. This is called the **survivor path**.

Since it is not known at time $t < L$ which states the final path passes through, the paths to *each* state are found for each time. The Viterbi algorithm thus maintains the following data:

- A path metric to each state at time t.
- A path to each state at time t.

The Viterbi algorithm is thus summarized as follows:

1. For each state q at time $t+1$, find the path metric for each path to state q by adding the path metric $M_{t-1}(p)$ of each survivor path to state p at time t to the branch metric $\mu_t(\mathbf{r}_t, \hat{\mathbf{x}}^{(p,q)})$.
2. The survivor path to q is selected as that path to state q which has the smallest path metric.
3. Store the path and path metric to each state q.
4. Increment t and repeat until complete.

In the event that the path metrics of merging paths are equal, a random choice can be made with no negative impact on the likelihood.

More formally, there is the description in Algorithm 12.1. In this description, the path to state q is specified by listing for each state its predecessor in the graph. Other descriptions of the path are also possible. The algorithm is initialized reflecting the assumption that the initial state is 0 by setting the path metric at state 0 to 0 and all other path metrics to ∞ (i.e., some large number).

Algorithm 12.1 The Viterbi Algorithm

1 **Input:** A sequence $\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{L-1}$
2 **Output:** The sequence $\hat{\mathbf{x}}_0, \hat{\mathbf{x}}_1, \ldots, \hat{\mathbf{x}}_{L-1}$ which maximizes the likelihood $f(\mathbf{r}_0^{L-1}|\hat{\mathbf{x}}_0^{L-1})$.
3 **Initialize:** Set $M(0) = 0$ and $M(p) = \infty$ for $p = 1, 2, \ldots, 2^\nu - 1$ (initial path costs)
4 Set $\Pi_p = \emptyset$ for $p = 0, 1, \ldots, 2^\nu - 1$ (initial paths)
5 Set $t = 0$
6 Begin
7 For each state q at time $t+1$
8 Find the path metric for each path to state q:
9 for each p_i connected to state q corresponding to input $\hat{\mathbf{x}}^{(p_i,q)}$, compute
$m_i = M(p_i) + \mu_t(\mathbf{r}_t, \hat{\mathbf{x}}^{(p_i,q)})$).
10 Select the smallest metric $M(q) = \min_i m_i$ and the corresponding predecessor state p.
11 Extend the path to state q: $\Pi_q = [\Pi_p\ p]$
12 end (for)
13 $t = t + 1$
14 if $t < L - 1$, goto line 6.
15 Termination:
16 If terminating in a known state (e.g. 0)
Return the sequences of inputs along the path to that known state
17 If terminating in any state
Find final state with minimal metric; Return the sequence of inputs along that path to that state.
18 End

The operations of extending and pruning that constitute the heart of the Viterbi algorithm are summarized as:

$$M_t(q) = \underbrace{\min_p \underbrace{[M_{t-1}(p) + \mu_t(\mathbf{r}_t, \hat{\mathbf{x}}^{(p,q)})]}_{\text{Extend all paths at time } t \text{ to state } q \ \ldots}}_{\text{Then choose smallest cost}}. \tag{12.18}$$

Example 12.13 Consider the encoder

$$G(x) = \begin{bmatrix} x^2 + 1 & x^2 + x + 1 \end{bmatrix}$$

of Example 12.1, whose realization and trellis diagram are shown in Figure 12.5, passing the data through a BSC. When the data sequence

$$\begin{aligned} \mathbf{m} &= [1, 1, 0, 0, 1, 0, 1, 0, \ldots] \\ &= [m_0, m_1, m_2, m_3, m_4, m_5, m_6, m_7, \ldots] \end{aligned}$$

is applied to the encoder, the coded output bit sequence is

$$\begin{aligned} \mathbf{c} &= [11, 10, 10, 11, 11, 01, 00, 01, \ldots] \\ &= [\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3, \mathbf{c}_4, \mathbf{c}_5, \mathbf{c}_6, \mathbf{c}_7, \ldots]. \end{aligned}$$

For the BSC, we take the mapped data the same as the encoder output $\mathbf{a}_t = \mathbf{c}_t$. The output sequence and corresponding states of the encoder are shown here, where $\Psi_0 = 0$ is the initial state.

t	Input m_k	Output $\mathbf{c}_t$	State Ψ_{t+1}
0	1	11	1
1	1	10	3
2	0	10	2
3	0	11	0
4	1	11	1
5	0	01	2
6	1	00	1
7	0	01	2

The sequence of states through the trellis for this encoder is shown in Figure 12.12; the solid line shows the state sequence for this sequence of outputs. The coded output sequence passes through a

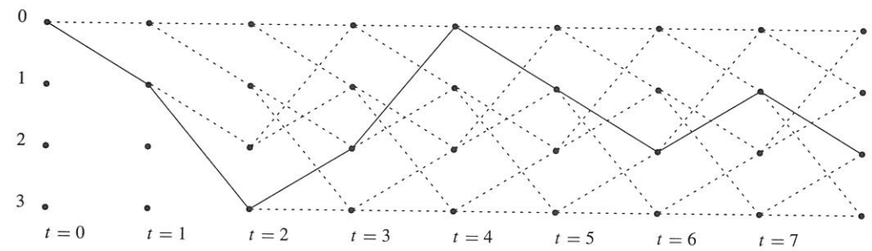


Figure 12.12: Path through trellis corresponding to true sequence.

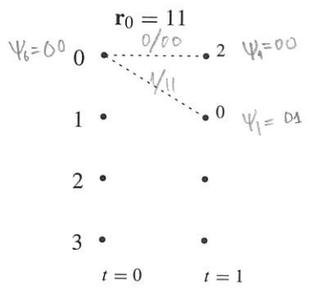
channel, producing the received sequence

$$\mathbf{r} = [11\ 10\ \underline{0}0\ 1\underline{0}\ 11\ 01\ 00\ 01\ \ldots] = [\mathbf{r}_0, \mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3, \mathbf{r}_4, \mathbf{r}_5, \mathbf{r}_6, \mathbf{r}_7, \ldots].$$

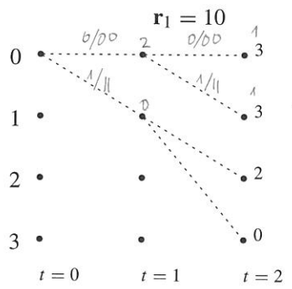
The two underlined bits are flipped by noise in the channel.

The algorithm proceeds as follows:

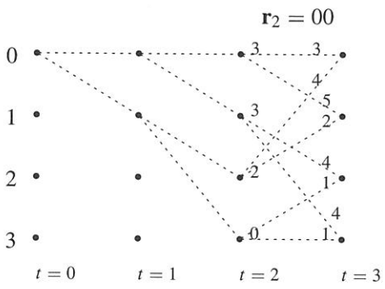
$t = 0$: The received sequence is $\mathbf{r}_0 = 11$. We compute the metric to each state at time $t = 1$ by finding the (Hamming) distance between $\mathbf{r}_0$ and the possible transmitted sequence $\mathbf{c}_0$ along the branches of the first stage of the trellis. Since state 0 was known to be the initial state, we end up with only two paths, with path metrics 2 and 0, as shown here:



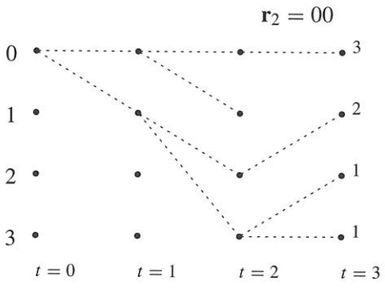
$t = 1$: The received sequence is $\mathbf{r}_1 = 10$. Again, each path at time $t = 1$ is simply extended, adding the path metric to each branch metric:



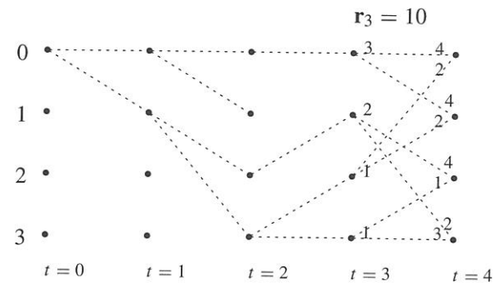
$t = 2$: The received sequence is $\mathbf{r}_2 = 00$. Each path at time $t = 2$ is extended, adding the path metric to each branch metric of each path.



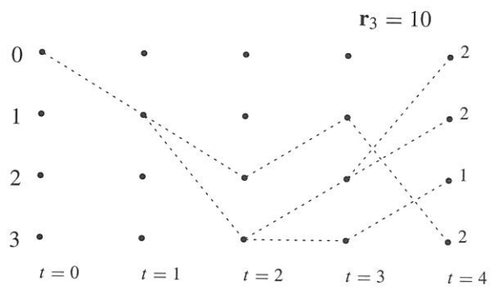
There are now multiple paths to each node at time $t = 3$. We select the path to each node with the best metric and eliminate the other paths. This gives the diagram as follows:



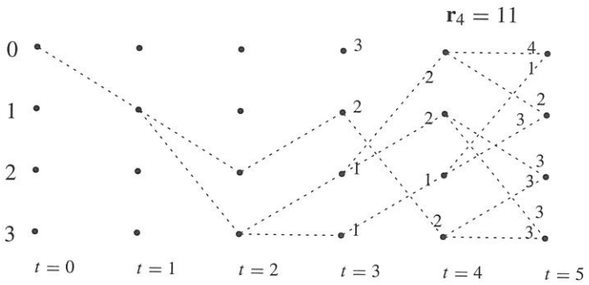
$t = 3$: The received sequence is $\mathbf{r}_3 = 10$. Each path at time $t = 3$ is extended, adding the path metric to each branch metric of each path.



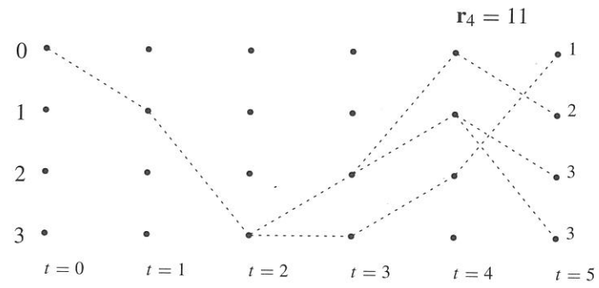
Again, the best path to each state is selected. We note that in selecting the best paths, some of the paths to some states at earlier times have no successors; these orphan paths are deleted now in our portrayal:



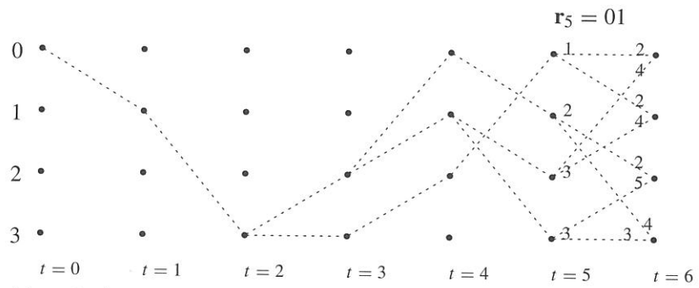
$t = 4$: The received sequence is $\mathbf{r}_4 = 11$. Each path at time $t = 4$ is extended, adding the path metric to each branch metric of each path.



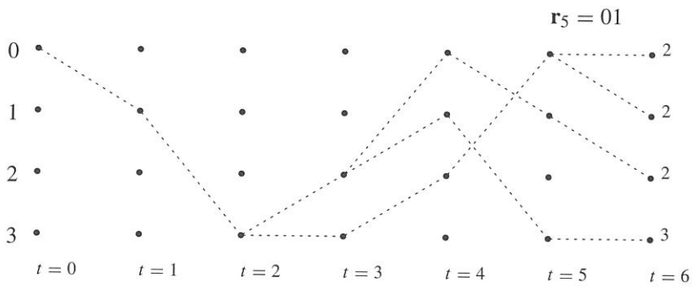
In this case, we note that there are multiple paths into state 3 which both have the same path metric; also there are multiple paths into state 2 with the same path metric. Since one of the paths must be selected, the choice can be made arbitrarily (e.g., at random). After selecting and pruning of orphan paths we obtain:



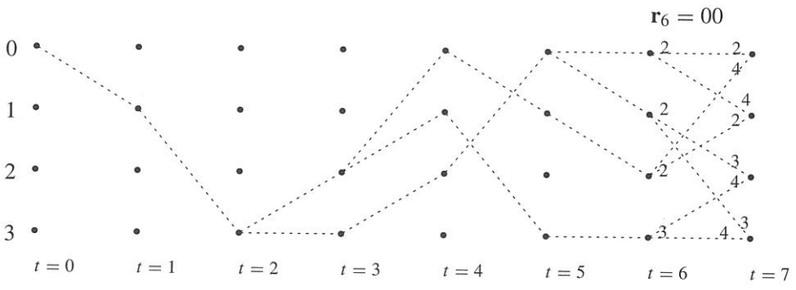
$t = 5$: The received sequence is $\mathbf{r}_5 = 01$.



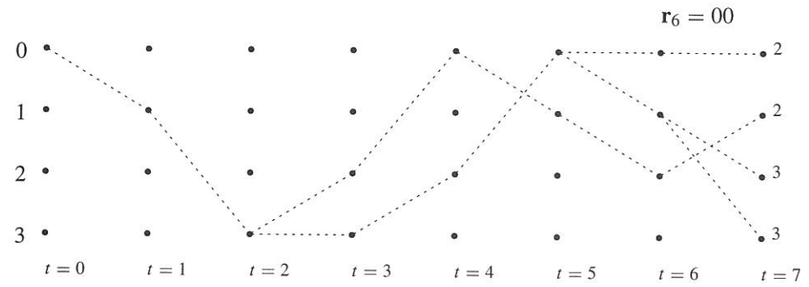
After selecting and pruning:



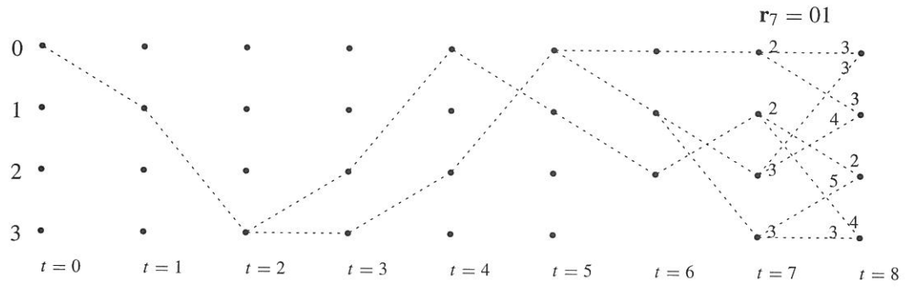
$t = 6$: The received sequence is $\mathbf{r}_6 = 00$.



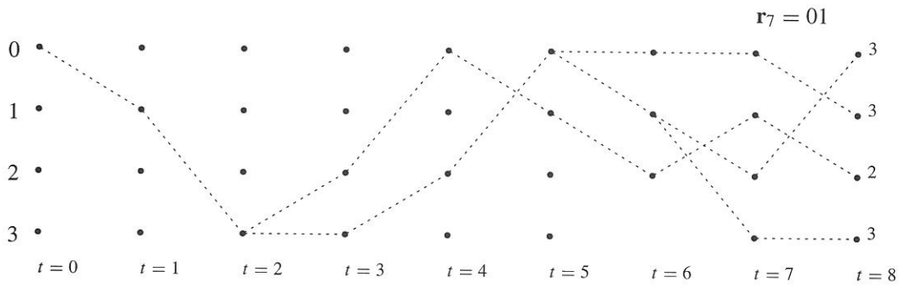
After selecting and pruning:



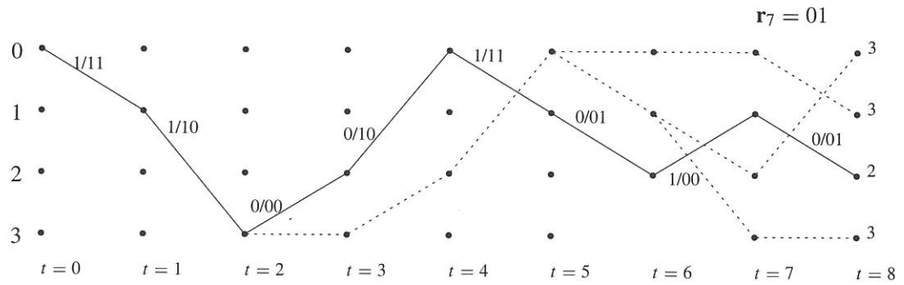
$t = 7$: The received sequence is $\mathbf{r}_7 = 01$.



After selecting and pruning:



The decoding is finalized at the end of the transmission (the 16 received data bits) by selecting the state at the last stage having the lowest cost, traversing backward along the path so indicated to the beginning of the trellis, then traversing forward again along the best path, reading the input bits and decoded output bits along the path. This is shown with the solid line below; input/output pairs are indicated on each branch.



Note that the path through the trellis is the same as in Figure 12.12 and that the recovered input bit sequence is the same as the original bit sequence. Thus, out of this sequence of 16 bits, two bit errors have been corrected. □

12.3.3 Some Implementation Issues

The Basic Operation: Add-Compare-Select

The basic operation of the Viterbi algorithm is Add-Compare-Select (ACS): Add the branch metric to the path metric for each path leading to a state; compare the resulting path metrics at that state; and select the better metric. A schematic of this idea appears in Figure 12.13. High-speed operation can be obtained in hardware by using a bank of 2^{ν} such ACS units in parallel. A variation on this theme, the compare-select-add (CSA) operation, is capable of somewhat improving the speed for some encoders. The algorithm employing CSA is called the *differential Viterbi algorithm*; see [105] for a description.

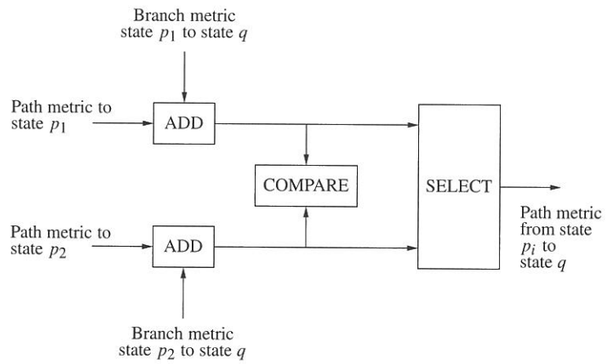


Figure 12.13: Add-compare-select Operation.

Decoding Streams of Data: Windows on the Trellis

In convolutional codes, data are typically encoded in a stream. Once the encoding starts, it may continue indefinitely, for example until the end of a file or until the end of a data transmission session. If such a data stream is decoded using the Viterbi algorithm as described above, the paths through the trellis would have to have as many stages as the code is long.

For a long data stream, this could amount to an extraordinary amount of data to be stored, since the decoder would have to store 2^{ν} paths whose lengths grow longer with each stage. Furthermore, this would result in a large decoding latency: strictly speaking it would not be possible to output *any* decoded values until the maximum likelihood path is selected at the end of the file.

Fortunately, it is not necessary to wait until the end of transmission. Consider the paths in Example 12.13. In this example, by $t = 4$, there is a single surviving path in the first two stages of the trellis. Regardless of how the Viterbi algorithm operates on the paths as it continues through the trellis, those first two stages could be unambiguously decoded.

In general, with very high probability there is a single surviving path some number of stages back from the "current" stage of the trellis. The initial stages of the survivor paths tend to merge if a sufficient decoding delay is allowed. Thus, it is only necessary to keep a "window" on the trellis consisting of the current stage and some number of previous stages. The number of stages back that the decoding looks to make its decision is called the **decoding depth**, denoted by Γ. At time t the decoder outputs a decision on the code bits $\mathbf{c}_{t-\Gamma}$. While it is possible to make an incorrect decoding decision on a finite decoding depth, this error, called the **truncation error**, is typically very small if the decoding depth is sufficiently large. It has been found (see, e.g., [90, 149]) that if a decoding depth of about five to ten constraint lengths is employed, then there is very little loss of performance compared to using the full length due to truncation error.

It is effective to implement the decoding window using a circular queue of length Γ to hold the current window. As the window is "shifted," it is only necessary to adjust the pointers to the beginning and end of the window.

As the algorithm proceeds through the stream of data, the path metrics continue to accumulate. Overflow is easily avoided by periodically subtracting from all path metrics an equal amount (for example, the smallest path metric). The path metrics then show the differential qualities of each path rather than the absolute metrics (or their approximations), but this is sufficient for decoding purposes.

Output Decisions

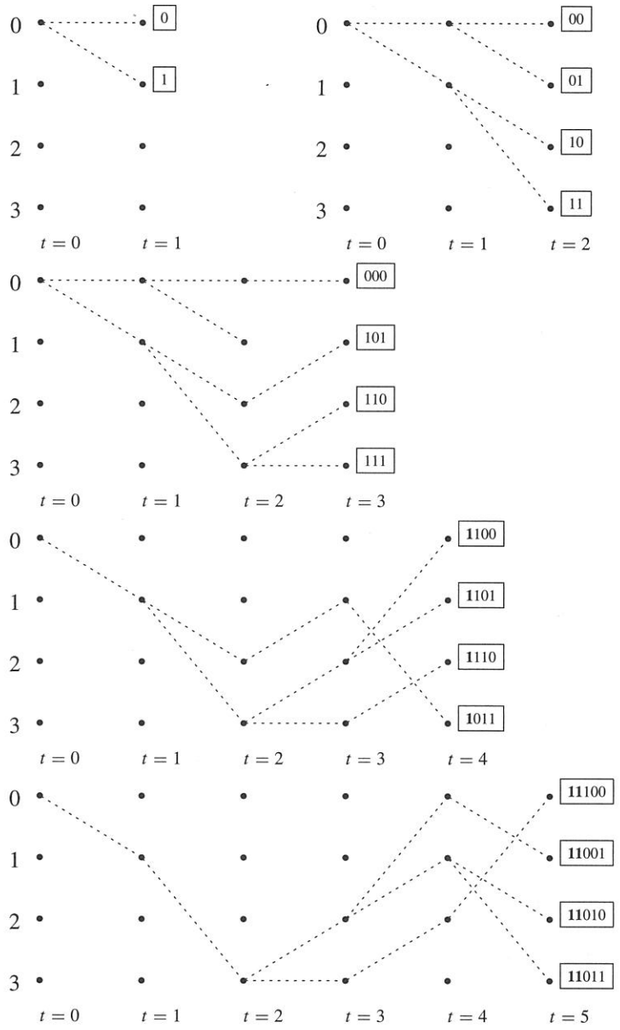
When a decision about the output $\mathbf{c}_{t-\Gamma}$ is to be made at time t, there are a few ways that this can be accomplished [373]: Output $\mathbf{c}_{t-\Gamma}$ on a randomly selected survivor path; Output $\mathbf{c}_{t-\Gamma}$ on the survivor path with the best metric; Output $\mathbf{c}_{t-\Gamma}$ that occurs most often among all the survivor paths; Output $\mathbf{c}_{t-\Gamma}$ on any path. In reality, if Γ is sufficiently large that all the survivor paths have merged Γ decoding stages back, then the performance difference among these alternatives is very small.

When the decision is to be output, a survivor path is selected (using one of the methods just mentioned). Then it is necessary to determine the $\mathbf{c}_{t-\Gamma}$. There are a couple of ways of accomplishing this, the register exchange and the traceback.

In the *register exchange* implementation, an input register at each state contains the sequence of input bits associated with the surviving path that terminates at that state. A register capable of storing the $k\Gamma$ bits is necessary for each state. As the decoding algorithm proceeds, for the path selected from state p to state q, the input register at state q is obtained by copying the input register for state p and appending the k input bits resulting in that state transition. (Double buffering of the data may be necessary, so that the input registers are not lost in copying the information over.) When an output is necessary, the first k bits of the register for the terminating state of the selected path can be read immediately from its

input register.

Example 12.14 For the decoding sequence of Example 12.13, the registers for the register exchange algorithm are shown here (boxed) for the first five steps of the algorithm.



□

The number of initial bits all the registers have in common is the number of branches of the path that are shared by all paths. These common input bits are shown in bold above. In some implementations, it may be of interest to output only those bits corresponding to path branches which have merged.

In the *traceback* method, the path is represented by storing the *predecessor* to each state. This sequence of predecessors is traced back Γ stages. The state transition $\Psi_{t-\Gamma}$ to $\Psi_{t-\Gamma+1}$ determines the output $\mathbf{c}_{t-\Gamma}$ and its corresponding input bits $\mathbf{x}_{t-\Gamma}$.

Example 12.15 The table

t:	1	2	3	4	5	6	7	8
State	Previous State/Input							
0:	0/0	0/0	0/0	**2/0**	2/0	0/0	0/0	2/0
1:	**0/1**	0/1	2/1	2/1	**0/1**	0/1	**2/1**	0/1
2:	-	1/0	**3/0**	3/0	1/0	**1/0**	1/0	**1/0**
3:	-	**1/1**	3/1	2/1	1/1	3/1	1/1	3/1

shows the previous state traceback table which would be built up by the decoding of Example 12.13. For example, at time $t = 8$, the predecessor of state 0 is 2, the predecessor of state 1 is 0, and so forth. Starting from state 2 (having the lowest path cost), the sequence of states can be read off in reverse order from this table (the bold entries):

$$2 \to 1 \to 2 \to 1 \to 0 \to 2 \to 3 \to 1 \to 0.$$

Thus the first state transition is from state 0 to state 1 and the input at that time is a 1.

The inputs for the entire sequence can also be read off, starting at the right, 11001010. □

In the traceback method, it is necessary to trace backward through the trellis once for each output. (As a variation on the theme, the predecessor to a state could be represented by the input bits that lead to that state.)

In comparing the requirements for these two methods, we note that the register exchange method requires shuffling registers among all the states at each time. In contrast, the traceback method requires no such shuffling, but it does require working back through the trellis to obtain a decision. Which is more efficient depends on the particular hardware available to perform the tasks. Typically, the traceback is regarded as faster but more complicated.

Hard and Soft Decoding; Quantization

Example 12.13 presents an instance of *hard-decision decoding*. If the outputs of a Gaussian channel had been used with the Euclidean distance as the branch metric, then *soft-decision* decoding could have been obtained. Comparing soft decision decoding using BPSK over an AWGN with hard decision decoding over a BSC, in which received values are converted to binary values with a probability of error of $p_c = Q(\sqrt{2E_b/N_0})$ (see Section 1.5.6), it has been determined that soft-decision decoding provides 2 to 3 dB of gain over hard-decision decoding.

For a hard-decision metric, $\mu_t(\mathbf{r}_t, \mathbf{x}^{(p,q)})$ can be computed and stored in advance. For example, for an $n = 2$ binary code, there are four possible received values, 00, 01, 10, 11, and four possible transmitted values. The metric could be stored in a 4×4 array, such as the following.

$\mu_t(\mathbf{r}_t, \hat{\mathbf{x}}^{(p,q)})$	$\mathbf{r}_t = 00$	01	10	11
$\mathbf{x}^{(p,q)} = 00$	0	1	1	2
$\mathbf{x}^{(p,q)} = 01$	1	0	2	1
$\mathbf{x}^{(p,q)} = 10$	1	2	0	1
$\mathbf{x}^{(p,q)} = 11$	2	1	1	0

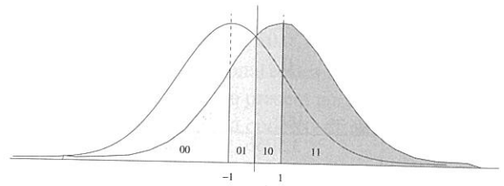


Figure 12.14: A two-bit quantization of the soft-decision metric.

Soft-decision decoding typically requires more expensive computation than hard-decision decoding. Furthermore, soft-decision decoding cannot exactly precompute these values to reduce the ongoing decoding complexity, since $\mathbf{r}_t$ takes on a continuum of values. Despite these disadvantages, it is frequently desirable to use soft-decision decoding because of its superior performance. A computational compromise is to *quantize* the received value to a reasonably small set of values, then precompute the metrics for each of these values. By converting these metrics to small integer quantities, it is possible to efficiently accumulate the metrics. It has been found [148] that quantizing each $r_t^{(i)}$ into 3 bits (eight quantization levels) results in a loss in coding gain of around only 0.25 dB. It is possible to trade metric computation complexity for performance, using more bits of quantization to reduce the loss.

As noted above, if a branch metric μ is modified by $\tilde{\mu} = a\mu + b$ for any $a > 0$ and any real b, an equivalent decoding algorithm is obtained; this simply scales and shifts the resulting path metrics. In quantizing, it may be convenient to find scale factors which make the arithmetic easier.

A widely used quantizer is presented in Section 12.4.

Example 12.16 A two-bit quantizer. In a BPSK-modulated system the transmitted signal amplitudes are $a = 1$ or $a = -1$. The received signal r_t is quantized by a quantization function $Q[\cdot]$ to obtain quantized values

$$q_t = Q[r_t]$$

using quantization thresholds at ± 1 and 0, as shown in Figure 12.14, where the quantized values are denoted as 00, 01, 10 and 11. These thresholds determine the regions $\mathcal{R}_q$. That is,

$$q_t = Q[r_t] = \begin{cases} 00 & \text{if } r_t \in \mathcal{R}_{00} = (-\infty, -1] \\ 01 & \text{if } r_t \in \mathcal{R}_{01} = (-1, 0] \\ 10 & \text{if } r_t \in \mathcal{R}_{10} = (0, 1] \\ 11 & \text{if } r_t \in \mathcal{R}_{11} = (1, \infty). \end{cases}$$

(This is not an optimal quantizer, merely convenient.) For each quantization bin we can compute the likelihood that r_t falls in that region, given a particular input, as

$$P(q_t|a) = \int_{\mathcal{R}_{q_t}} f_R(r|a)\, dr.$$

For example,

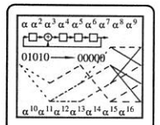
$$P(00|a = -1) = \int_{-\infty}^{-1} f_R(r|-1)\, dr = \frac{1}{2}.$$

Suppose the likelihoods for all quantized points are computed as follows.

$P(q_t\|a)$	$q_t =$	00	01	10	11
$a=1$		0.02	0.14	0.34	0.5
$a=-1$		0.5	0.34	0.14	0.02

The $-\log$ probabilities are

$-\log(P(q_t\|a))$	$q_t =$	00	01	10	11
$a=1$		3.5066	2.0402	1.0788	0.6931
$a=-1$		0.6931	1.0788	2.0402	3.5066



The logarithms of these probabilities can be approximated as integer values by computing

$$-1.619(\log P(q_t|a - \log P(00|1))$$

(the factor $a = 1.619$ was found by a simple computer search and b was chosen to make the smallest value 0), resulting in the following metrics:

$a(-\log(P(q_t\|a)-b)$	$q_t =$	00	01	10	11
$a=1$		5.0028	2.109	0.618	0
$a=-1$		0	0.618	2.109	5.0028

which can be rounded to

$\lfloor a(-\log(P(q_t\|a)-b)\rfloor$	$q_t =$	00	01	10	11
$a=1$		5	2	1	0
$a=-1$		0	1	2	5

Although the signal is quantized into two bits, the metric requires three bits to represent it. With additional loss of coding gain, this could be reduced to two bits of metric (reducing the hardware required to accumulate the path metrics). For example, the first row of the metric table could be approximated as 3, 2, 1, 0.

Note that, by the symmetry of the pdf and constellation, both rows of the table have the same values, so that in reality only a single row would need to be saved in an efficient hardware implementation. □

Synchronization Issues

The decoder must be synchronized with the stream of incoming data. If the decoder does not know which of the n symbols in a block initiates a branch of the trellis, then the data will be decoded with a very large number of errors. Fortunately, the decoding algorithm can detect this. If the data are correctly aligned with the trellis transitions, then with high probability, one (or possibly two) of the path metrics are significantly smaller than the other path metrics within a few stages of decoding. If this does not occur, the data can be shifted relative to the decoder and the decoding re-initialized. With at most n tries, the decoder can obtain symbol synchronization.

Many carrier tracking devices employed in communication systems experience a phase ambiguity. For a BPSK system, it is common that the phase is determined only up to $\pm\pi$, resulting in a sign change. For QPSK or QAM systems, the phase is often known only up to a multiple of $\pi/2$. The decoding algorithm can possibly help determine the absolute phase. For example, in a BPSK system if the all ones sequence is not a codeword, then for a given code sequence $\mathbf{c}$, $1 + \mathbf{c}$ cannot be a codeword. In this case, if decoding seems to indicate that no path is being decoded correctly (i.e., no path seems to emerge as a strong candidate compared to the other paths), then the receiver can complement all of its zeros and ones and decode again. If this decodes correctly the receiver knows that it has the phase off by π. For a QPSK system, four different phase shifts could be examined to see when correct decoding behavior emerges.

12.4 Some Performance Results

Bit error rate characterization of convolutional codes is frequently accomplished by simulation and approximation. In this section, we present performance as a function of quantization, constraint length, window length, and codeword size. These results generally follow [148], but have been recomputed here.

Quantization of the metric was discussed in the previous section. In the results here, a simpler quantized metric is used. Assume that BPSK modulation is employed and that the transmitted signal amplitude a is normalized to ± 1. The received signal r is quantized to m bits, resulting in $M = 2^m$ different quantization levels, using uniformly spaced quantization thresholds. The distance between quantization thresholds is Δ. Figure 12.15 shows 4-level quantization with $\Delta = 1$ and 8-level quantization using $\Delta = 0.5$ and $\Delta = \frac{1}{3}$. Rather than compute the log likelihood of the probability of falling in a decision region, in this approach the quantized q value itself is used as the branch metric if the signal amplitude 1 is sent, or the complement $M - q - 1$ is used if -1 is sent. The resulting integer branch metric is computed as shown in Table 12.1. This branch metric is clearly suboptimal, not being an affine transformation of the log likelihood. However, simulations have shown that it performs very close to optimal and it is widely used. Obviously, the performance depends upon the quantization threshold Δ employed. For the 8-level quantizer, the value $\Delta = 0.4$ is employed. For the 16-level quantizer, $\Delta = 0.25$ is used, and for the 4-level quantizer, $\Delta = 1$ is used. We demonstrate below the dependence of the bit-error rate upon Δ.

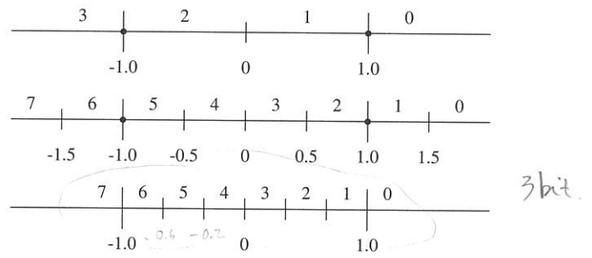
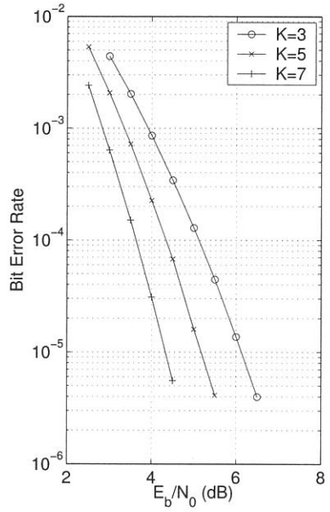


Figure 12.15: Quantization thresholds for 4- and 8-level quantization.

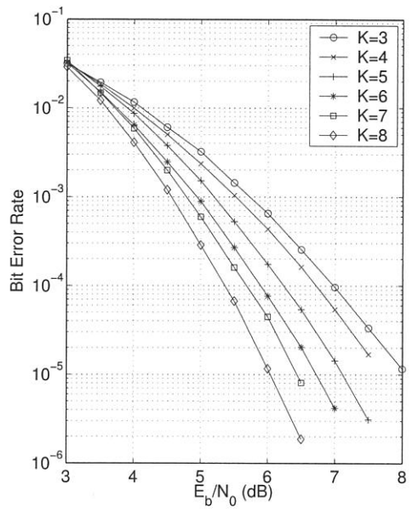
Table 12.1: Quantized Branch Metrics Using Linear Quantization

	Quantization Level							
Signal Amplitude	0	1	2	3	4	5	6	7
	Branch Metric μ							
-1	7	6	5	4	3	2	1	0
1	0	1	2	3	4	5	6	7

Figure 12.16(a) shows the bit error rate as a function of SNR for codes with constraint lengths (here employing $K = 1 + \max \deg(g_j)$ as the constraint length) of $K = 3$, $K = 5$ and $K = 7$ using 8-level uniform quantization with $\Delta = 0.42$. The generators employed



(a) 8-level quantization, $K = 3, 5, 7$.



(b) 1-bit (hard) quantization, $K = 3$ through 8.

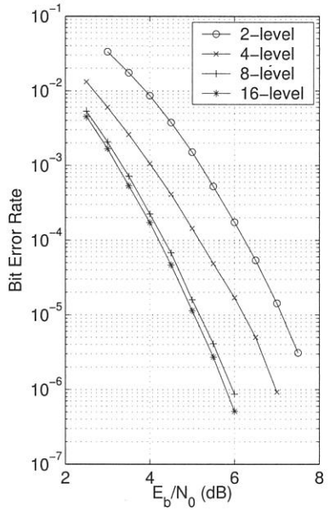
Figure 12.16: Bit error rate as a function of E_b/N_0 of $R = 1/2$ convolutional codes with 32 bit window decoding (following [148]).

in this and the other simulations are the following:

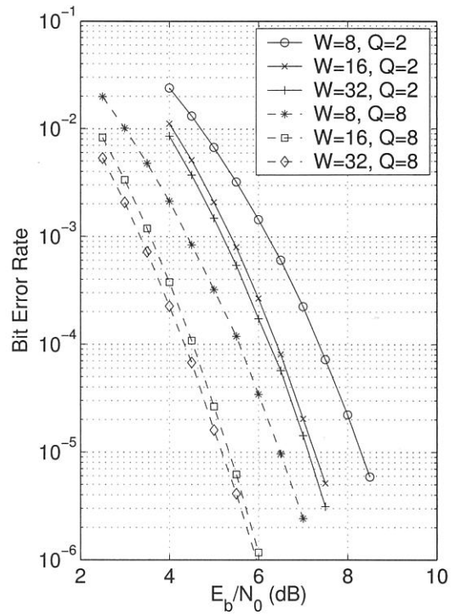
K	$g_1(x)$	$g_2(x)$	d_{free}
3	$1+x^2$	$1+x+x^2$	5
4	$1+x+x^3$	$1+x+x^2+x^3$	6
5	$1+x^3+x^4$	$1+x+x^2+x^4$	7
6	$1+x^2+x^4+x^5$	$1+x+x^2+x^3+x^5$	8
7	$1+x^2+x^3+x^5+x^6$	$1+x+x^2+x^3+x^6$	10
8	$1+x^2+x^5+x^6+x^7$	$1+x+x^2+x^3+x^4+x^7$	10

The Viterbi decoder uses a window of 32 bits. As this figure demonstrates, the performance improves with the constraint length. Figure 12.16(b) shows 1-bit (hard) quantization for $K = 3$ through 8.

Comparisons of the effect of the number of quantization levels and the decoding window are shown in Figure 12.17. In part (a), the performance of a code with $K = 5$ is shown with 2, 4, 8, and 16 quantization levels. As the figure demonstrates, there is very little improvement from 8 to 16 quantization levels; 8 is frequently chosen as an adequate performance/complexity tradeoff. In part (b), again a $K = 5$ code is characterized. In this case, the effect of the length of the decoding window is shown for two different quantization levels. With a decoding window of length 32, most of the achievable performance is attained.



(a) 2-, 4-, 8- and 16-level quantization, decoding window 32.



(b) Decoding window length $W = 8$, 16, and 32; Quantization levels $Q = 2$ and 8.

Figure 12.17: Bit error rate as a function of E_b/N_0 of $R = 1/2$, $K = 5$ convolutional code with different quantization levels and decoding window lengths (following [148]).

Figure 12.18 shows the effect of the quantizer threshold spacing Δ on the performance for an 8-level quantizer with a $K = 5$, $R = 1/2$ code and a $K = 5$, $R = 1/4$ code. The plots are at SNRs of 3.3 dB, 3.5 dB, and 3.7 dB (reading from top to bottom) for the $R = 1/2$ code and 2.75 dB, 2.98 dB, and 3.19 dB (reading from top to bottom) for the $R = 1/4$ code. (These latter SNRs were selected to provide roughly comparable bit error rate for the two codes.) These were obtained by simulating, counting 20,000 bit errors at each point of data.

A convolutional code can be employed as a block code by simply truncating the sequence at a block length N (see Section 12.9). This truncation results in the last few bits in the codeword not having the same level of protection as the rest of the bits, a problem referred to as unequal error protection. The shorter the block length N, the higher the fraction of unequally protected bits, resulting in a higher bit error rate. Figure 12.19 shows BER for maximum likelihood decoding of convolutional codes truncated to blocks of length $N = 200$, $N = 2000$, as well as the "conventional" mode in which the codeword simply streams.

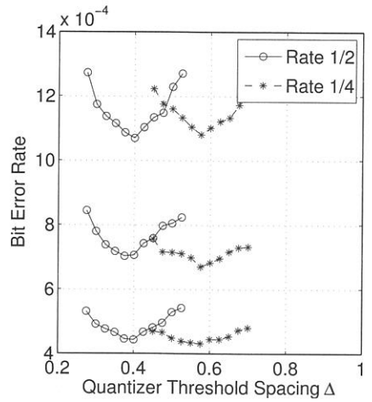


Figure 12.18: Viterbi algorithm bit error rate performance as a function of quantizer threshold spacing.

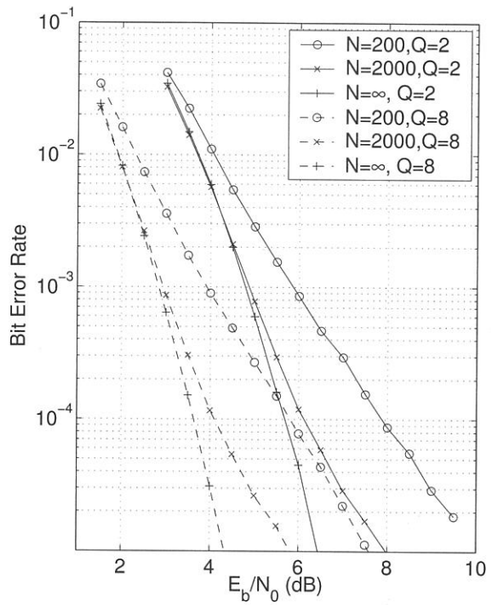


Figure 12.19: BER performance as a function of truncation block length, $N = 200$ and $N = 2000$, for 2- and 8-level quantization.

12.5 Error Analysis for Convolutional Codes

While for block codes it is conventional to determine (or estimate) the probability of decoding a block incorrectly, the performance of convolutional codes is largely determined by the rate and the constraint length. It is not very meaningful to determine the probability of a block in error, since the block may be very long. It is more useful to explore the *probability of bit error*, or the *bit error rate*, which is the average number of message bits in error in a given sequence of bits divided by the total number of message bits in the sequence. We shall denote the bit error rate by P_b. In this section we develop an upper bound for P_b [357].

Consider how errors can occur in the decoding process. The decision mechanism of the Viterbi algorithm operates when two paths join together. If two paths join together and the path with the lower (better) metric is actually the incorrect path, then an incorrect decision is made at that point. We call such an error a *node error* and say that the error event occurs at the place where the paths first diverged. We denote the probability of a node error as P_e. A node error, in turn, could lead to a number of input bits being decoded incorrectly.

Since the code is linear, it suffices to assume that the all-zero codeword is sent: With $d_H(\mathbf{r}, \mathbf{c})$ the Hamming distance between $\mathbf{c}$ and $\mathbf{r}$, we have $d_H(\mathbf{r}, \mathbf{c}) = d_H(\mathbf{r} + \mathbf{c}, \mathbf{c} + \mathbf{c}) = d_H(\mathbf{r} + \mathbf{c}, \mathbf{0})$. Consider the error events portrayed in Figure 12.20. The horizontal line across the top represents the all-zero path through the trellis. Suppose the path diverging from the all-zero path at a has a lower (better) metric when the paths merge at a'. This gives rise to an error event at a. Suppose that there are error events also at b and d. Now consider the path diverging at c: even if the metric is lower (better) at c', the diverging path from c may not ultimately be selected if its metric is worse than the path emerging at b. Similarly the path emerging at d may not necessarily be selected, since the path merging at e may take precedence. This overlapping of decision paths makes the exact analysis of the bit error rate difficult. We must be content with bounds and approximations.

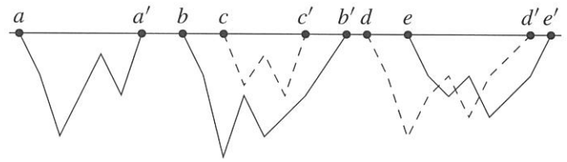


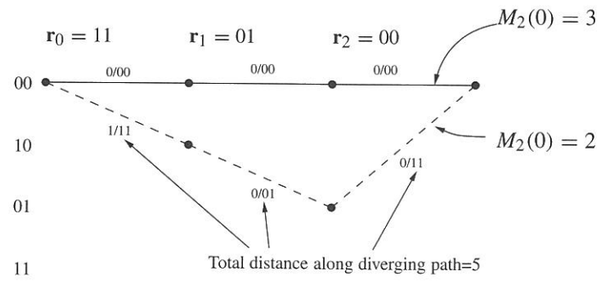
Figure 12.20: Error events due to merging paths.

The following example illustrates some of these issues.

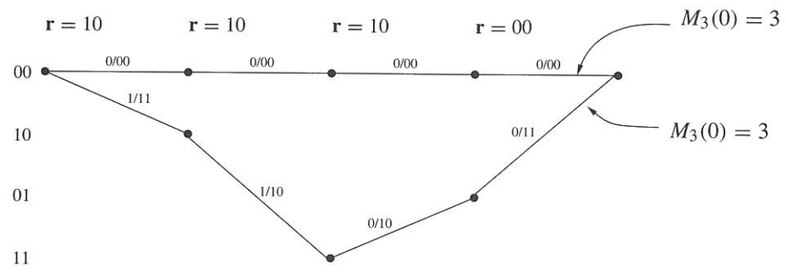
Example 12.17 Consider again the convolutional code from Example 12.1, with

$$G(x) = \left[1 + x^2, 1 + x + x^2\right].$$

Suppose that the input sequence is $\mathbf{x} = [0, 0, 0, 0, \ldots]$ with the resulting transmitted sequence $\mathbf{c} = [00, 00, 00, 00, \ldots]$, but that the received sequence after transmission through a BSC is $\mathbf{r} = [11, 01, 00, \ldots]$. A portion of the decoding trellis for this code is shown in Figure 12.21. After three stages of the trellis when the paths merge, the metric for the lower path (shown as a dashed line) is lower than the metric for the all-zero path (the solid line). Accordingly, the Viterbi algorithm selects the erroneous path, resulting in a node error at the first node. However, while the decision results



(a) Diverging path of distance 5.



(b) Diverging path of distance 6.

Figure 12.21: Two decoding examples.

in three incorrect branches on the path through the trellis, the input sequence corresponding to this selected path is [1, 0, 0], so that only one bit is incorrectly decoded due to this decision.

As the figure shows, there is a path of metric 5 which deviates from the all-zero path. The probability of incorrectly selecting this path is denoted as P_5. This error occurs when the received sequence has three or more errors (1s) in it. In general, we denote

$$P_d = \text{Probability of a decoding error on a path of metric } d.$$

For a deviating path of odd weight, there will be an error if more than half of the bits are in error. The probability of this event for a BSC with crossover probability p_c is

$$P_d = \sum_{i=(d+1)/2}^{d} \binom{d}{i} p_c^i (1-p_c)^{d-i} \quad \text{(with } d \text{ odd)}. \tag{12.19}$$

Suppose now the received signal is $\mathbf{r} = [10, 10, 10, 00]$. Then the trellis appears as in Figure 12.21(b). In this case, the path metrics are equal; one-half of the time the decoder chooses the wrong path. If the incorrect path is chosen, the decoded input bits would be [1, 1, 0, 0], with two bits incorrectly decoded. The probability of the event of choosing the incorrect path in this case is P_6,

where

$$P_d = \frac{1}{2}\binom{d}{d/2} p_c^{d/2}(1-p_c)^{d/2} + \sum_{i=d/2+1}^{d} \binom{d}{i} p_c^i (1-p_c)^{d-i} \quad \text{(with } d \text{ even)} \tag{12.20}$$

is the probability that more than half of the bits are in error, plus $\frac{1}{2}$ times the probability that exactly half the bits are in error. □

We can glean the following information from this example:

- Error events can occur in the decoding algorithm when paths merge together. If the erroneous path has lower (better) path metric than the correct path, the algorithm selects it.
- Merging paths may be of different lengths (number of stages).
- This trellis has a shortest path of metric 5 (three stages long) which diverges from the all-zero path then remerges. We say there is an error path of metric 5. There is also an error path of metric 6 (four stages long) which deviates then remerges.
- When an error path is selected, the number of *input* bits that are erroneously decoded depends on the particular path.
- The probability of a particular error event can be calculated and is denoted as P_d.
- The error path of metric 5 was not disjoint of the error path of metric 6, since they both share a branch.

In the following sections, we first describe how to enumerate the paths through the trellis. Then bounds on the probability of node error and the bit error rate are obtained by invoking the union bound.

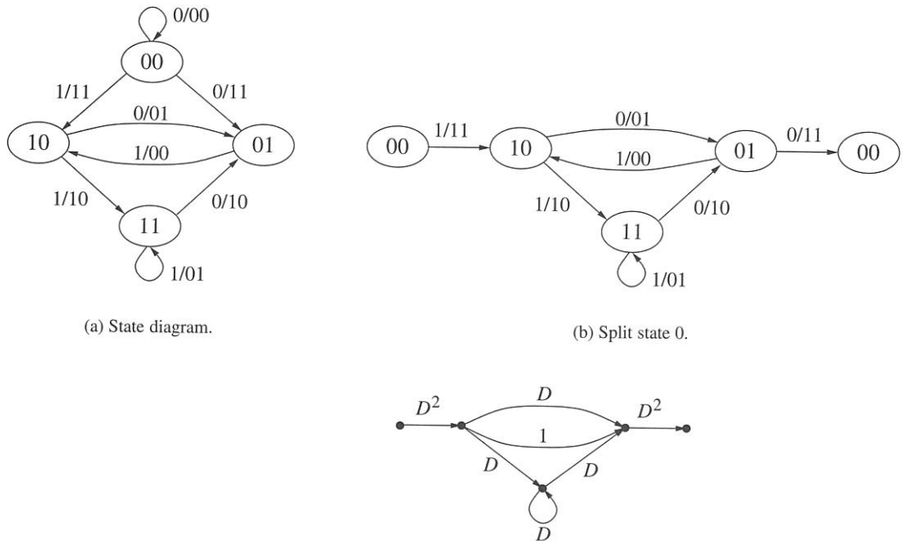
12.5.1 Enumerating Paths Through the Trellis

In computing (or bounding) the overall probability of decoder error, it is expedient to have a method of enumerating all the paths through the trellis. This initially daunting task is aided somewhat by the observation that, for the purposes of computing the probability of error, since the convolutional code is linear it is sufficient to consider only those paths which diverge from the all-zero path then remerge.

We develop a transfer function method which enumerates all the paths that diverge from the all-zero path then remerge. This transfer function is called the **path enumerator**. We demonstrate the technique for the particular code we have been examining.

Example 12.18 Figure 12.22(a) shows the state diagram for the encoder of Example 12.1. In Figure 12.22(b), the 00 state has been "split," or duplicated. Furthermore, the transition from state 00 to state 00 has been omitted, since we are interested only in paths which diverge from the 00 state. Any path through the graph in Figure 12.22(b) from the 00 node on the left to the 00 node on the right represents a path which diverges from the all-zero path then remerges. In Figure 12.22(c), the output codeword of weight i along each edge is represented using D^i. (For example, an output of 11 is represented by D^2; an output of 10 is represented by $D^1 = D$ and an output of 00 is represented by $D^0 = 1$.) For convenience the state labels have been removed.

The labels on the edges in the graph are to be thought of as transfer functions. We now employ the conventional rules for flow graph simplification as summarized in Figure 12.23



(a) State diagram.

(b) Split state 0.

(c) Output weight i represented by D^i.

Figure 12.22: The state diagram and graph for diverging/remerging paths.

- Blocks in series multiply the transfer functions.
- Blocks in parallel add the transfer functions.
- Blocks in feedback configuration employ the rule "forward gain over 1 minus loop gain."

(For a thorough discussion on more complicated flow graphs, see [221].) For the state diagram of Figure 12.22, we take each node as a summing node. The sequence of steps by successively applying the simplification rules is shown in Figure 12.24. Simplifying the final diagram, we find

$$T(D) = D^2 \frac{\frac{D}{1-D}}{1-\frac{D}{1-D}} D^2 = \frac{D^5}{1-2D}.$$

To interpret this, we use the formal series expansion[4] (check by long division)

$$\boxed{\frac{1}{1-D} = 1 + D + D^2 + D^3 + \cdots.}$$

Expanding $T(D)$ we find

$$T(D) = D^5(1 + 2D + (2D)^2 + (2D)^3 + \cdots) = D^5 + 2D^6 + 4D^7 + \cdots + 2^k D^{k+5} + \cdots.$$

Interpreting this, we see that we have:

[4] A formal series is an infinite series that is obtained by symbolic manipulation, without particular regard to convergence.

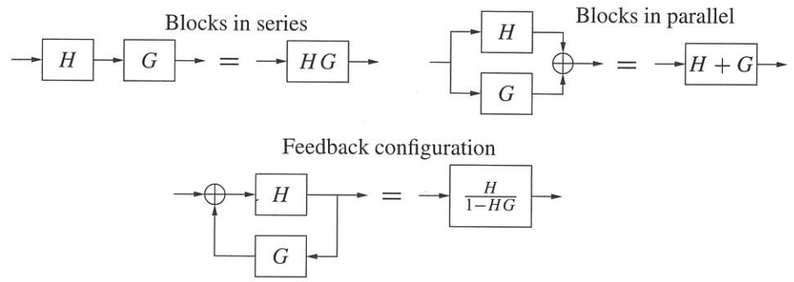


Figure 12.23: Rules for simplification of flow graphs.

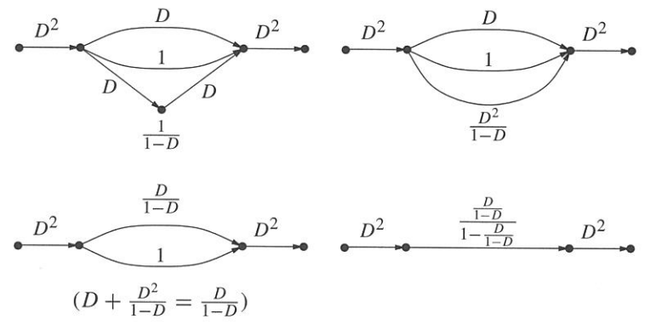


Figure 12.24: Steps simplifying the flow graph for a convolutional code.

- One diverging/remerging error path at metric 5 from the all-zero path;
- 2 error paths of metric 6;
- 4 error paths of metric 7, etc.

Furthermore, the shortest error path has metric 5. □

Definition 12.8 The minimum metric of a path diverging from then remerging to the all-zero path is called the **free distance** of the convolutional code, and is denoted as d_{free}. The number of paths at that metric is denoted as N_{free}. □

In general, we write

$$T(D) = \sum_{d=d_{\text{free}}}^{\infty} a(d) D^d,$$

where $a(d_{\text{free}}) = N_{\text{free}}$.

Additional information about the paths in the trellis can be obtained with a more expressive transfer function. We label each path with three variables: D^i, where i is the output code weight; N^i, where i is the input weight; and L, to account for the length of the branch.

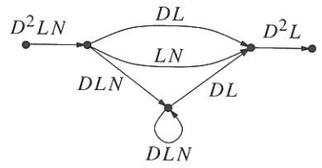


Figure 12.25: State diagram labeled with output weight, input weight, and branch length.

Example 12.19 Returning to the state diagram of Figure 12.22, we obtain the labeled diagram in Figure 12.25. Using the same rules for block diagram simplification as previously, the transfer function for this diagram is

$$T(D, N, L) = \frac{D^5 L^3 N}{1 - DLN(1 + L)}. \tag{12.21}$$

Expanding this as a formal series we have

$$T(D, N, L) = D^5 L^3 N + D^6 L^4 (1 + L) N^2 + D^7 L^5 (1 + L)^2 N^3 + \cdots, \tag{12.22}$$

which has the following interpretation:

- There is one error path of metric 5 which is three branches long (from L^3) and one input bit is 1 (from N^1) along that path.
- There are two error paths of metric 6: one of them is four branches long and the other is five branches long. Along both of them, there are two input bits that are 1.
- There are four error paths of metric 7. One of them is five branches long; two of them are six branches long; and one is seven branches long. Along each of these, there are three input bits that are 1.
- Etc.

□

Clearly, we can obtain the simpler transfer function by $T(D) = T(D, N, L)|_{N=1, L=1}$. When we don't care to keep track of the number of branches, we write

$$T(D, N) = T(D, N, L)|_{L=1}.$$

Enumerating on More Complicated Graphs: Mason's Rule

Some graphs are more complicated than the three rules introduced above can accommodate. A more general approach is Mason's rule [221]. Its generality leads to a rather complicated notation, which we summarize here and illustrate by example (not by proof) [373]. We will enumerate all paths from the 0 state to the 0 state in the state diagram shown in Figure 12.26.

A **loop** is a sequence of states which starts and ends in the same state, but otherwise does not enter any state more than once. We will say that a **forward loop** is a loop that starts and stops in state 0. A set of loops is **nontouching** if they have no vertices in common. Thus $\{0, 1, 2, 4, 0\}$ is a forward loop. The loop $\{3, 6, 5, 3\}$ is a loop that does not touch this forward loop. The set of all forward loops in the graph is denoted as $L = \{L_1, L_2, \ldots\}$. The corresponding set of path gains is denoted as $F = \{F_1, F_2, \ldots\}$. Let $\mathcal{L} = \{\mathcal{L}_1, \mathcal{L}_2, \ldots\}$ denote the set of loops in the graph that does *not* contain the vertex 0. Let $\mathcal{F} = \{\mathcal{F}_1, \mathcal{F}_2, \ldots\}$ be the set of corresponding path gains. (Determining

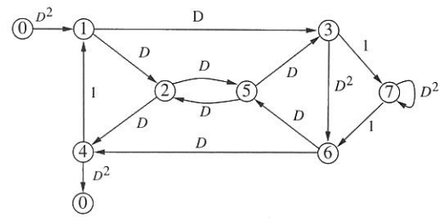


Figure 12.26: A state diagram to be enumerated.

all the loops requires some care.) For the graph in Figure 12.26, the forward loops and their gains are

L_1 :	$\{0, 1, 3, 7, 6, 5, 2, 4, 0\}$	$F_1 = D^8$
L_2 :	$\{0, 1, 3, 7, 6, 4, 0\}$	$F_2 = D^6$
L_3 :	$\{0, 1, 3, 6, 5, 2, 4, 0\}$	$F_3 = D^{10}$
L_4 :	$\{0, 1, 3, 6, 4, 0\}$	$F_4 = D^8$
L_5 :	$\{0, 1, 2, 5, 3, 7, 6, 4, 0\}$	$F_5 = D^8$
L_6 :	$\{0, 1, 2, 5, 3, 6, 4, 0\}$	$F_6 = D^{10}$
L_7 :	$\{0, 1, 2, 4, 0\}$	$F_7 = D^6$

and the other loops and their gains are

$\mathcal{L}_1$:	$\{1, 3, 7, 6, 5, 2, 4, 1\}$	$\mathcal{F}_1 = D^4$
$\mathcal{L}_2$:	$\{1, 3, 7, 6, 4, 1\}$	$\mathcal{F}_2 = D^2$
$\mathcal{L}_3$:	$\{1, 3, 6, 5, 2, 4, 1\}$	$\mathcal{F}_3 = D^6$
$\mathcal{L}_4$:	$\{1, 3, 6, 4, 1\}$	$\mathcal{F}_4 = D^4$
$\mathcal{L}_5$:	$\{1, 2, 5, 3, 7, 6, 4, 1\}$	$\mathcal{F}_5 = D^4$
$\mathcal{L}_6$:	$\{1, 2, 5, 3, 6, 4, 1\}$	$\mathcal{F}_6 = D^6$
$\mathcal{L}_7$:	$\{1, 2, 4, 1\}$	$\mathcal{F}_6 = D^2$
$\mathcal{L}_8$:	$\{2, 5, 2\}$	$\mathcal{F}_8 = D^2$
$\mathcal{L}_9$:	$\{3, 7, 6, 5, 3\}$	$\mathcal{F}_9 = D^2$
$\mathcal{L}_{10}$:	$\{3, 6, 5, 3\}$	$\mathcal{F}_{10} = D^4$
$\mathcal{L}_{11}$:	$\{7, 7\}$	$\mathcal{F}_{11} = D^2$

We also need to identify the *pairs* of nontouching loops in $\mathcal{L}$, the triples of nontouching loops in $\mathcal{L}$, etc., and their corresponding product gains. There are ten pairs of nontouching loops in $\mathcal{L}$:

$(\mathcal{L}_2, \mathcal{L}_8)$	$\mathcal{F}_2\mathcal{F}_8 = D^4$	$(\mathcal{L}_3, \mathcal{L}_{11})$	$\mathcal{F}_3\mathcal{F}_{11} = D^8$
$(\mathcal{L}_4, \mathcal{L}_8)$	$\mathcal{F}_4\mathcal{F}_8 = D^6$	$(\mathcal{L}_4, \mathcal{L}_{11})$	$\mathcal{F}_4\mathcal{F}_{11} = D^6$
$(\mathcal{L}_6, \mathcal{L}_{11})$	$\mathcal{F}_6\mathcal{F}_{11} = D^8$	$(\mathcal{L}_7, \mathcal{L}_9)$	$\mathcal{F}_7\mathcal{F}_9 = D^4$
$(\mathcal{L}_7, \mathcal{L}_{10})$	$\mathcal{F}_7\mathcal{F}_{10} = D^6$	$(\mathcal{L}_7, \mathcal{L}_{11})$	$\mathcal{F}_7\mathcal{F}_{11} = D^4$
$(\mathcal{L}_8, \mathcal{L}_{11})$	$\mathcal{F}_8\mathcal{F}_{11} = D^4$	$(\mathcal{L}_{10}, \mathcal{L}_{11})$	$\mathcal{F}_{10}\mathcal{F}_{11} = D^6$

There are two **triplets** of nontouching loops in $\mathcal{L}$,

$$(\mathcal{L}_4, \mathcal{L}_8, \mathcal{L}_{11}) \quad \mathcal{F}_4\mathcal{F}_8\mathcal{F}_{11} = D^8 \qquad (\mathcal{L}_7, \mathcal{L}_{10}, \mathcal{L}_{11}) \quad \mathcal{F}_7\mathcal{F}_{10}\mathcal{F}_{11} = D^8$$

but no sets of four or more nontouching loops in $\mathcal{L}$.

With these sets collected, we define the **graph determinant** Δ as

$$\Delta = 1 - \sum_{\mathcal{L}_i} \mathcal{F}_i + \sum_{(\mathcal{L}_i, \mathcal{L}_j)} \mathcal{F}_i\mathcal{F}_j - \sum_{(\mathcal{L}_i, \mathcal{L}_j, \mathcal{L}_k)} \mathcal{F}_i\mathcal{F}_j\mathcal{F}_k + \cdots$$

where the first sum is over all the loops in $\mathcal{L}$, the second sum is over all pairs of nontouching loops in $\mathcal{L}$, the third sum is over all triplets of nontouching loops in $\mathcal{L}$, and so forth.

We also define the **graph cofactor of the forward path** L_i, denoted as Δ_i, which is similar to the graph determinant except that all loops touching L_i are removed from the summations. This can be written as

$$\Delta_i = 1 - \sum_{(L_i,\mathcal{L}_j)} \mathcal{F}_j + \sum_{(L_i,\mathcal{L}_j,\mathcal{L}_k)} \mathcal{F}_j\mathcal{F}_k - \sum_{(L_i,\mathcal{L}_j,\mathcal{L}_k,\mathcal{L}_l)} \mathcal{F}_j\mathcal{F}_k\mathcal{F}_l + \cdots$$

With this notation we finally can express **Mason's rule** for the transfer function through the graph:

$$T(D) = \frac{\sum_l F_l \Delta_l}{\Delta}, \tag{12.23}$$

where the sum is over all forward paths.

For our example, we have

$$\Delta_1 = 1 \qquad \Delta_5 = 1$$

(since there are no loops that do not contain vertices in the forward paths L_1 and L_5),

$$\Delta_3 = \Delta_6 = 1 - \mathcal{F}_{11} = 1 - D^2$$

(since L_3 and L_6 do not cross vertex 7 and so do not touch loop $\mathcal{L}_{11}$),

$$\Delta_2 = 1 - \mathcal{F}_8 = 1 - D^2$$

(since L_2 does not touch $\mathcal{L}_8$ but it does touch all other loops), and

$$\Delta_4 = 1 - (\mathcal{F}_8 + \mathcal{F}_{11}) + \mathcal{F}_8\mathcal{F}_{11} = 1 - 2D^2 + D^4$$

$$\Delta_7 = 1 - (\mathcal{F}_9 + \mathcal{F}_{10} + \mathcal{F}_{11}) + (\mathcal{F}_{10}\mathcal{F}_{11}) = 1 - 2D^2 - D^4 + D^6.$$

The graph determinant is

$$\begin{aligned}\Delta &= 1 - (D^4 + D^2 + D^6 + D^4 + D^4 + D^6 + D^2 + D^2 + D^2 + D^4 + D^2) \\ &\quad + (D^4 + D^8 + D^6 + D^6 + D^8 + D^4 + D^6 + D^4 + D^4 + D^6) - (D^8 + D^8) \\ &= 1 - 5D^2 + 2D^6.\end{aligned}$$

Finally, using (12.23) we obtain

$$T(D) = \frac{2D^6 + D^{10}}{1 - 5D^2 + 2D^6}.$$

12.5.2 Characterizing the Node Error Probability P_e and the Bit Error Rate P_b

We now return to the question of the probability of error for convolutional codes. Let $\mathcal{P}_j$ denote the set of all error paths that diverge from node j of the all-zero path in the trellis then remerge and let $\mathbf{p}_{i,j} \in \mathcal{P}_j$ be one of these paths. Let $\Delta M(\mathbf{p}_{i,j}, 0)$ denote the difference between the metric accumulated along path $\mathbf{p}_{i,j}$ and the all-zero path. An error event at node j occurs due to path $\mathbf{p}_{i,j}$ if $\Delta M(\mathbf{p}_{i,j}, 0) < 0$. Letting $P_e(j)$ denote the probability of an error event at node j, we have

$$P_e(j) \le \Pr\left[\bigcup_i \{\Delta M(\mathbf{p}_{i,j}, 0) \le 0\}\right], \tag{12.24}$$

where the inequality follows since an error might not occur when $\Delta M(\mathbf{p}_{i,j}, 0) = 0$. The paths $\mathbf{p}_{i,j} \in \mathcal{P}$ are not all disjoint since they may share branches, so the events $\{\Delta M(\mathbf{p}_{i,j}, 0) \le 0\}$ are not disjoint. This makes (12.24) very difficult to compute exactly. However, it can be upper bounded by the union bound (see Box 1.1 in chapter 1) as

$$P_e(j) \le \sum_i \Pr(\Delta M(\mathbf{p}_{i,j}, 0) \le 0). \tag{12.25}$$

Each term in this summation is now a *pairwise* event between the paths $\mathbf{p}_{i,j}$ and the all-zero path.

We here develop expressions or bounds on the pairwise events in (12.25) for the case that the channel is memoryless. (For example, we have already seen that for the BSC, the probability P_d developed in (12.19) and (12.20) is the probability of the pairwise events in question.) For a memoryless channel, $\Delta M(\mathbf{p}_{i,j}, 0)$ depends only on those branches for which $\mathbf{p}_{i,j}$ is nonzero. Let d be the Hamming weight of $\mathbf{p}_{i,j}$ and let P_d be the probability of the event that this path has a lower (better) metric than the all-zero path. Let $a(d)$ be the number of paths at a distance d from the all-zero path. The probability of a node error event can now be written as follows:

$$\begin{aligned}P_e(j) &\le \sum_{d=d_{\text{free}}}^{\infty} \Pr(\text{error caused by any of the } a(d) \text{ incorrect paths at distance } d) \\ &= \sum_{d=d_{\text{free}}}^{\infty} a(d) P_d.\end{aligned} \tag{12.26}$$

Any further specification on $P_e(j)$ requires characterization of P_d. We show below that bounds on P_d can be written in the form

$$P_d < Z^d \tag{12.27}$$

for some channel-dependent function Z and develop explicit expressions for Z for the BSC and AWGN channel. For now, we simply express the results in terms of Z. With this bound we can write

$$P_e(j) < \sum_{d=d_{\text{free}}}^{\infty} a(d) Z^d.$$

Recalling that the path enumerator for the encoder is $T(D) = \sum_{d=d_{\text{free}}}^{\infty} a(d) D^d$, we obtain a closed-form expression for the bound:

$$P_e(j) < T(D)|_{D=Z}. \tag{12.28}$$

The bound (12.28) is a bound on the probability of a node error. From this, a bound on the bit error rate can be obtained by enumerating the number of bits in error for each node error. The derivative

$$\frac{\partial}{\partial N} T(D, N)$$

brings exponents of N down as multipliers for each term in the series.

Example 12.20 For the weight enumerator of Example 12.18,

$$T(D, N) = D^5 N + 2D^6 N^2 + 4D^7 N^3 + \cdots,$$

we have

$$\frac{\partial}{\partial N}T(D,N) = (1)D^5 + (2)2D^6N + (3)4D^7N^2 + \cdots$$

so that a node error on the error path of metric 5 contributes one bit of error; a node error on either of the error paths of metric 6 contributes two bits of error, and so forth. □

The average number of bits in error along the branches of the trellis is

$$\left.\frac{\partial}{\partial N}T(D,N)\right|_{N=1,D=Z}.$$

For a rate $R = k/n$ code, each branch corresponds to k message bits, so that

$$P_b < \frac{1}{k}\left.\frac{\partial}{\partial N}T(D,N)\right|_{N=1,D=Z}. \tag{12.29}$$

An approximation on the probability of bit error can be obtained by retaining only the first term of the series in (12.29) and using the bound $P_d < Z^d$. To retain only the first term of the series, write

$$T(D,N) = D^{d_{\text{free}}}(N^{n_1} + N^{n_2} + \cdots) + \cdots.$$

Then

$$\left.\frac{\partial}{\partial N}T(D,N)\right|_{N=1} = D^{d_{\text{free}}}(n_1 + n_2 + \cdots) + \cdots.$$

The number $n_1 + n_2 + \cdots$ is the number of nonzero message bits associated with codewords of weight d_{free}. Let us denote this number as $b_{\text{dfree}} = n_1 + n_2 + \cdots$

Example 12.21 Suppose

$$T(D,N) = D^6N + D^6N^3 + 3D^8N + 5D^8N^4 + \cdots$$

Then there are two codewords of weight 6: one corresponding to a message of weight 1 and one corresponding to a message of weight 3. We could write

$$T(D,N) = D^6(N + N^3) + 3D^8N + 5D^8N^4 + \cdots.$$

Then $b_6 = 1 + 3 = 4$. □

Then the approximation is

$$P_b \approx \frac{1}{k}b_{d_{\text{free}}}D^{d_{\text{free}}}\Big|_{D=Z}. \tag{12.30}$$

A lower bound can be found as

$$P_b > \frac{1}{k}b_{d_{\text{free}}}P_{d_{\text{free}}}, \tag{12.31}$$

where $P_{d_{\text{dfree}}}$ is P_d at $d = d_{\text{free}}$.

Example 12.22 For

$$T(D,N) = \frac{D^5}{1-2DN} = D^5N + 2D^6N^2 + 4D^7N^3 + \cdots$$

the derivative is

$$\frac{\partial}{\partial N}T(D,N) = D^5 + 4D^6N + 12D^7N^2 + \cdots$$

so the probability of error is approximated by

$$P_b \approx Z^5$$

or lower bounded by

$$P_b > P_5.$$

□

12.5.3 A Bound on P_d for Discrete Channels

In this section we develop a bound on P_d for discrete channels such as the BSC [373, Section 12.3.1]. Let $p(r_i|1)$ denote the likelihood of the received signal r_i, given that the symbol corresponding to 1 was sent through the channel; similarly $p(r_i|0)$. Then

$$\begin{aligned} P_d &= P(\Delta M(\mathbf{p}_{i,j}, 0) \le 0 \text{ and } d_H(\mathbf{p}_{i,j}, \mathbf{0}) = d) \\ &= P\Big[\sum_{i=1}^{d}\log P(r_i|1) - \sum_{i=1}^{d}\log P(r_i|0) \ge 0\Big] \end{aligned}$$

where $\{r_1, r_2, \ldots, r_d\}$ are the received signals at the d coordinates where $\mathbf{p}_{i,j}$ is nonzero. Continuing,

$$P_d = P\Big[\sum_{i=1}^{d}\log\frac{p(r_i|1)}{p(r_i|0)} \ge 0\Big] = P\Big[\prod_{i=1}^{d}\frac{p(r_i|1)}{p(r_i|0)} \ge 1\Big].$$

Let R' be the set of vectors of elements $\mathbf{r} = (r_1, r_2, \ldots, r_d)$ such that

$$\prod_{i=1}^{d}\frac{p(r_i|1)}{p(r_i|0)} \ge 1. \tag{12.32}$$

(For example, for $d = 5$ over the BSC, R' is the set of vectors for which 3 or 4 or 5 of the r_i are equal to 1.) The probability of any one of these elements is $\prod_{i=1}^{d} p(r_i|0)$, since we are assuming that all zeros are sent. Thus, the probability of any vector in R' is $\prod_{i=1}^{d} p(r_i|0)$. The probability P_d can be obtained by summing over all the vectors in R':

$$P_d = \sum_{\mathbf{r}\in R'}\prod_{i=1}^{d}p(r_i|0).$$

Since the left-hand side of (12.32) is ≥ 1, we have

$$P_d \le \sum_{\mathbf{r}\in R'}\prod_{i=1}^{d}p(r_i|0)\left[\frac{p(r_i|1)}{p(r_i|0)}\right]^s$$

for any s such that $0 \le s < 1$. The tightest bound is obtained by minimizing with respect to s:

$$P_d \le \min_{0\le s<1}\sum_{\mathbf{r}\in R'}\prod_{i=1}^{d}p(r_i|0)\left[\frac{p(r_i|1)}{p(r_i|0)}\right]^s = \min_{0\le s<1}\sum_{\mathbf{r}\in R'}\prod_{i=1}^{d}p(r_i|0)^{1-s}p(r_i|1)^s.$$

This is made more tractable (and larger) by summing over the set R of *all* sequences $(r_1, r_2, \ldots, r_d)$:

$$P_d < \min_{0\le s<1}\sum_{\mathbf{r}\in R}\prod_{i=1}^{d}p(r_i|0)^{1-s}p(r_i|1)^s.$$

The order of summation and product can be reversed, resulting in

$$P_d < \min_{0\le s<1} \prod_{i=1}^{d} \sum_{r_i} p(r_i|0)^{1-s} p(r_i|1)^s.$$

This is known as the **Chernoff bound**. Let

$$Z = \min_{0\le s<1} \sum_{r_i} p(r_i|0)^{1-s} p(r_i|1)^s. \tag{12.33}$$

Then $P_d < Z^d$.

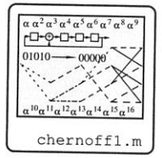
If the channel is symmetric, then by symmetry arguments the minimum must occur when $s = 1/2$. Then $Z = \sum_{r_i} \sqrt{p(r_i|0)p(r_i|1)}$ and

$$P_d < \prod_{i=1}^{d} \sum_{r_i} \sqrt{p(r_i|0)p(r_i|1)}. \tag{12.34}$$

This bound is known as the **Bhattacharya bound**.

Example 12.23 Suppose the encoder with $T(D, N) = \frac{D^5 N}{1-2DN}$ is used in conjunction with an asymmetric channel having the following transition probabilities:

$P(r\|a)$	$a = 0$	$a = 1$
$r = 0$	0.98	0.003
$r = 1$	0.02	0.997.



Then

$$\begin{aligned} Z &= \min_{0\le s<1} \sum_{r_i} p(r_i|0)^{1-s} p(r_i|1)^s \\ &= p(0|0)^{1-s} p(0|1)^s + p(1|0)^{1-s} p(1|1)^s \\ &= (0.98)^{1-s}(0.003)^s + (0.02)^{1-s}(0.997)^s. \end{aligned}$$

The minimum value can be found numerically as $Z = 0.1884$, which occurs when $s = 0.442$.

The node error probability can be bounded using (12.28) as

$$P_e(j) < \left.\frac{D^5 N}{1-2DN}\right|_{N=1, D=0.1884} = 3.8 \times 10^{-4}$$

and the bit error rate is bounded using (12.29) as

$$P_b < \left.\frac{1}{k}\frac{\partial}{\partial N}\frac{D^5 N}{1-2DN}\right|_{N=1, D=0.1884} = 6.1 \times 10^{-4}$$

The approximate bit error rate from (12.30) is

$$P_b \approx \left.\frac{1}{k} b_{d_{\text{free}}} D^{d_{\text{free}}}\right|_{D=0.1884} = 2.4 \times 10^{-4},$$

where $d_{\text{free}} = 5$ and $b_{d_{\text{free}}} = 1$. □

Performance Bound on the BSC

For the BSC with crossover probability p_c, (12.34) can be written as follows:

$$\begin{aligned} P_d &< \prod_{i=1}^{d} \sum_{r_i} \sqrt{p(r_i|0)p(r_i|1)} = \prod_{i=1}^{d} (\sqrt{p(0|0)p(0|1)} + \sqrt{P(1|0)p(1|1)}) \\ &= \prod_{i=1}^{d} 2\sqrt{p_c(1-p_c)} = [4p_c(1-p_c)]^{d/2} = \sqrt{4p_c(1-p_c)}^{d}. \end{aligned}$$

The expression Z in (12.27) is thus $Z = [4p_c(1-p_c)]^{1/2}$.

Let us now return to $P_e(j)$. Inserting this bound on P_d in (12.26) we obtain

$$P_e(j) < \sum_{d=d_{\text{free}}}^{\infty} a(d) P_d < \sum_{d=d_{\text{free}}}^{\infty} a(d) \sqrt{4p_c(1-p_c)}^{d}.$$

The closed-form expression for the bound on the probability of error is

$$P_e(j) < T(D)|_{D=\sqrt{4p_c(1-p_c)}}. \tag{12.35}$$

The bound on the bit error rate of (12.29) is

$$P_b < \left.\frac{1}{k}\frac{\partial}{\partial N} T(D, N)\right|_{N=1, D=\sqrt{4p_c(1-p_c)}}. \tag{12.36}$$

A lower bound can be obtained using (12.31)

$$P_b > \frac{1}{k} b_{d_{\text{free}}} P_{d_{\text{free}}}, \tag{12.37}$$

where $P_{d_{\text{free}}}$ can be computed using (12.19) and (12.20).

12.5.4 A Bound on P_d for BPSK Signaling Over the AWGN Channel

Suppose that the coded bits $c_t^{(i)}$ are mapped to a BPSK signal constellation by $a_t^{(i)} = \sqrt{E_c}(2c_t^{(i)} - 1)$, where E_c is the coded signal energy, with $E_c = RE_b$, and E_b is the energy per message bit. If the all-zero sequence is sent, then the sequence of amplitudes $(-\sqrt{E_c}, -\sqrt{E_c}, -\sqrt{E_c}, \ldots)$ is sent. A sequence which deviates from this path in d locations is at a squared distance $2dE_c$ from it. Then P_d is the probability that a d-symbol sequence is decoded incorrectly, compared to the sequence for all-zero transmission. That is, it is the problem of distinguishing $\mathbf{p}_1 = (-\sqrt{E_c}, -\sqrt{E_c}, -\sqrt{E_c}, \ldots, -\sqrt{E_c})$ from $\mathbf{p}_2 = (\sqrt{E_c}, \sqrt{E_c}, \sqrt{E_c}, \ldots, \sqrt{E_c})$, where these vectors each have d elements. The Euclidean distance between these two points is

$$d_{\text{Euclidean}}(\mathbf{p}_1, \mathbf{p}_2) = 2[dE_c]^{1/2}.$$

The probability of a detection error is (see Section 1.5.4)

$$P_d = Q\left(\sqrt{\frac{2dE_c}{N_0}}\right).$$

To express this in the form Z^d (for use in the bound (12.28)) use the bound $Q(x) < \frac{1}{2}e^{-x^2/2}$ (see Exercise 1).12. We thus obtain

$$P_d < \frac{1}{2} e^{-dE_c/N_0}.$$

Then (12.28) and (12.29) give

$$P_e(j) < \frac{1}{2}T(D)\Big|_{D=e^{-E_c/N_0}} \qquad P_b < \frac{1}{2}\frac{1}{k}\frac{\partial}{\partial N}T(D,N)\Big|_{N=1,D=e^{-E_c/N_0}}. \tag{12.38}$$

Another bound on the Q function is [359]

$$Q(\sqrt{x+y}) \le Q(\sqrt{x})e^{-y/2}, \qquad x \ge 0, y \ge 0. \tag{12.39}$$

Then

$$P_d = Q\left(\sqrt{\frac{2dE_c}{N_0}}\right) = Q\left(\sqrt{\frac{2d_{\text{free}}E_c}{N_0} + \frac{2(d-d_{\text{free}})E_c}{N_0}}\right)$$
$$\le Q\left(\sqrt{\frac{2d_{\text{free}}E_c}{N_0}}\right)e^{d_{\text{free}}E_c/N_0}e^{-dE_c/N_0} = Q\left(\sqrt{\frac{2d_{\text{free}}E_c}{N_0}}\right)e^{d_{\text{free}}E_c/N_0}(e^{-E_c/N_0})^d.$$

Then

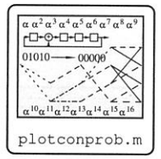
$$P_e(j) \le e^{d_{\text{free}}E_c/N_0}Q\left(\sqrt{\frac{2d_{\text{free}}E_c}{N_0}}\right)T(D)\Big|_{D=e^{-E_c/N_0}}$$

and

$$P_b \le \frac{1}{k}e^{d_{\text{free}}E_c/N_0}Q\left(\sqrt{\frac{2d_{\text{free}}E_c}{N_0}}\right)\frac{\partial}{\partial N}T(D,N)\Big|_{N=1,D=e^{-E_c/N_0}}. \tag{12.40}$$

A lower bound can be obtained using (12.31)

$$P_b > \frac{1}{k}b_{d_{\text{free}}}Q\left(\sqrt{\frac{2d_{\text{free}}E_c}{N_0}}\right). \tag{12.41}$$



Example 12.24 For the $R = 1/2$ code of Example 12.1 with $d_{\text{free}} = 5$, Figure 12.27 shows the bounds on the probability of bit error of the code for both hard- and soft-decision decoders compared with uncoded performance. For soft decoding, the lower bound and the upper bound of (12.40) approach each other for high signal to noise ratios, so the bounds are asymptotically tight. (The bound of (12.38) is looser.) Gains of approximately 4 dB at high SNR are evident for soft decision decoding.

The hard-decision decoding bounds are clearly not as tight. Also, there is approximately 3 dB less coding gain for the hard-decision decoder. □

12.5.5 Asymptotic Coding Gain

The lower bound for the probability of bit error for the coded signal (using soft-decision decoding)

$$P_b > \frac{1}{k}a(d_{d_{\text{free}}})n_{d_{\text{free}}}Q\left(\sqrt{\frac{2d_{\text{free}}E_c}{N_0}}\right)$$

can be approximated using the bound $Q(x) < \frac{1}{2}e^{-x^2/2}$ as

$$P_b \approx \frac{1}{k}\frac{1}{2}a(d_{d_{\text{free}}})n_{d_{\text{free}}}e^{-d_{\text{free}}E_c/N_0}. \tag{12.42}$$

The probability of bit error for uncoded transmission is

$$P_b = Q\left(\sqrt{\frac{2E_b}{N_0}}\right) < \frac{1}{2}e^{-E_b/N_0}. \tag{12.43}$$

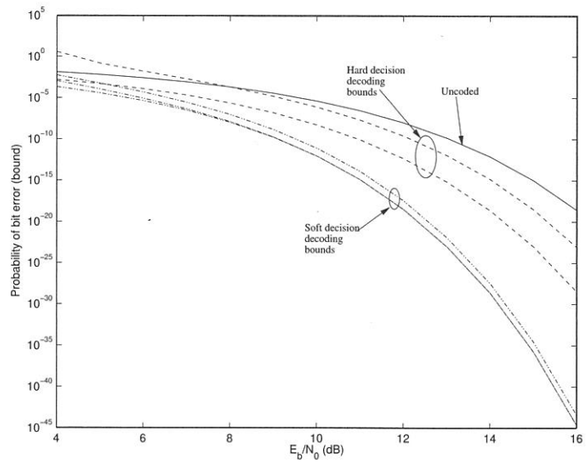


Figure 12.27: Performance of a (3, 1) convolutional code with $d_{\text{free}} = 5$.

The dominant factor in (12.42) and (12.43) for large values of signal-to-noise ratio is determined by the exponents. Comparing the exponents in these two using $E_c = RE_b$ we see that the exponent in the probability of bit error for the coded case is a factor of Rd_{free} larger than the exponent for the uncoded case. The quantity

$$\gamma_{\text{soft}} = 10\log_{10} Rd_{\text{free}}$$

is called the *asymptotic coding gain* of the code. For sufficiently large SNR, performance essentially equivalent to uncoded performance can be obtained with γ dB less SNR when coding is employed. A similar argument can be made to show that the asymptotic coding gain for hard decision decoding is

$$\gamma_{\text{hard}} = 10\log_{10}\frac{Rd_{\text{free}}}{2}.$$

This shows that asymptotically, soft-decision decoding is 3 dB better than hard-decision decoding.

As the SNR increases, the dominant term in computing the bit error rate is the first term in $T(x, N)$. As a result the free distance has a very strong bearing on the performance of the code.

12.6 Tables of Good Codes

Unlike block codes, where many good codes have been found by exploiting the algebraic structure of the codes, good convolutional codes have been found mostly by computer search. As a result, good codes are known only for relatively short constraint lengths. The following tables [251, 254, 197, 65] provide the best known polynomial codes. It may be observed that all of these codes are nonsystematic.

There are separate tables for different rates. Within each table, different memory lengths L, are used, where

$$L = \nu + 1,$$

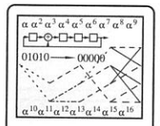
where $\nu = \max_{i,j} \deg(g_{ij}(x))$ is the degree of the highest polynomial. (This quantity is called in many sources the constraint length.) For the rate k/n codes with $k > 1$, L represents the largest degree and ν represents the total memory.

In these tables, the coefficients are represented using octal digits with the least significant bit on the right. Thus, $0 \to 000$, $1 \to 001$, $2 \to 010$, $3 \to 011$, and so forth. There may be trailing zeros on the right. For example, for the rate 1/4 code, the entry with $L = 5$ has generators $g_0 = 52$, $g_1 = 56$, $g_2 = 66$ and $g_3 = 76$. The corresponding bit values are

$$\mathbf{g}_0 = (101\,010) \quad \mathbf{g}_1 = (101\,110) \quad \mathbf{g}_2 = (110\,110) \quad \mathbf{g}_3 = (111\,110).$$

The first coefficient (on the left) is the first coefficient in the encoder.[5] Thus the coded output streams are

$$c_t^{(1)} = m_t + m_{t-2} + m_{t-4} \qquad c_t^{(2)} = m_t + m_{t-2} + m_{t-3} + m_{t-4}$$
$$c_t^{(3)} = m_t + m_{t-1} + m_{t-3} + m_{t-4} \quad c_t^{(4)} = m_t + m_{t-1} + m_{t-2} + m_{t-3} + m_{t-4}$$



finddfree.cc

The program `finddfree` finds d_{free} for a given set of connection coefficients. It has been used to check these results. (Currently implemented only for $k = 1$ codes.)

$R = 1/2$ [251, 197]			
L	$g^{(1)}$	$g^{(2)}$	d_{free}
3	5	7	5
4	64	74	6
5	46	72	7
6	65	57	8
7	554	744	10
8	712	476	10
9	561	753	12
10	4734	6624	12
11	4762	7542	14
12	4335	5723	15
13	42554	77304	16
14	43572	56246	16
15	56721	61713	18
16	447254	627324	19
17	716502	514576	20

$R = 1/3$ [251, 197]				
L	$g^{(1)}$	$g^{(2)}$	$g^{(3)}$	d_{free}
3	5	7	7	8
4	54	64	74	10
5	52	66	76	12
6	47	53	75	13
7	554	624	764	15
8	452	662	756	16
9	557	663	711	18
10	4474	5724	7154	20
11	4726	5562	6372	22
12	4767	5723	6265	24
13	42554	43364	77304	24
14	43512	73542	76266	26

[5] To use the class `BinConvFIR`, the left bit must be interpreted as the LSB of a binary number. The function `octconv` returns an integer value that can be used directly in `BinConvFIR`.

$R = 1/4$ [251, 197]					
L	$g^{(1)}$	$g^{(2)}$	$g^{(3)}$	$g^{(4)}$	d_{free}
3	5	7	7	7	10
4	54	64	64	74	13
5	52	56	66	76	16
6	53	67	71	75	18
7	564	564	634	714	20
8	472	572	626	736	22
9	463	535	733	745	24
10	4474	5724	7154	7254	27
11	4656	4726	5562	6372	29
12	4767	5723	6265	7455	32
13	44624	52374	66754	73534	33
14	42226	46372	73256	73276	36

$R = 2/3$ [254, 172]					
		$g^{(1,1)}$	$g^{(1,2)}$	$g^{(1,3)}$	
L	ν	$g^{(2,1)}$	$g^{(2,2)}$	$g^{(2,3)}$	d_{free}
2	2	6	2	6	3
		2	4	8	
3	3	5	2	6	4
		1	4	7	
3	4	7	1	4	5
		2	5	7	
4	5	60	30	70	6
		14	40	74	
4	6	64	30	64	7
		30	64	74	
5	7	60	34	54	8
		16	46	74	
5	8	64	12	52	8
		26	66	44	
6	9	52	06	74	9
		05	70	53	
6	10	63	15	46	10
		32	65	61	

$R = 3/4$ [254, 172]						
		$g^{(1,1)}$	$g^{(1,2)}$	$g^{(1,3)}$	$g^{(1,4)}$	
		$g^{(2,1)}$	$g^{(2,2)}$	$g^{(2,3)}$	$g^{(2,4)}$	
L	ν	$g^{(3,1)}$	$g^{(3,2)}$	$g^{(3,3)}$	$g^{(3,4)}$	d_{free}
2	3	4	4	4	4	4
		0	6	2	4	
		0	2	5	5	
3	5	6	2	2	6	5
		1	6	0	7	
		0	2	5	5	
3	6	6	1	0	7	6
		3	4	1	6	
		2	3	7	4	
4	8	70	30	20	40	7
		14	50	00	54	
		04	10	74	40	
4	9	40	14	34	60	8
		04	64	20	70	
		34	00	60	64	

Table 12.2 presents a comparison of d_{free} for systematic and nonsystematic codes (with polynomial generators), showing that nonsystematic codes have generally better distance properties. Results are even more pronounced for longer constraint lengths.

12.7 Puncturing

In Section 3.9, puncturing was introduced as a modification to block codes, in which one of the parity symbols is removed. In the context of convolutional codes, **puncturing** is accomplished by periodically removing bits from one or more of the encoder output streams [40]. This has the effect of increasing the rate of the code.

Example 12.25 Let the coded output sequence of a rate $R = 1/2$ code be

$$\mathbf{c} = (c_0^{(1)}, c_0^{(2)}, c_1^{(1)}, c_1^{(2)}, c_2^{(1)}, c_2^{(2)}, c_3^{(1)}, c_3^{(2)}, c_4^{(1)}, c_4^{(2)}, \ldots)$$

Table 12.2: Comparison of Free Distance as a Function of Constraint Length for Systematic and Nonsystematic Codes

$R = 1/3$ [251]		
	Systematic	Nonsystematic
L	d_{free}	d_{free}
2	3	3
3	4	5
4	4	6
5	5	7
6	6	8
7	6	8
8	7	10

$R = 1/2$ [251]		
	Systematic	Nonsystematic
L	d_{free}	d_{free}
2	5	5
3	6	8
4	8	10
5	9	12
6	10	13
7	12	15
8	12	16

When the code is punctured by removing every fourth coded symbol, the punctured sequence is

$$\tilde{\mathbf{c}} = (c_0^{(1)}, c_0^{(2)}, c_1^{(1)}, -, c_2^{(1)}, c_2^{(2)}, c_3^{(1)}, -, c_4^{(1)}, c_4^{(2)}, \ldots).$$

The $-$ symbols merely indicate where the puncturing takes place; they are not transmitted. The punctured sequence thus produces three coded symbols for every two input symbols, resulting in a rate $R = 2/3$ code. □

Decoding of a punctured code can be accomplished using the same trellis as the unpunctured code, but simply not accumulating any branch metric for the punctured symbols. One way this can be accomplished is by inserting symbols into the received symbol stream whose branch metric computation would be 0, then using conventional decoding.

The pattern of puncturing is often described by means of a **puncturing matrix** P. For a rate k/n code, the puncture matrix has n rows. The number of columns is the number of symbols over which the puncture pattern repeats. For example, for the puncturing of the previous example,

$$P = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}.$$

The element P_{ij} is 1 if the ith symbol is sent in the jth epoch of the puncturing period.

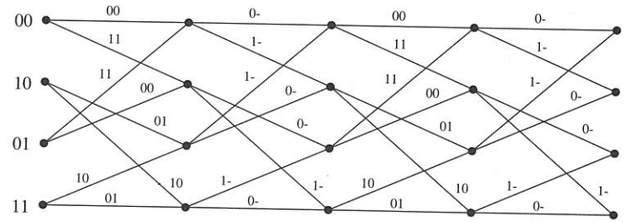
While the punctured code can be encoded as initially described — by encoding with the lower rate code then puncturing — this is wasteful, since computations are made which are promptly ignored. However, since the code obtained is still a convolutional code, it has its own trellis, which does not require any explicit puncturing.

Example 12.26 We demonstrate puncturing for the code which has been a *leitmotif* for this chapter, with generators $g^{(1)}(x) = 1 + x^2$ and $g^{(2)}(x) = 1 + x + x^2$. Puncturing is accomplished by deleting every other bit of the second output stream (as above). Four stages of the trellis for this punctured code are shown in Figure 12.28(a).

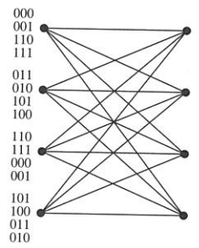
Now draw the trellis for the resulting $R = 2/3$ code by taking the input bits two at a time, or two stages in the original trellis, and think of this as representing a *single* transition of the new code. The resulting trellis is shown in Figure 12.28(b).

□

Besides being used to increase the rate of the code, puncturing can sometimes be used to reduce decoding complexity. In decoding, each state must be extended to 2^k states at the



(a) Trellis for initial punctured code.



(b) Trellis by collapsing two stages of the initial trellis into a single stage.

Figure 12.28: Trellises for a punctured code.

next time. Thus, the decoding complexity scales exponentially with k. Given the trellis for an unpunctured code with rate $R = k/n$ with $k > 1$, if a trellis for an equivalent punctured code having $k' < k$ input bits can be found, then the decoding complexity can be decreased.

Suppose, for example, that the encoder having the trellis in Figure 12.28(b) is used. In decoding, four successor states must be examined for each state, so that the best of four paths to a state must be selected. However, we know that this encoder also has the trellis representation in Figure 12.28(a). Decoding on this trellis only has two successor states for each state. This results in only a two-way comparison, which can be done using a conventional add/compare/select circuit.

Of course, puncturing changes the distance properties of the code: a good rate $R = 2/3$ code is not necessarily obtained by puncturing a good $R = 1/2$ code. Tables of the best $R = 3/4$ and $R = 2/3$ codes obtainable by puncturing are presented in [40].

12.7.1 Puncturing to Achieve Variable Rate

Puncturing can also be used to generate codes of various rates using the same encoder. Such flexibility might be used, for example, to match the code to the channel in a situation in which the channel characteristics might change. Suppose that a rate $R = 1/2$ encoder is used as the "basic" code. As mentioned above, puncturing 1 bit out of every 4 results in a

$R = 2/3$ code. Puncturing 3 out of every 8 bits results in a $R = 4/5$ code.

If the puncturing is done in such a way that bits punctured to obtain the $R = 2/3$ code are included among those punctured to obtain the $R = 4/5$ code, then the $R = 4/5$ codewords are embedded in the $R = 2/3$ codewords. These codewords are, in turn, embedded in the original $R = 1/2$ codewords. Such codes are said to be **rate compatible punctured convolutional** (RCPC) codes. Assuming that all the RCPC codes have the same period (the same width of the P matrix), then the P matrix of a higher rate code is obtained simply by changing one or more of the 1s to 0s. An RCPC code system can be designed so that the encoder and the decoder have the same structure for all the different rates. Extensive tables of codes and puncturing schedules which produce rate compatible codes appear in [130]. Abbreviated tables appear in Table 12.3.

Table 12.3: Best Known $R = 2/3$ and $R = 3/4$ Convolutional Codes Obtained by Puncturing a $R = 1/2$ Code [198]

Initial Code			Punctured Code			Initial Code			Punctured Code		
ν	$g^{(0)}$	$g^{(1)}$	P	d_{free}	$N_{d_{\text{free}}}$	ν	$g^{(0)}$	$g^{(1)}$	P	d_{free}	$N_{d_{\text{free}}}$
2	5	7	$\begin{bmatrix}1&0\\1&1\end{bmatrix}$	3	1	2	5	7	$\begin{bmatrix}1&0&1\\1&1&0\end{bmatrix}$	3	6
3	13	17	$\begin{bmatrix}1&1\\0&1\end{bmatrix}$	4	3	3	13	17	$\begin{bmatrix}1&1&0\\1&0&1\end{bmatrix}$	4	29
4	31	27	$\begin{bmatrix}1&1\\0&1\end{bmatrix}$	4	1	4	31	27	$\begin{bmatrix}1&0&1\\1&1&0\end{bmatrix}$	4	1
5	65	57	$\begin{bmatrix}1&0\\1&1\end{bmatrix}$	6	19	5	65	57	$\begin{bmatrix}1&0&0\\1&1&1\end{bmatrix}$	4	1
6	155	117	$\begin{bmatrix}1&1\\1&0\end{bmatrix}$	6	1	6	155	117	$\begin{bmatrix}1&1&0\\1&0&1\end{bmatrix}$	5	8

12.8 Suboptimal Decoding Algorithms for Convolutional Codes

While the Viterbi algorithm is an optimal decoding algorithm, its complexity grows as 2^{ν}, exponentially with the number of states. The probability bound presented in (1.49) suggests that better performance is obtained by codes with longer memory (constraint length). These two facts conflict: it may not be possible to build a decoder with a sufficiently long memory to achieve some desired level of performance.

The Viterbi algorithm also has fixed decoding costs, regardless of the amount of noise. It would be desirable to have an algorithm which is able to perform fewer computations when there is less noise, adjusting the amount of effort required to decode to the severity of the need.

In this section we present two algorithms which address these problems. These algorithms have decoding complexity which is essentially *constant* as a function of constraint length. Furthermore, the less noisy the channel, the less work the decoders have to do, on average. This makes them typically very fast decoders. These positive attributes are obtained, however, at some price. These are *suboptimal* decoding algorithms: they do not always provide the maximum-likelihood decision. Furthermore, the decoding time and decoder memory required are a random variables, depending on the particular received sequence.

In recent years, the availability of high-speed hardware has led to almost universal use of

Viterbi decoding. However, there are still occasions where very long constraint lengths may be of interest, so these algorithms are still of value. Viterbi algorithms can be practically used on codes with constraint lengths up to about 10, while the sequential algorithms discussed here can be used with constraint lengths up to 50 or more.

The first algorithm is known as the **stack algorithm**, or the ZJ algorithm, after Zigangirov (1966) [388] and Jelinek (1969) [165]. The second algorithm presented is the **Fano algorithm** (1963) [80]. These are both instances of **sequential decoding** algorithms [378].

12.8.1 Tree Representations

While the Viterbi algorithm is based on a trellis representation of the code, the sequential algorithms are best understood using a tree representation. Figure 12.29 shows the tree for the convolutional code with generator $G(x) = \left[1 + x^2, 1 + x + x^2\right]$ whose state diagram and trellis are shown in Figure 12.5. At each instant of time, the input bit selects either the upper branch (input bit = 0) or the lower branch (input bit = 1). The output bits for the code are shown along the branches of the tree. By recognizing common states, it is possible to "fold" the tree back into a trellis diagram.

The tree shown in figure 12.29 is for an input sequence of length 4 in the "branching portion" of the tree, followed by a sequence of zeros which drives the tree back to the all-zero state in the "nonbranching" portion of the tree. The length of the codeword is L branches. Each path of length L from the root node to a leaf node of the tree corresponds to a unique convolutional codeword.

Since the size of the tree grows exponentially with the code length, it is not feasible to search the whole tree. Instead, a partial search of the tree is done, searching those portions of the tree that appear to have the best possibility of succeeding. The sequential decoding algorithms which perform this partial search can be described heuristically as follows: Start at the root node and follow the branches that appear to best match the noisy received data. If, after some decisions, the received word and the branch labels are not matching well, back up and try a different route.

12.8.2 The Fano Metric

As a general rule, paths of differing lengths are compared as the algorithm moves around the tree these in sequential decoding algorithms. A path of length five branches through the tree might be compared with a path of length twenty branches. A first step, therefore, is to determine an appropriate metric for comparing different paths. The log likelihood function used as the branch metric for the Viterbi algorithm is *not* appropriate to use for the sequential algorithm. This is because log likelihood functions are biased against long paths.

Example 12.27 Suppose the transmitted sequence of a rate $R = 1/2$ code is

$$\mathbf{a} = [11, 10, 10, 11, 11, 01, 00, 01]$$

and the received sequence is

$$\mathbf{r} = [01, 10, 00, 11, 11, 01, 00, 01].$$

Using the Hamming distance as the path metric, this is to be compared with a partial path of one branch, $\tilde{\mathbf{a}}^{(1)} = [00]$ and a partial path of six branches, $\mathbf{a}^{(2)} = [11, 10, 10, 11, 11, 01]$. Letting $[\mathbf{r}]_i$ denote i branches of received data, we have

$$d_H([\mathbf{r}]_1, \mathbf{a}^{(1)}) = 1 \qquad d_H([\mathbf{r}]_6, \mathbf{a}^{(2)}) = 2.$$

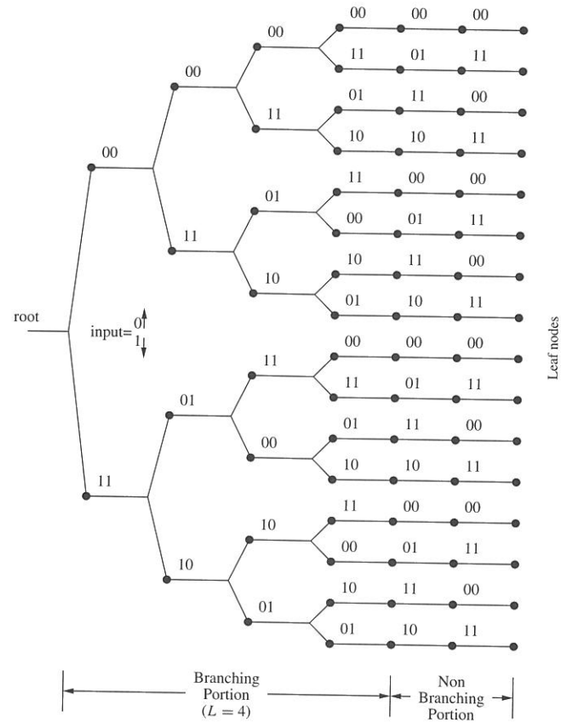


Figure 12.29: A tree representation for a rate $R = 1/2$ code.

By not taking into account the fact that branches of different length are being compared, $\mathbf{a}_1$ appears to be better, since the Hamming distance is smaller. But intuitively, it seems that 2 errors out of 12 bits should be superior to 1 error out of 2 bits. □

The Fano metric is designed to take into account paths of different lengths. Let

$$\tilde{\mathbf{a}}^{(i)} = (\mathbf{a}_0^{(i)}, \mathbf{a}_1^{(i)}, \ldots, \mathbf{a}_{n_i-1}^{(i)})$$

be a partial input sequence of length n_i corresponding to a particular path through the tree, where each $\mathbf{a}_j^{(i)}$ consists of n bit symbols. Accordingly, let us write this as a vector of nn_i bits,

$$\tilde{\mathbf{a}}^{(i)} = (\mathbf{a}_0^{(i)}, \mathbf{a}_1^{(i)}, \ldots, \mathbf{a}_{n_i-1}^{(i)}) = (a_0^{(i)}, a_1^{(i)}, \ldots, a_{n_i n-1}^{(i)}).$$

Assuming that each encoded bit occurs with equal probability, each sequence $\tilde{\mathbf{a}}^{(i)}$ occurs with probability

$$P(\tilde{\mathbf{a}}^{(i)}) = (2^{-k})^{n_i} = 2^{-Rnn_i}. \tag{12.44}$$

Suppose that there are M partial sequences to be compared, represented as elements of the set $\mathcal{X}$,

$$\mathcal{X} = \{\tilde{\mathbf{a}}^{(1)}, \tilde{\mathbf{a}}^{(2)}, \ldots, \tilde{\mathbf{a}}^{(M)}\}.$$

Let $n_{\max}$ be the longest of the partial sequences,

$$n_{\max} = \max(n_1, n_2, \ldots, n_M).$$

Let $\mathbf{r} = (\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{L-1})$ be a received sequence corresponding to a codeword and let

$$\tilde{\mathbf{r}} = (\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{n_{\max}-1})$$

be a "partial" sequence, starting at the beginning of $\mathbf{r}$, but extending only through $n_{\max}$ branches. (The tilde is used to represent partial sequences.) Each $\mathbf{r}_i$ consists of n symbols, so we can also write this as a vector of $nn_{\max}$ elements

$$\tilde{\mathbf{r}} = (r_0, r_1, \ldots, r_{nn_{\max}-1}).$$

From among the sequences in $\mathcal{X}$, the optimal receiver chooses the $\tilde{\mathbf{a}}^{(i)}$ which maximizes

$$P(\tilde{\mathbf{a}}^{(i)}|\tilde{\mathbf{r}}) = \frac{P[\tilde{\mathbf{a}}^{(i)}]p(\tilde{\mathbf{r}}|\tilde{\mathbf{a}}^{(i)})}{p(\tilde{\mathbf{r}})}.$$

Assuming (as is typical) that the channel is memoryless, this can be written as

$$P(\tilde{\mathbf{a}}^{(i)}|\tilde{\mathbf{r}}) = \frac{P[\tilde{\mathbf{a}}^{(i)}]\prod_{j=0}^{n_i-1} p(\mathbf{r}_j|\mathbf{a}_j^{(i)}) \prod_{j=n_i}^{n_{\max}-1} p(\mathbf{r}_j)}{p(\tilde{\mathbf{r}})}, \tag{12.45}$$

where the second product arises since there are no known data associated with the sequence $\tilde{\mathbf{a}}^{(i)}$ for $j \geq n_i$. Canceling common terms in the numerator and denominator of (12.45) we obtain

$$P(\tilde{\mathbf{a}}^{(i)}|\tilde{\mathbf{r}}) = P[\tilde{\mathbf{a}}^{(i)}] \prod_{j=0}^{n_i-1} \frac{p(\mathbf{r}_j|a_j^{(i)})}{p(\mathbf{r}_j)}.$$

Taking the logarithm of both sides and using (12.44), we have

$$\log_2 P(\tilde{\mathbf{a}}^{(i)}|\tilde{\mathbf{r}}) = \sum_{j=0}^{n_i-1} \left[p(\mathbf{r}_j|\mathbf{a}_j^{(i)}) - p(\mathbf{r}_j)\right] - \log_2 Rnn_i.$$

Each $\mathbf{r}_j$ and $\mathbf{a}_j^{(i)}$ consists of n symbols, so we can write this as

$$\log_2 P(\tilde{\mathbf{a}}^{(i)}|\tilde{\mathbf{r}}) = \sum_{j=0}^{nn_i-1} \left[p(r_j|a_j^{(i)}) - p(r_j)\right] - \log_2 Rnn_i.$$

We use this as the *path metric* and denote it as

$$M(\tilde{\mathbf{a}}^{(i)}, \mathbf{r}) = \log_2 P(\tilde{\mathbf{a}}^{(i)}|\tilde{\mathbf{r}}).$$

The corresponding *branch metric* which is accumulated for each new symbol is

$$\boxed{\mu(r_j, a_j^{(i)}) = \underbrace{\log_2 P(r_j|a_j^{(i)})}_{\text{ML metric}} \underbrace{- \log_2 P(r_j) - R}_{\text{path length bias}}.}$$

This metric is called the **Fano metric**. As indicated, the Fano metric consists of two parts. The first part is the same maximum likelihood metric used for conventional Viterbi decoding. The second part consists of a bias term which accounts for different path lengths. Thus, when

comparing paths of different lengths, the path with the largest Fano metric is considered the best path, most likely to be part of the maximum likelihood path. If all paths are of the same length, then the path length bias becomes the same for all paths and may be neglected.

If transmission takes place over a BSC with transition probability p_c, then $P(r_j = 0) = P(r_j = 1) = \frac{1}{2}$. The branch length bias is

$$\log_2 \frac{1}{P(r_j)} - R = \log_2 2 - R = 1 - R,$$

which is > 0 for all codes of rate $R < 1$. The cumulative path length bias for a path of nn_i bits is $nn_i(1 - R)$: the path length bias increases linearly with the path length. For the BSC, the branch metric is

$$\mu(r_j, a_j) = \begin{cases} \log_2(1 - p_c) - \log_2 \frac{1}{2} - R = \log_2 2(1 - p_c) - R & \text{if } r_j = a_j \\ \log_2 p_c - \log_2 \frac{1}{2} - R = \log_2 2p_c - R & \text{if } r_j \neq a_j. \end{cases} \quad (12.46)$$

Example 12.28 Let us contrast the Fano metric with the ML metric for the data in Example 12.27, assuming that $p_c = 0.1$. Using the Fano metric, we have

$$M([00], \mathbf{r}) = \log_2(1 - p_c) + \log_2 p_c + 2(1 - 1/2) = -2.479$$

$$M([11, 10, 10, 11, 11, 01], \mathbf{r}) = 10\log_2(1 - p_c) + 2\log_2 p_c + 12(1 - 1/2) = -2.164.$$

Thus the longer path is has a better (higher) Fano metric than the shorter path. □

Example 12.29 Suppose that $R = 1/2$ and $p_c = 0.1$. Then from (12.46),

$$\mu(r_j, a_j) = \begin{cases} 0.348 & r_j = a_j \\ -2.82 & r_j \neq a_j. \end{cases}$$

It is common to scale the metrics by a constant so that they can be closely approximated by integers. Scaling the metric by $1/0.348$ results in the metric

$$\mu(r_j, a_j) = \begin{cases} 1 & r_j = a_j \\ -8 & r_j \neq a_j. \end{cases}$$

Thus, each bit a_j that agrees with r_j results in a $+1$ added to the metric. Each bit a_j that disagrees with r_j results in -8 added to the metric. □

A path with only a few errors (the correct path) tends to have a slowly increasing metric, while an incorrect path tends to have a rapidly decreasing metric. Because the metric decreases so rapidly, incorrect paths are not extended far before being effectively rejected.

For BPSK transmission through an AWGN , the branch metric is

$$\mu(r_j, a_j) = \log_2 p(r_j|a_j) - \log_2 p(r_j) - R,$$

where $p(r_j|a_j)$ is the PDF of a Gaussian r.v. with mean a_j and variance $\sigma_2 = N_0/2$ and

$$p(r_j) = \frac{p(r_j|a_j = 1) + p(r_j|a_j = -1)}{2}.$$

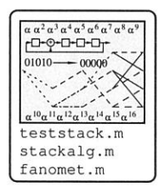
12.8.3 The Stack Algorithm

Let $\tilde{\mathbf{a}}^{(i)}$ represent a path through the tree and let $M(\tilde{\mathbf{a}}^{(i)}, \mathbf{r})$ represent the Fano metric between $\tilde{\mathbf{a}}^{(i)}$ and the received sequence $\mathbf{r}$. These are stored together as a pair $(M(\tilde{\mathbf{a}}^{(i)}, \mathbf{r}), \tilde{\mathbf{a}}^{(i)})$ called a stack entry.

In the stack algorithm, an ordered list of stack entries is maintained which represents all the partial paths which have been examined so far. The list is ordered with the path with the largest (best) metric on top, with decreasing metrics beneath. Each decoding step consists of pulling the top stack entry off the stack, computing the 2^k successor paths and their path metrics to that partial path, then rearranging the stack in order of decreasing metrics. When the top partial path consists of a path from the root node to a leaf node of the tree, then the algorithm is finished.

Algorithm 12.2 The Stack Algorithm

1 **Input:** A sequence $\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{L-1}$
2 **Output:** The sequence $\mathbf{a}_0, \mathbf{a}_1, \ldots, \mathbf{a}_{L-1}$
3 **Initialize:** Load the stack with the empty path with Fano path metric 0: $S = (\emptyset, 0)$.
4 Compute the metrics of the successors of the top path in the stack
5 Delete the top path from the stack
6 Insert the paths computed in step 4 into the stack, and
  rearrange the stack in order of decreasing metric values.
7 If the top path in the stack terminates at a leaf node of the tree, Stop.
  Otherwise, goto step 4.



Example 12.30 The encoder and received data of Example 12.13 are used in the stack algorithm. We have

$$\mathbf{r} = [11\ 10\ \underline{00}\ \underline{10}\ 11\ 01\ 00\ 01\ \ldots].$$

Figure 12.30 shows the contents of the stack as the algorithm progresses. After 14 steps of the algorithm, the algorithm terminates with the correct input sequence on top. (The metrics here are not scaled to integers.) □

A major part of the expense of the stack algorithm is the need to sort the metrics at every iteration of the algorithm. A variation on this algorithm due to Jelinek [165] known as the *stack bucket algorithm* avoids some of this complication. In the stack bucket algorithm, the range of possible metric values (e.g., for the data in Figure 12.30, the range is from 0.7 to -9.2) is partitioned into fixed intervals, where each interval is allocated a certain number of storage locations called a *bucket*. When a path is extended, it is deleted from its bucket and a new path is inserted as the top item in the bucket containing the metric interval for the new metric. Paths within buckets are *not* reordered. The top path in the nonempty bucket with the highest metric interval is chosen as the path to be extended. Instead of sorting, it only becomes necessary to determine which bucket new paths should be placed in. Unfortunately, the bucket approach does not always choose the best path, but only a "very good" path, to extend. Nevertheless, if there are enough buckets that the quantization into metric intervals is not too coarse, and if the received signal is not too noisy, then the top bucket contains only the best path. Any degradation from optimal is minor.

Another practical problem is that the size of the stack must necessarily be limited. For long codewords, there is always the probability that the stack fills up before the correct

Step 1	
0.7	[1]
−5.6	[0]

Step 2	
1.4	[11]
−4.9	[10]
−5.6	[0]

Step 3	
−1.1	[111]
−1.1	[110]
−4.9	[10]
−5.6	[0]

Step 4	
−0.39	[1110]
−1.1	[110]
−4.9	[10]
−5.6	[0]
−6.7	[1111]

Step 5	
0.31	[11100]
−1.1	[110]
−4.9	[10]
−5.6	[0]
−6	[11101]
−6.7	[1111]

Step 6	
−1.1	[110]
−2.2	[111001]
−2.2	[111000]
−4.9	[10]
−5.6	[0]
−6	[11101]
−6.7	[1111]

Step 7	
−2.2	[111000]
−2.2	[111001]
−3.6	[1101]
−3.6	[1100]
−4.9	[10]
−5.6	[0]
−6	[11101]
−6.7	[1111]

Step 8	
−1.5	[1110000]
−2.2	[111001]
−3.6	[1100]
−3.6	[1101]
−4.9	[10]
−5.6	[0]
−6	[11101]
−6.7	[1111]
−7.8	[1110001]

Step 9	
−2.2	[111001]
−3.6	[1101]
−3.6	[1100]
−3.9	[11100001]
−3.9	[11100000]
−4.9	[10]
−5.6	[0]
−6	[11101]
−6.7	[1111]
−7.8	[1110001]

Step 10	
−3.6	[1100]
−3.6	[1101]
−3.9	[11100000]
−3.9	[11100001]
−4.6	[1110011]
−4.6	[1110010]
−4.9	[10]
−5.6	[0]
−6	[11101]
−6.7	[1111]
−7.8	[1110001]

Step 11	
−2.9	[11001]
−3.6	[1101]
−3.9	[11100001]
−3.9	[11100000]
−4.6	[1110010]
−4.6	[1110011]
−4.9	[10]
−5.6	[0]
−6	[11101]
−6.7	[1111]
−7.8	[1110001]
−9.2	[11000]

Step 12	
−2.2	[110010]
−3.6	[1101]
−3.9	[11100000]
−3.9	[11100001]
−4.6	[1110011]
−4.6	[1110010]
−4.9	[10]
−5.6	[0]
−6	[11101]
−6.7	[1111]
−7.8	[1110001]
−8.5	[110011]
−9.2	[11000]

Step 13	
−1.5	[1100101]
−3.6	[1101]
−3.9	[11100001]
−3.9	[11100000]
−4.6	[1110010]
−4.6	[1110011]
−4.9	[10]
−5.6	[0]
−6	[11101]
−6.7	[1111]
−7.8	[1110001]
−7.8	[1100100]
−8.5	[110011]
−9.2	[11000]

Step 14	
−0.77	[11001010]
−3.6	[1101]
−3.9	[11100000]
−3.9	[11100001]
−4.6	[1110011]
−4.6	[1110010]
−4.9	[10]
−5.6	[0]
−6	[11101]
−6.7	[1111]
−7.1	[11001011]
−7.8	[1110001]
−7.8	[1100100]
−8.5	[110011]
−9.2	[11000]

Figure 12.30: Stack contents for stack algorithm decoding example: metric, [input list].

codeword is found. This is handled by simply throwing away the paths at the bottom of the stack. Of course, if the path that would ultimately become the best path is thrown away at an earlier stage of the algorithm, the best path can never be found. However, if the stack is sufficiently large the probability that the correct path will be thrown away is negligible. Another possibility is to simply throw out the block where the frame overflow occurs and declare an erasure.

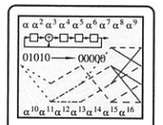
12.8.4 The Fano Algorithm

While the stack algorithm moves around the decoding tree, and must therefore save information about each path still under consideration, the Fano algorithm retains only one path and moves through the tree only along the edges of the tree. As a result, it does not require as much memory as the stack algorithm. However, some of the nodes are visited more than once, requiring recomputation of the metric values.

The general outline of the decoder is as follows. A threshold value T is maintained by the algorithm. The Fano decoder moves forward through the tree as long as the path metric at the next node (the "forward" node), denoted by M_F, exceeds the threshold and the path metric continues to increase (improve). The algorithm is thus a depth-first search. When the path metric would drop below the threshold if a forward move were made, the decoder examines the preceding node (the "backward" node, with path metric M_B). If the path metric at the backward node does not exceed the current threshold, then the threshold is reduced by Δ, and the decoder examines the next forward node. If the path metric at the previous node does exceed the threshold, then the decoder backs up and begins to examine other paths from that node. (This process of moving forward, then backward, then adjusting the threshold and moving forward again is why nodes may be visited many times.) If all nodes forward of that point have already been examined, then the decoder once again considers backing up. Otherwise, the decoder moves forward on one of the remaining nodes.

Each time a node is visited for the first time (and if it is not at the end of the tree) the decoder "tightens" the threshold by the largest multiple of Δ such that the adjusted threshold does not exceed the current metric. (As an alternative, tightening is accomplished in some algorithms by simply setting $T = M_F$.)

Since the Fano algorithm does backtracking, the algorithm needs to keep the following information at each node along the path that it is examining: the path metric at the previous node, M_B; which of the 2^k branches it has taken; the input at that node; and also the state of the encoder, so that next branches in the tree can be computed. A forward move consists of adding this information to the end of the path list. A backward move consists of popping this information from the end of the path list. The path metric at the root node is set at $M_B = -\infty$; when the decoder backs up to that node, the threshold is always reduced and the algorithm moves forward again.



fanoalg.m

Figure 12.31 shows the flowchart for the Fano algorithm. The number i indicates the length of the current path. At each node, all possible branches might be taken. The metric to each next node is stored in sorted order in the array P. At each node along the path, the number t_i indicates which branch number has been taken. When $t_i = 0$, the branch with the best metric is chosen, when $t_i = 1$ the next best metric is chosen, and so forth. The information stored about each node along the path includes the input, the state at that node, and t_i. (Recomputing the metric when backtracking could be avoided by also storing P at each node.)

The threshold is adjusted by the quantity Δ. In general, the larger Δ is, the fewer the

number of computations are required. Ultimately, Δ must be below the likelihood of the maximum likelihood path, and so must be lowered to that point. If Δ is too small, then many iterations might be required to get to that point. On the other hand, if Δ is lowered in steps that are too big, then the threshold might be set low enough that other paths which are not the maximum likelihood path also exceed the threshold and can be considered by the decoder. Based on simulation experience [203], Δ should be in the range of (2,8) if unscaled metrics are used. If scaled metrics are used, then Δ should be scaled accordingly. The value of Δ employed should be explored by thorough computer simulation to ensure that it gives adequate performance.

Example 12.31 Suppose the same code and input sequence as in Example 12.13 is used. The following traces the execution of the algorithm when the scaled (integer) metric of Example 12.29 is used with $\Delta = 10$. The step number n is printed every time the algorithm passes through point A in the flow chart.

$n = 1$: $T = 0$ $P = [2 - 16]$ $t_0 = 0$
Look forward: $M_F = 2$
$M_F \geq T$. Move forward
First visit: Tighten T
$M_F = 2$ $M_B = 0$ Node=[1]
$M = 2$ $T = 0$

$n = 2$: $T = 0$ $P = [4 - 14]$ $t_1 = 0$
Look forward: $M_F = 4$
$M_F \geq T$. Move forward
First visit: Tighten T
$M_F = 4$ $M_B = 2$ Node=[11]
$M = 4$ $T = 0$

$n = 3$: $T = 0$ $P = [-3 - 3]$ $t_2 = 0$
Look forward: $M_F = -3$
$M_F < T$: Look back
$M_B = 2$
$M_B \geq T$: Move back
All forward nodes not yet tested. $t_1 = 1$
$M_F = -3$ $M_B = 2$ Node=[1]
$M = 2$ $T = 0$

$n = 4$: $T = 0$ $P = [4 - 14]$ $t_1 = 1$
Look forward: $M_F = -14$
$M_F < T$: Look back
$M_B = 0$
$M_B \geq T$: Move back
All forward nodes not yet tested. $t_0 = 1$
$M_F = -14$ $M_B = 0$ Node=[]
$M = 0$ $T = 0$

$n = 5$: $T = 0$ $P = [2 - 16]$ $t_0 = 1$
Look forward: $M_F = -16$
$M_F < T$: Look back
$M_B = -\infty$
$M_B < T$: $T = T - \Delta$
$M_F = -16$ $M_B = -\infty$ Node=[]
$M = 0$ $T = -10$

$n = 6$: $T = -10$ $P = [2 - 16]$ $t_0 = 0$
Look forward: $M_F = 2$
$M_F \geq T$. Move forward
$M_F = 2$ $M_B = 0$ Node=[1]
$M = 2$ $T = -10$

$n = 7$: $T = -10$ $P = [4 - 14]$ $t_1 = 0$
Look forward: $M_F = 4$
$M_F \geq T$. Move forward
$M_F = 4$ $M_B = 2$ Node=[11]
$M = 4$ $T = -10$

$n = 8$: $T = -10$ $P = [-3 - 3]$ $t_2 = 0$
Look forward: $M_F = -3$
$M_F \geq T$. Move forward
First visit: Tighten T
$M_F = -3$ $M_B = 4$ Node=[111]
$M = -3$ $T = -10$

$n = 9$: $T = -10$ $P = [-1 - 19]$ $t_3 = 0$
Look forward: $M_F = -1$
$M_F \geq T$. Move forward
First visit: Tighten T
$M_F = -1$ $M_B = -3$ Node=[1110]
$M = -1$ $T = -10$

$n = 10$: $T = -10$ $P = [1 - 17]$ $t_4 = 0$
Look forward: $M_F = 1$
$M_F \geq T$. Move forward
First visit: Tighten T
$M_F = 1$ $M_B = -1$ Node=[11100]
$M = 1$ $T = 0$

$n = 11$: $T = 0$ $P = [-6 - 6]$ $t_5 = 0$
Look forward: $M_F = -6$
$M_F < T$: Look back
$M_B = -1$
$M_B < T$: $T = T - \Delta$
$M_F = -6$ $M_B = -1$ Node=[11100]
$M = 1$ $T = -10$

$n = 12$: $T = -10$ $P = [-6 - 6]$ $t_5 = 0$
Look forward: $M_F = -6$
$M_F \geq T$. Move forward
First visit: Tighten T
$M_F = -6$ $M_B = 1$ Node=[111001]
$M = -6$ $T = -10$

$n = 13$: $T = -10$ $P = [-13 - 13]$
$t_6 = 0$
Look forward: $M_F = -13$
$M_F < T$: Look back
$M_B = 1$
$M_B \geq T$: Move back
All forward nodes not yet tested. $t_5 = 1$
$M_F = -13$ $M_B = 1$ Node=[11100]
$M = 1$ $T = -10$

$n = 14$: $T = -10$ $P = [-6 - 6]$ $t_5 = 1$
Look forward: $M_F = -6$
$M_F \geq T$. Move forward
First visit: Tighten T
$M_F = -6$ $M_B = 1$ Node=[111000]
$M = -6$ $T = -10$

$n = 15$: $T = -10$ $P = [-4 - 22]$ $t_6 = 0$
Look forward: $M_F = -4$
$M_F \geq T$. Move forward
First visit: Tighten T
$M_F = -4$ $M_B = -6$ Node=[1110000]
$M = -4$ $T = -10$

$n = 16$: $T = -10$ $P = [-11 - 11]$
$t_7 = 0$
Look forward: $M_F = -11$
$M_F < T$: Look back
$M_B = -6$
$M_B \geq T$: Move back
All forward nodes not yet tested. $t_6 = 1$
$M_F = -11$ $M_B = -6$ Node=[111000]
$M = -6$ $T = -10$

$n = 17$: $T = -10$ $P = [-4 - 22]$ $t_6 = 1$
Look forward: $M_F = -22$
$M_F < T$: Look back
$M_B = 1$
$M_B \geq T$: Move back
No more forward nodes
$M_B = -1$
$M_B \geq T$: Move back
All forward nodes not yet tested. $t_4 = 1$
$M_F = -22$ $M_B = -1$ Node=[1110]
$M = -1$ $T = -10$

$n = 18$: $T = -10$ $P = [1 - 17]$ $t_4 = 1$
Look forward: $M_F = -17$
$M_F < T$: Look back
$M_B = -3$
$M_B \geq T$: Move back
All forward nodes not yet tested. $t_3 = 1$
$M_F = -17$ $M_B = -3$ Node=[111]
$M = -3$ $T = -10$

$n = 19$: $T = -10$ $P = [-1 - 19]$ $t_3 = 1$
Look forward: $M_F = -19$
$M_F < T$: Look back
$M_B = 4$
$M_B \geq T$: Move back
All forward nodes not yet tested. $t_2 = 1$
$M_F = -19$ $M_B = 4$ Node=[11]
$M = 4$ $T = -10$

$n = 20$: $T = -10$ $P = [-3 - 3]$ $t_2 = 1$
Look forward: $M_F = -3$
$M_F \geq T$. Move forward
First visit: Tighten T
$M_F = -3$ $M_B = 4$ Node=[110]
$M = -3$ $T = -10$

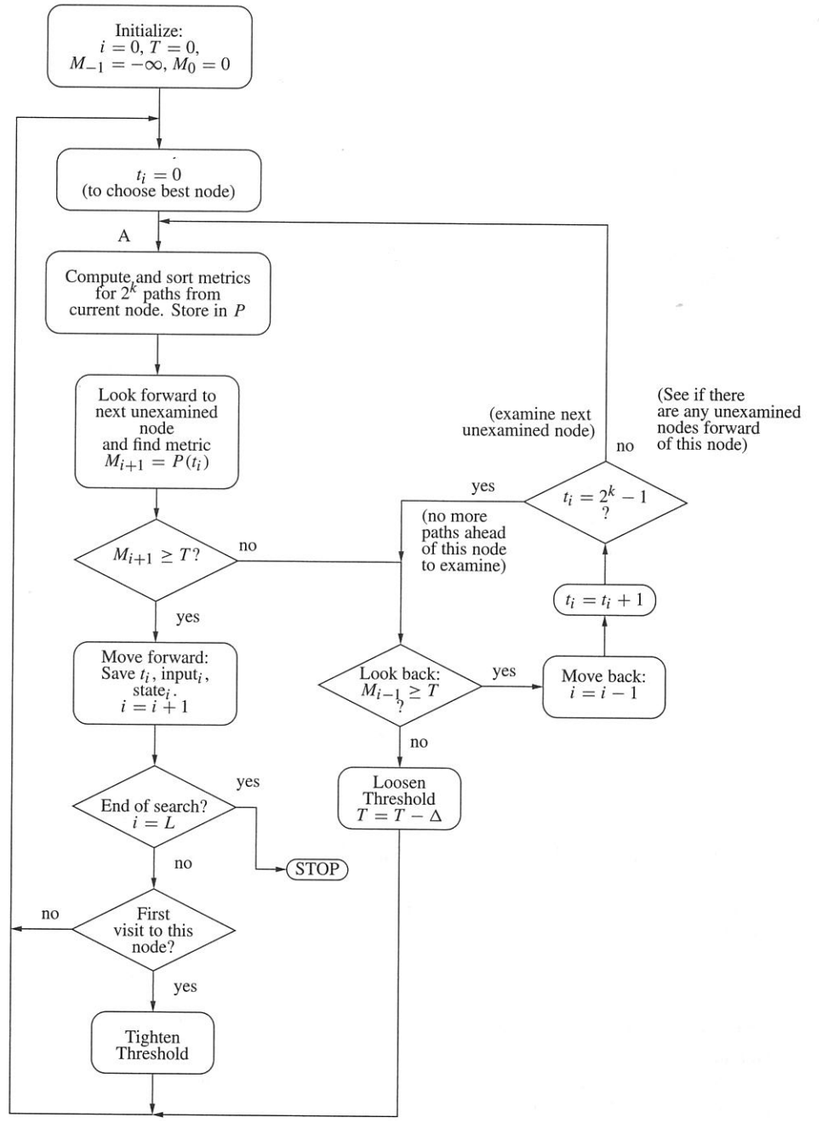


Figure 12.31: Flowchart for the Fano algorithm.

$n = 21$: $T = -10$ $P = [-10 - 10]$
$t_3 = 0$
Look forward: $M_F = -10$
$M_F \geq T$. Move forward
First visit: Tighten T
$M_F = -10$ $M_B = -3$ Node=[1101]
$M = -10$ $T = -10$

$n = 23$: $T = -10$ $P = [-10 - 10]$
$t_3 = 1$
Look forward: $M_F = -10$
$M_F \geq T$. Move forward
First visit: Tighten T
$M_F = -10$ $M_B = -3$ Node=[1100]
$M = -10$ $T = -10$

$n = 25$: $T = -10$ $P = [-6 - 24]$ $t_5 = 0$
Look forward: $M_F = -6$
$M_F \geq T$. Move forward
First visit: Tighten T
$M_F = -6$ $M_B = -8$ Node=[110010]
$M = -6$ $T = -10$

$n = 26$: $T = -10$ $P = [-4 - 22]$ $t_6 = 0$
Look forward: $M_F = -4$
$M_F \geq T$. Move forward
First visit: Tighten T
$M_F = -4$ $M_B = -6$ Node=[1100101]
$M = -4$ $T = -10$

$n = 22$: $T = -10$ $P = [-17 - 17]$
$t_4 = 0$
Look forward: $M_F = -17$
$M_F < T$: Look back
$M_B = -3$
$M_B \geq T$: Move back
All forward nodes not yet tested. $t_3 = 1$
$M_F = -17$ $M_B = -3$ Node=[110]
$M = -3$ $T = -10$

$n = 24$: $T = -10$ $P = [-8 - 26]$ $t_4 = 0$
Look forward: $M_F = -8$
$M_F \geq T$. Move forward
First visit: Tighten T
$M_F = -8$ $M_B = -10$ Node=[11001]
$M = -8$ $T = -10$

$n = 27$: $T = -10$ $P = [-2 - 20]$ $t_7 = 0$
Look forward: $M_F = -2$
$M_F \geq T$. Move forward
First visit: Tighten T
$M_F = -2$ $M_B = -4$ Node=[11001010]
$M = -2$ $T = -10$

For this particular set of data, the value of Δ has a tremendous impact both on the number of steps the algorithm takes and whether it decodes correctly. Table 12.4 shows that the number of decoding steps decreases typically as Δ gets larger, but that the decoding might be incorrect for some values of Δ.

Table 12.4: Performance of Fano Algorithm on a Particular Sequence as a Function of Δ

Δ	Number of Decoding Steps	Correct Decoding	Δ	Number of Decoding Steps	Correct Decoding
1	158	yes	7	31	no
2	86	yes	8	31	no
3	63	no	9	31	no
4	47	no	10	27	yes
5	40	yes	11	16	no
6	33	no	12	16	no

□

In comparing the stack algorithm and the Fano algorithm, we note the following.

- The stack algorithm visits each node only once, but the Fano algorithm may revisit nodes.
- The Fano algorithm does not have to manage the stack (e.g., resort the metrics).

Despite its complexity, when the noise is low the Fano algorithm tends to decode faster than the stack algorithm. However, as the noise increases more backtracking might be required and the stack algorithm has the advantage. Overall, the Fano algorithm is usually selected when sequential decoding is employed.

12.8.5 Other Issues for Sequential Decoding

We briefly introduce some issues related to sequential decoding, although space precludes a thorough treatment. References are provided for interested readers.

Computational complexity The computational complexity is a random variable, and so is described by a probability distribution. Discussions of the performance of the decoder appear in [302, 161, 164, 90].

Code design The performance of a code decoded using the Viterbi algorithm is governed largely by the free distance d_{free}. For sequential decoding however, the codewords must have a distance that increases as rapidly as possible over the first few symbols of the codeword (i.e., the code must have a good **column distance function**) so that the decoding algorithm can make good decisions as early as possible. A large free distance and a small number of nearest neighbors are also important. A code having an optimum distance profile is one in which the column distance function over the first constraint length is better than all other codes of the same rate and constraint length. Tables of codes having optimum distance profiles are provided in [174]. Further discussion and description of the algorithms for finding these codes appear in [169, 170, 171, 172, 43, 173].

Variations on sequential decoding algorithms In the interest of reducing the statistical variability of the decoding, or improving the decoder performance, variations on the decoding algorithms have been developed. In [48], a multiple stack algorithm is presented. This operates like the stack algorithm, except that the stack size is limited to a fixed number of entries. If the stack fills up before decoding is complete, the top paths are transferred to a second stack and decoding proceeds using these best paths. If this stack also fills up before decoding is complete, a third stack is created using the best paths, and so forth. In [79] and [166], hybrid algebraic/sequential decoding was introduced in which algebraic constraints are imposed across frames of sequentially decoded data. In [129], a generalized stack algorithm was proposed, in which more than one path in the stack can be extended at one time (as in the Viterbi algorithm) and paths merging together are selected as in the Viterbi algorithm. Compared to the stack algorithm, the generalized stack algorithm does not have buffer size variations as large and the error probability is closer to that of the Viterbi algorithm.

12.8.6 A Variation on the Viterbi Algorithm: The M Algorithm

For a trellis with a large number of states at each time instant, the Viterbi algorithm can be very complex. Furthermore, since there is only one correct path, most of the computations are expended in propagating paths that are not be used, but must be maintained to ensure the optimality of the decoding procedure. The M algorithm (see, e.g., [270]) is a suboptimal, breadth-first decoding algorithm whose complexity is parametric, allowing for more complexity in the decoding algorithm while decoding generally closer to the optimum.

A list of M paths is maintained. At each time step, these M paths are extended to $M2^k$ paths (where k is the number of input bits), and the path metric along each of these paths is computed just as for the Viterbi algorithm. The path metrics are sorted, then the best M paths are retained in preparation for the next step. At the end of the decoding cycle, the path with the best metric is used as the decoded value. While the underlying graphical structure for the Viterbi algorithm is a trellis, in which paths merge together, the underlying graphical structure for the M algorithm is a tree: the merging of paths is not explicitly represented, but better paths are retained by virtue of the sorting operation. The M-algorithm is thus a cross between the stack algorithm (but using all paths of the same length) and the Viterbi

algorithm. If M is equal to the number of states, the M algorithm is nearly equivalent to the Viterbi algorithm. However, it is possible for M to be significantly less than the number of states with only modest loss of performance.

Another variation is the T-algorithm. It starts just like the M algorithm. However, instead of retaining only the best M paths, in the T algorithm all paths which are within a threshold T of the best path at that stage are retained.

12.9 Convolutional Codes as Block Codes

In this section, we use $m = \max_i v_i$ as the maximum amount of memory in any of the elements of the transfer function matrix.

A block code takes a fixed-length block of k symbols and maps it to a block of n symbols. Convolutional codes, on the other hand, can operate on an entire "stream" of data: a stream of data is simply passed through the filtering system of the convolutional coder. As a result, the "block length" of the code is not usually referred to in the context of convolutional codes.

However, convolutional encoders can be used as encoders for block codes. In fact, this perspective allows bounds on block codes to provide useful bounds for convolutional codes (see, e.g., [197]). Here are some natural ways that block codes can be obtained from convolutional codes. (This discussion applies to polynomial transfer function matrices. Some modifications are necessary for creation of the transfer function matrices.)

Truncation A sequence of L blocks of k-bits can be input to the decoder. This results in an (nL, nk) decoder. This has the disadvantage that there is little (if any) error protection afforded to the last digits input into the encoder [213], resulting in what is called unequal error protection. The effect of unequal error protection is shown in Figure 12.19. Decoding takes place using the Viterbi algorithm starting in state 0 and ending after L stages at any state.

Zero tail Following L k-bit blocks of bits, a sequence of m k-bit blocks of zeros is input to the encoder, driving the state of the encoder to the zero state. The resultant code is an $((L+m)n, kL)$ decoder, with rate $R = kL/(L+m)n$. There is thus a loss of rate, but for large block lengths the rate reduction is negligible. Decoding is accomplished using the Viterbi algorithm starting in state 0 and ending in state 0.

Tail biting In a tail-biting codeword, no additional bits are appended to drive the encoder to the zero state. Instead, the encoder ends in whatever state it happens to end in after the input bits are encoded. There is thus no loss of rate due to the zero-state forcing sequence. The encoder is modified to avoid the problem of unequal error protection by allowing it to start in *any state* and not just the 0-state. Then the initial state is determined by the terminal bits in the sequence. Then valid codewords are those that *start and end in the same state.*

For feedforward encoders, the state is determined by the most recent v input bits. The final state is thus determined by the last v input bits. Since the initial and final state must match, the initial state is also determined by the last v input bits. This allows one to view the trellis as a circular trellis: the final state of the trellis wraps around to become the initial state of the trellis. (This circular trellis structure initially gave rise to the term tail-biting.)

Decoding a tail-biting code more complicated: the Viterbi algorithm should find a path which starts and ends in the same state. In principle, this could require running the decoder 2^k times, starting in each possible state, then checking that the best path terminates in the same state as the starting state. This is computationally infeasible for many codes. Variations on tail biting codes and their decoding algorithms, are described in [213]. One variation runs through two passes. In the first pass, decoding starts at an arbitrary state (such as the 0 state) and finds the terminal state with the best path metric. Then the Viterbi algorithm is run again, starting with the initial state as that terminal state. Other alternatives are in [4].

12.10 Trellis Representations of Block and Cyclic Codes

In this section we take a dual perspective to that of the previous section: we describe how linear block codes can be represented in terms of a trellis. Besides theoretical insight, the trellis representation can also be used to provide a means of soft-decision decoding that does not depend upon any particular algebraic structure of the code. These decoding algorithms can make block codes "more competitive with convolutional codes" [205, p. 3].

12.10.1 Block Codes

We demonstrate the trellis idea with a $(7, 4, 3)$ binary Hamming code. Let

$$H = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{h}_1 & \mathbf{h}_2 & \mathbf{h}_3 & \mathbf{h}_4 & \mathbf{h}_5 & \mathbf{h}_6 & \mathbf{h}_7 \end{bmatrix} \tag{12.47}$$

be the parity check matrix for the code. Then a column vector $\mathbf{x}$ is a codeword if and only if $\mathbf{s} = H\mathbf{x} = \mathbf{0}$; that is, the syndrome $\mathbf{s}$ must satisfy

$$\mathbf{s} = \begin{bmatrix} \mathbf{h}_1 & \mathbf{h}_2 & \mathbf{h}_3 & \mathbf{h}_4 & \mathbf{h}_5 & \mathbf{h}_6 & \mathbf{h}_7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \sum_{i=1}^{7} \mathbf{h}_i x_i = \mathbf{0}.$$

We define the *partial syndrome* by

$$\mathbf{s}_{r+1} = \sum_{i=1}^{r} \mathbf{h}_i x_i = \mathbf{s}_r + \mathbf{h}_r x_r,$$

with $\mathbf{s}_1 = \mathbf{0}$. Then the $\mathbf{s}_{n+1} = \mathbf{s}$.

A trellis representation of a code is obtained by using $\mathbf{s}_r$ as the state, with an edge between a state $\mathbf{s}_r$ and $\mathbf{s}_{r+1}$ if $\mathbf{s}_{r+1} = \mathbf{s}_r$ (corresponding to $x_r = 0$) or if $\mathbf{s}_{r+1} = \mathbf{s}_r + \mathbf{h}_r$ (corresponding to $x_r = 1$). Furthermore, the trellis is terminated at the state $\mathbf{s}_{n+1} = \mathbf{0}$, corresponding to the fact that a valid codeword has a syndrome of zero. The trellis has at most 2^{n-k} states in it.

Figure 12.32 shows the trellis for the parity check matrix of (12.47). Horizontal transitions correspond to $x_i = 0$ and diagonal transitions correspond to $x_i = 1$. Only those paths which end up at $\mathbf{s}_8 = \mathbf{0}$ are retained.

As may be observed, the trellis for a block code is "time-varying" — it has different connections for each section of the trellis. The number of states "active" at each section of the trellis also varies.

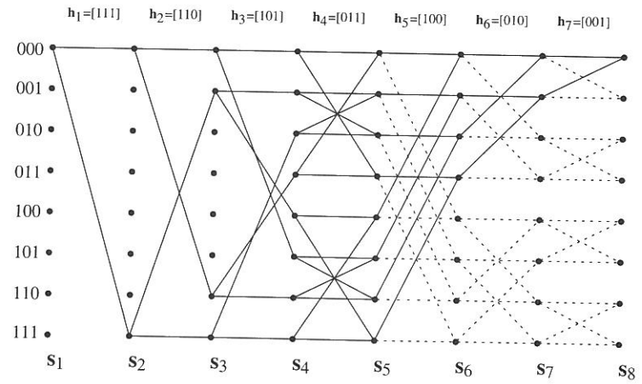


Figure 12.32: The trellis of a (7, 4) Hamming code.

For a general code, the trellis structure is sufficiently complicated that it may be difficult to efficiently represent in hardware. There has been recent work, however, on families of codes whose trellises have a much more regular structure. These are frequently obtained by recursive constructions (e.g., based on Reed-Muller codes). Interested readers can consult [205].

12.10.2 Cyclic Codes

An alternative formulation of a trellis is available for a cyclic code. Recall that a cyclic code can be encoded using a linear feedback shift register as a syndrome computer. The sequence of possible states in this encoder determines a trellis structure which can be used for decoding. We demonstrate the idea again using a (7, 4, 3) Hamming decoder, this time represented as a cyclic code with generator polynomial $g(x) = x^3 + x + 1$.

Figure 12.33 shows a systematic encoder. For the first $k = 4$ clock instants, switch 1 is closed (enabling feedback) and switch 2 is in position 'a'. After the systematic part of the data has been clocked through, switch 1 is opened and switch 2 is moved to position 'b'. The state contents then shift out as the coefficients of the remainder polynomial. Figure

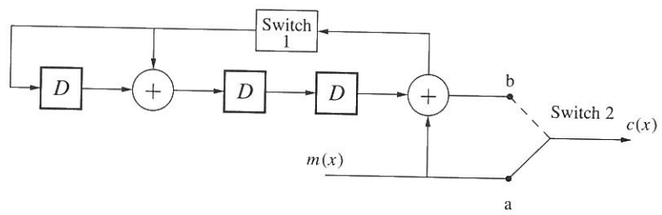


Figure 12.33: A systematic encoder for a (7, 4, 3) Hamming code.

12.34 shows the trellis associated with this encoder. For the first $k = 4$ bits, the trellis state depends upon the input bit. The coded output bit is equal to the input bit. For the last

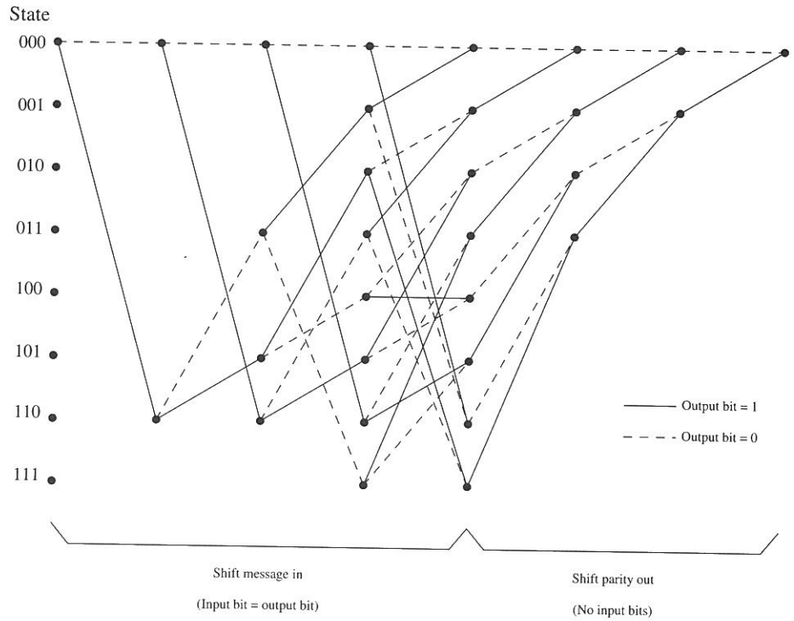


Figure 12.34: A trellis for a cyclically encoded (7,4,3) Hamming code.

$n - k = 3$ bits, the next state is determined simply by shifting the current state. There are no input bits so the output is equal to the bit that is shifted out of the registers.

12.10.3 Trellis Decoding of Block Codes

Once a trellis for a code is established by either of the methods described above, the code can be decoded with a Viterbi algorithm. The time-varying structure of the trellis makes the indexing in the Viterbi algorithm perhaps somewhat more complicated, but the principles are the same. For example, if BPSK modulation is employed, so that the transmitted symbols are $a_i = 2c_i - 1 \in \{\pm 1\}$, and that the channel is AWGN, the branch metric for a path taken with input x_i is $(r_i - (2x_i - 1))^2$. Such soft decision decoding can be shown to provide up to 2 dB of gain compared to hard decision decoding (see, for example [303, pp. 222–223]). However, this improvement does not come without a cost: for codes of any appreciable size, the number of states 2^{n-k} can be so large that trellis-based decoding is infeasible.

Programming Laboratory 9:

Programming Convolutional Encoders

Objective

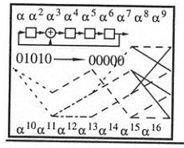
In this lab you are to create a program structure to implement both polynomial and systematic rational convolutional encoders.

Background

Reading: Sections 12.1, 12.2.

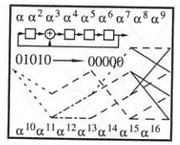
Since both polynomial and systematic ration encoders are "convolutional encoders" and they share many attributes. Furthermore, when we get to the decoding operations, it is convenient to employ one decoder which operates on data from either kind of encoder. As a result, it is structurally convenient to create a base class `BinConv`, then create two derived classes, `BinConvFIR` and `BinConvIIR`. Since the details of the encoding operation and the way the state is determined differ, each of these classes employs its own encoder function. To achieve this, a virtual function `encode` is declared in the base class, which is then realized separately in each derived class.[6] Also, virtual member functions `getstate` and `setstate` are used for reading and setting the state of the encoder. These can be used for testing purposes; they are also used to build information tables that the decoder uses.

The declaration for the `BinConv.h` base class is shown here.



Algorithm 12.3 Base Class for Binary Convolutional Encoder
File: `BinConv.h`

The derived classes `BinConvFIR` and `BinConvIIR` are outlined here.



Algorithm 12.4 Derived classes for FIR and IIR Encoders
File: `BinConvFIR.h`
`BinConvIIR.h`
`BinConvFIR.cc`
`BinConvIIR.cc`

Programming Part

1) Write a class `BinConvFIR` that implements convolutional encoding for a general polynomial encoder. That is, the generator matrix is of the form in (12.1), where each $g^{(i,j)}(x)$ is a polynomial. The class should have an appropriate constructor and destructor. The class should implement the virtual functions `encode`, `getstate`, and `setstate`, as outlined above.

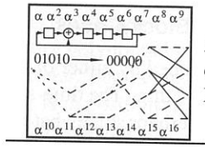
Test your encoder as follows:

a) Using the encoder with transfer function

$$G(x) = \begin{bmatrix} 1+x^2 & 1+x+x^2 \end{bmatrix},$$

verify that the impulse response is correct, that the `getstate` and `nextstate` functions work as expected, and that the state/nextstate table is correct. Use Figure 12.5.

The program `testconvenc.cc` may be helpful.



Algorithm 12.5 Test program for convolutional encoders
File: `testconvenc.cc`

b) The polynomial transfer function

$$G(x) = \begin{bmatrix} x^2+x+1 & x^2 & 1+x \\ x & 1 & 0 \end{bmatrix} \qquad (12.48)$$

has the state diagram and trellis shown in Figure 12.35. Verify that for this encoder, the impulse response is correct, that the `getstate` and `nextstate` functions work as expected, and that the state/nextstate table is correct.

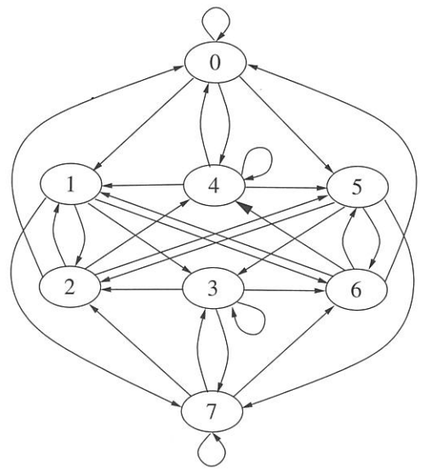
2) Write a class `BinConvIIR` that implements a systematic encoder (possibly employing IIR filters). The generator matrix is of the form

$$G(x) = \begin{bmatrix} & \frac{p_1(x)}{q_1(x)} \\ & \frac{p_2(x)}{q_2(x)} \\ I & \vdots \\ & \frac{p_k(x)}{q_k(x)} \end{bmatrix}$$

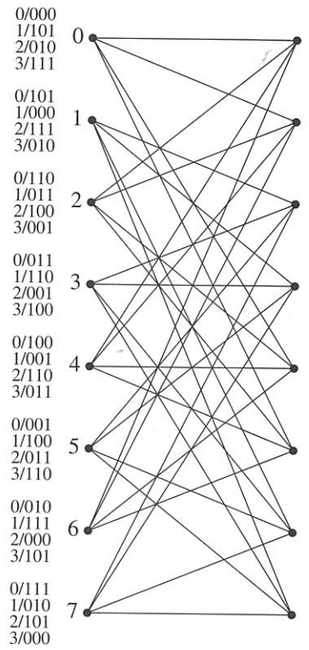
for polynomials $p_i(x)$ and $q_i(x)$.

Test your class using the recursive systematic encoder of (12.2), checking as for the first case. (You may find it convenient to find the samples of the impulse by long division.)

[6] This is a tradeoff between flexibility and speed. In operation, the virtual functions are called via a pointer, so there is a pointer-lookup overhead associated with them. This also means that virtually called functions cannot be inline, even if they are very small. However, most of the computational complexity associated with these codes is associated with the decoding operation, which takes advantage of precomputed operations. So for our purposes, the virtual function overhead is not too significant.



(a) State diagram



(b) Trellis

Figure 12.35: State diagram and trellis for the encoder in (12.48)

Programming Laboratory 10: Convolutional Decoders: The Viterbi Algorithm

Objective

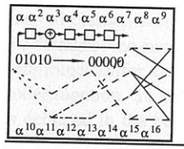
You are to write a convolutional decoder class that decodes using both hard and soft metrics with the Viterbi algorithm.

Background

Reading: Section 12.3

While there are a variety of ways that the Viterbi algorithm can be structured in C++, we recommend using a base class `Convdec.h` that implements that actual Viterbi algorithm and using a virtual function `metric` to compute the metric. This is used by derived classes `BinConvdec01` (for binary 0-1 data) and `BinConvBPSK` (for BPSK modulated data), where each derived class has its own metric function.

The base class `Convdec` This class is a base class for all of the Viterbi-decoded objects.

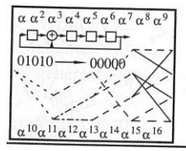


Algorithm 12.6 The Base Decoder Class Declarations
File: `Convdec.h`
`Convdec.cc`

In this class, an object of type `BinConv` (which could be either an FIR or IIR convolutional encoder, if you have used the class specification in lab 9) is passed in. The constructor builds appropriate data arrays for the Viterbi algorithm, placing them in the variables `prevstate` and `inputfrom`. A virtual member function `metric` is used by derived classes to compute the branch metric. The core of the algorithm is used in the member function `viterbi`, which is called by the derived classes. Some other functions are declared:

- `showpaths` — You may find it helpful while debugging to dump out information about the paths. This function (which you write) should do this for you.
- `getinpnow` — This function decodes the last available branch in the set of paths stored, based on the best most recent metric. If `adv` is asserted, the pointer to the end of the branches is incremented. This can be used for dumping out the decisions when the end of the input stream is reached.
- `buildprev` builds the state/previous state array, which indicates the connections between states of the trellis.

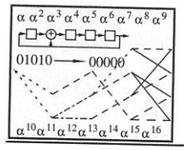
The derived class `BinConvdec01` The first derived class is `BinConvdec01.h`, for binary 0-1 decoding using the Hamming distance as the branch metric.



Algorithm 12.7 Convolutional decoder for binary (0,1) data
File: `BinConvdec01.h`
`BinConvdec01.cc`

This class provides member data `outputmat`, which can be used for direct lookup of the output array given the state and the input. Since the output is, in general, a vector quantity, this is a three-dimensional array. It is recommended that space be allocated using `CALLOCTENSOR` defined in `matalloc.h`. The member variable `data` is used by the `metric` function, as shown. The class description is complete as shown here, except for the function `buildoutputmat`, which is part of the programming assignment.

The derived class `BinConvdecBPSK` The next derived class is `BinConvdecBPSK.h`, for decoding BPSK-modulated convolutionally coded data using the Euclidean distance as the branch metric.



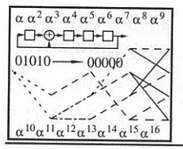
Algorithm 12.8 Convolutional decoder for BPSK data
File: `BinConvdecBPSK.h`
`BinConvdecBPSK.cc`

As for the other derived class, space is provided for `outputmat` and `data`; the class is complete as presented here except for the function `buildoutputmat`.

Programming Part

1) Finish the functions in `Convdec.cc`.

2) Test the binary (0,1) `BinConvdec01` decoder for the encoder $G(x) = [1 + x^2, 1 + x + x^2]$ by reproducing the results in Example 12.13. The program `testconvdec` can help.



Algorithm 12.9 Test the convolutional decoder
File: `testconvdec.cc`

3) Test the convolutional decoder `BinConvdecBPSK` by modulating the $\{0, 1\}$ data. Again, `testconvdec` can help.

4) Determine the performance of the encoder $G(x) = [1 + x^2 \quad 1 + x + x^2]$ by producing an error curve on the AWGN channel using BPSK modulation with a soft metric. Compare the soft metric performance with hard metric performance, where the BSC is modeled as having crossover probability $p_c = Q(\sqrt{2E_c/N_0})$. Compare the two kinds coded performances with uncoded BPSK modulation. Also, plot the bound (12.40) and the approximation (12.42) on the same graph.

How much coding gain is achieved? How do the simulation results compare with the theoretical bounds/approximations? How do the simulations compare with the theoretically predicted asymptotic coding gain? How much better is the soft-decoding than the hard-decoding?

5) Repeat the testing, but use the catastrophic code with encoder $G(x) = [1 + x, 1 + x^2]$. How do the results for the noncatastrophic encoder compare with the results for the catastrophic encoder?

12.11 Exercises

12.1 For the $R = 1/2$ convolutional encoder with

$$G(x) = \begin{bmatrix} 1 + x^2 + x^3 & 1 + x + x^3 \end{bmatrix} \tag{12.49}$$

(a) Draw a hardware realization of the encoder.
(b) Determine the convolutional generator matrix G.
(c) For the input sequence $\mathbf{m} = [1, 0, 1, 1, 0, 1, 1]$ determine the coded output sequence.
(d) Draw the state diagram. Label the branches of the state diagram with input/output values.
(e) Draw the trellis.
(f) What is the constraint length of the code?
(g) Determine the State/Next State table.
(h) Determine the State/Previous State table.
(i) Is this a catastrophic realization? Justify your answer.
(j) Determine the weight enumerator $T(D, N)$.
(k) What is d_{free}?
(l) Determine upper and lower bounds on P_b for a BSC using (12.36) and (12.37) and an approximation using (12.30). Plot as a function of the signal-to-noise ratio, where $p_c = Q(\sqrt{2E_c/N_0})$. Compare the bounds to uncoded performance.
(m) Determine upper and lower bounds on P_b for an AWGN channel using (12.40) and (12.41) and plot as a function of the signal-to-noise ratio.
(n) Determine the theoretical asymptotic coding gain for the BSC and AWGN channels. Compare with the results from the plots. Also, comment on the difference (in dB) between the hard and soft metrics.
(o) Express $G(x)$ as a pair of octal numbers using both leading 0 and trailing 0 conventions.
(p) Suppose the output of a BSC is $\mathbf{r} = [11, 11, 00, 01, 00, 00, 10, 10, 10, 11]$. Draw the trellis for the Viterbi decoder and indicate the maximum likelihood path through the trellis. Determine the maximum likelihood estimate of the transmitted codeword and the message bits. According to this estimate, how many bits of $\mathbf{r}$ are in error?

12.2 For the $R = 1/3$ convolutional coder with

$$G(x) = \begin{bmatrix} 1 + x & 1 + x^2 & 1 + x + x^2 \end{bmatrix}$$

(a) Draw a hardware realization of the encoder.
(b) Determine the convolutional generator matrix G.
(c) For the input sequence $\mathbf{m} = [1, 0, 1, 1, 0, 1, 1]$ determine the coded output sequence.
(d) Draw the state diagram. Label the branches of the state diagram with input/output values.
(e) Draw the trellis.
(f) What is the constraint length of the code?
(g) Determine the State/Next State table.
(h) Determine the State/Previous State table.
(i) Is this a catastrophic realization? Justify your answer.
(j) Determine the weight enumerator $T(D, N)$.
(k) What is d_{free}?
(l) Express $G(x)$ as a triplet of octal numbers.

12.3 Find a catastrophic encoder equivalent to $G(x) = \begin{bmatrix} 1+x^2 & 1+x+x^2 \end{bmatrix}$ and determine an infinite-weight message $m(x)$ that results in a finite-weight codeword for this catastrophic encoder.

12.4 Show that $G_4(x)$ defined in (12.8) is equivalent to $G_2(x)$ of (12.5).

12.5 Let $G(x)$ be the transfer function matrix of a basic convolutional code. Show that $G(x)$ is equivalent to a basic transfer function matrix $G'(x)$ if and only if $G'(x) = T(x)G(x)$, where $T(x)$ is a unimodular matrix.

12.6 Determine a systematic encoder transfer function matrix equivalent to

$$G(x) = \begin{bmatrix} 1+x & x & 1 \\ x^2 & 1 & 1+x+x^2 \end{bmatrix}.$$

12.7 For the transfer function matrices

$$G(x) = \begin{bmatrix} 1+x & x & 1 \\ x^2 & 1 & 1+x+x^2 \end{bmatrix} \quad \text{and} \quad G'(x) = \begin{bmatrix} 1+x & x & 1 \\ 1+x^2+x^3 & 1+x+x^2+x^3 & 0 \end{bmatrix}$$

(a) Show that $G(x)$ is equivalent to $G'(x)$.
(b) Show that $G(x)$ is a minimal basic encoder matrix.
(c) Show that $G'(x)$ is not a minimal basic encoder matrix.
(d) Using the procedure described in association with (12.13), determine a transfer function matrix $G''(x)$ which is a minimal basic encoding matrix equivalent to $G'(x)$, but different from $G(x)$.

12.8 For the code generated by

$$G(x) = \begin{bmatrix} \frac{1}{1+x+x^2} & \frac{x}{1+x^3} & \frac{1}{1+x^3} \\ \frac{x^2}{1+x^3} & \frac{1}{1+x^3} & \frac{1}{1+x^3} \end{bmatrix},$$

use elementary row operations to convert the generator matrix to systematic form. Draw a circuit realization of the systematic encoder.

12.9 Catastrophic codes.

(a) For a rate $R = 1/2$ code, let $g_1(x) = 1+x$, $g_2(x) = x^2+1$. Show that when $m(x) = \frac{1}{1+x}$ that the transmitted sequence has finite weight. Determine $\text{GCD}(g_1(x), g_2(x))$.
(b) Motivated by this result, prove the following: For a rate $1/n$ code, if

$$\text{GCD}[g_1(x), g_2(x), \ldots, g_n(x)] = 1,$$

then the code is noncatastrophic.

12.10 For the catastrophic code with generators $g_1(x) = 1+x$, $g_2(x) = 1+x^2$:

(a) Draw the state diagram.
(b) Determine the weight enumerator $T(D, N)$ for the code.
(c) What is the minimum free distance of the code?
(d) How is the catastrophic nature of the code evidenced in the weight enumerator?

12.11 For the generator $G(x) = \begin{bmatrix} 1+x^2, 1+x+x^2+x^3 \end{bmatrix}$:

(a) Find the GCD of the generator polynomials.
(b) Find an infinite-weight message sequence that generates a codeword of finite weight.

12.12 Prove that d_{free} is independent of the encoder realization, so that it is a property of the *code* and not of a particular *encoder* for the code.

12.13 Show that the formal series expansion

$$\frac{1}{1-x} = 1+x+x^2+x^3+\cdots$$

is correct. Show the formal series expansions of

$$\frac{1}{1-2D} \quad \text{and} \quad \frac{D^5L^3N}{1-DLN(1+L)}.$$

12.14 Show that the expressions for P_d in (12.19) and (12.20) can be bounded by $P_d < [4p_c(1-p_c)]^{d/2}$. *Hint*: Show that $\sum_{i=(d+1)/2}^{d} \binom{d}{i} p_c^i (1-p_c)^{d-i} < \sum_{i=(d+1)/2}^{d} \binom{d}{i} p_c^{d/2}(1-p_c)^{d/2}$.

12.15 For a BSC where $Z = \sqrt{4p_c(1-p_c)}$, show that (12.27) can be replaced by $P_d < Z^{d+1}$ when d is odd. Using this result, show that (12.28) can be replaced by

$$P_e(j) < \frac{1}{2}[(1+Z)T(Z) + (1-Z)T(-Z)].$$

12.16 [147] An upper bound on d_{free}. Let K be the number of outputs determining the output of a rate $R = 1/n$ code (i.e., K is the constraint length). The code can be represented by a matrix such as that in (12.3), in which all rows are obtained by shifting the first row.

(a) Show that for any binary linear code, if the codewords are arranged as the rows of a matrix, then any column is either all zeros or half zeros and half ones.
(b) Consider the set of all sequences of length no greater than L. Show that the code generated by these finite-length sequences has length $(K-1+L)n$ symbols. Also show that the average weight of all codewords (excluding the all-zero codeword) is $w_{\text{av}}(L) \leq 2^{L-1}(K-1+L)n/(2^L-1)$.
(c) Argue that the code has a minimum distance between paths of $d_{\text{free}} \leq w_{\text{av}}(L)$.

12.17 Show that (12.21) and (12.22) are correct.

12.18 A code with $k = 1$ has weight enumerator $T(D, N) = \frac{D^5N}{1-2ND}$. The codewords are passed through a BSC with $p_c = 0.01$. Compute upper and lower bounds on the node error probability and the bit-error rate for Viterbi decoding. Repeat this when the code is passed through an AWGN channel with $E_b/N_0 = 6$ dB.

12.19 For a rate $R = 1/2$ code, suppose the output sequence

$$\mathbf{c} = (c_0^{(1)}, c_0^{(2)}, c_1^{(1)}, c_1^{(2)}, c_2^{(1)}, c_2^{(2)}, c_3^{(1)}, c_3^{(2)}, c_4^{(1)}, c_4^{(2)}, c_5^{(1)}, c_5^{(2)}, \ldots)$$

is punctured as

$$\mathbf{c} = (c_0^{(1)}, c_0^{(2)}, -, c_1^{(2)}, c_2^{(1)}, -, c_3^{(1)}, c_3^{(2)}, -, c_4^{(2)}, c_5^{(1)}, -, \ldots).$$

Write down the puncture matrix P.

12.20 A binary-input/binary-output channel with input a and output r has transition probabilities

$$P(r=0|a=0)=0.9 \quad P(r=0|a=1)=0.3$$
$$P(r=1|a=0)=0.1 \quad P(r=1|a=1)=0.7.$$

(a) Determine the log likelihoods.

(b) Scale and shift these values to obtain a set of bit metrics that can be reasonably approximated with not more than 3 bits.

12.21 A channel has binary inputs and three outputs, 0 and 1, and E, where E denotes an erasure. When an erasure occurs, the symbol is known to be suspicious and does not influence the decoding process — it is erased. (It is a lot like a punctured bit). This channel is called the *binary erasure channel*. The channel has transition probabilities

$$P(r=0|a=0)=0.6 \quad P(r=E|a=0)=0.3 \quad P(r=1|a=0)=0.1$$
$$P(r=0|a=1)=0.2 \quad P(r=E|a=1)=0.2 \quad P(r=1|a=1)=0.6$$

(a) Determine the log likelihood ratios.

(b) Scale and shift these values to obtain a set of bit metrics that can be reasonably approximated with not more than 3 bits, making sure that erased symbols do not contribute differentially to the path metric.

12.22 An AWGN with variance $\sigma^2 = 2$ is used with BPSK-modulated data sending signals with amplitudes $a = \pm 1$. The received signal r_t is quantized to four different values $q = Q[r]$ with quantization thresholds at ± 1.5 and 0.

(a) Determine the probabilities $P(q_t|a)$ and the log probabilities $-\log P(q_t|a)$.

(b) Determine a and b so that $a(-\log P(q_t|a) - b)$ can be approximated well by integers using at most two bits.

12.23 A binary input/binary output channel with input a and output r has

$$P(r=0|a=0)=0.99999 \quad P(r=0|a=1)=0.05$$
$$P(r=1|a=0)=0.00001 \quad P(r=1|a=1)=0.95.$$

(a) Determine Z in the Chernoff bound from (12.33).

(b) The input to this channel is coded using a convolutional code whose path enumerator is given by

$$T(D,N)=\frac{D^5N}{1-2ND}.$$

Using (12.28), determine an upper bound on the node error probability $P_e(j)$. Using (12.29) and (12.30), determine an upper bound and an approximation on the bit error rate P_b.

12.24 Chernoff bound. Let $X_1, X_2, \ldots, X_n$ be independent random variables with densities $p_i(x)$ and moment generating functions $\phi_i(s) = E[e^{sX_i}]$. Let $Z = \sum_{i=1}^n X_i$, with moment generating function $\phi_Z(s)$. Using the following steps, show that

$$P(Z \geq \gamma) \leq e^{-s\gamma} \prod_{i=1}^n \phi_i(s).$$

for all $s \geq 0$ such that $\phi_i(s)$ exists.

(a) Let $\phi_Z(s)$ be the moment generating function for Z. Show that $\phi_Z(s) = \prod_{i=1}^n \phi_i(s)$.

(b) Show that $\int_{-\infty}^{\infty} e^{sZ} f_Z(z)\,dz \geq \int_{\gamma}^{\infty} e^{sZ} f_Z(z)\,dz$.

(c) Finish the proof.

12.25 Show that (12.39) is correct.

12.26 A rate-compatible punctured convolutional code (RCPC) based on a rate $R = 1/4$ convolutional code has puncturing period 8 and puncturing matrices

$$P_1 = \begin{bmatrix} 1&1&1&1&1&1&1&1 \\ 1&1&1&1&1&1&1&1 \\ 1&1&1&1&1&1&1&1 \\ 1&1&1&1&1&1&1&1 \end{bmatrix} \quad P_2 = \begin{bmatrix} 1&1&1&1&1&1&1&1 \\ 1&0&1&0&1&0&1&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \end{bmatrix}$$

$$P_3 = \begin{bmatrix} 1&1&1&1&0&1&1&1 \\ 1&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \end{bmatrix}$$

(a) Determine the actual rate when using the puncture matrix P_1. Also for P_2 and P_3.

(b) The generators for the convolutional code are $g^{(1)}(x) = 1 + x^3 + x^4$, $g^{(2)}(x) = 1 + x + x^2 + x^4$, $g^{(3)}(x) = 1 + x^2 + x^3 + x^4$, and $g^{(4)}(x) = 1 + x + x^3 + x^4$. Draw a convolutional encoder capable of transmitting at these three different rates.

12.27 Determine the branch Fano metric for binary transmission of a rate $R = 1/3$ code through a BSC with $p_c = 0.1$. Then scale the metric so it has nearly integer values. Repeat for $p_c = 0.05$ and $p_c = 0.001$.

12.28 For the asymmetric channel in Exercise 12.20, determine the Fano metric for a rate $R = 1/2$ code. Then shift and scale the metric so it has nearly integer values.

12.29 For the code with generator

$$G(x) = \begin{bmatrix} 1 + x^2 + x^3 & 1 + x + x^3 \end{bmatrix},$$

the sequence $\mathbf{r} = [11, 11, 11, 01, 11, 00, 00]$ is received through a BSC with $p_c = 0.125$. Using the stack algorithm, determine the transmitted sequence. Repeat using the Fano algorithm. (The Matlab code may prove very helpful.)

12.30 Draw a trellis representing the binary block code with parity check matrix

$$H = \begin{bmatrix} 0&0&0&1&1&1 \\ 0&1&1&0&0&1 \\ 1&0&1&0&1&0 \end{bmatrix}.$$

12.31 Draw a trellis representation for the cyclic code with generator $g(x) = x^3 + x^2 + 1$ that employs a systematic encoder.

12.12 References

Convolutional codes were introduced in 1955 by Elias [76]. Our presentation overall has benefited greatly from [303]. The discussion of structural properties comes from that source, which, in turn, closely follows [175]. This, in turn, builds on the landmark paper on the algebraic structure of convolutional codes [97]. Catastrophic codes were first discussed in [226]. The criterion for catastrophic codes in terms of the GCD of the generators appears in [226]. Extensive simulation studies of convolutional codes and error curves appears in [148]. The results here were computed by Ojas Chauhan.

The Viterbi algorithm was described in [358]. However, it was not until [89] that the Viterbi algorithm was shown to be a maximum likelihood sequence estimator. This paper also presented the weight enumerator and the graph analysis associated with the performance of convolutional codes.

An important and still very relevant source on convolutional codes is [357]. This book presents random coding performance bounds for convolutional codes and shows that convolutional codes have a higher cutoff rate than block codes. A recent book dedicated to convolutional codes is [174]. Convolutional codes are also presented in most books on coding theory and digital communication theory.

Puncturing appears to have been first explored in [40]. Tail biting was introduced in [213]. Work on short tailbiting codes with many examples of good codes appears in [320]. Basic results regarding the structure of tail-biting trellises appears in [189]. The trellis representation of a block code was presented first in [11] and developed more fully in [377]. It has been the topic of a detailed monograph [205]. Readers interested in fully developed design methodologies should consult that source. A summary of this work is in [303].

The stack algorithm was explored in [388] and [165]. The genre of sequential decoding algorithms was explored early on in [378]. The Fano algorithm appeared first in [80]. The Fano metric received theoretical foundation as a maximum likelihood metric in [223]. A comparison of sequential decoding algorithms appears in [115, 116]. A discussion of the performance of the M algorithm as a function of M and comparison with the Viterbi algorithm is summarized in [303].

Chapter 13

Trellis Coded Modulation

13.1 Adding Redundancy by Adding Signals

The error correction codes studied up to this point in the book have added redundancy by increasing the number of coded symbols. If the channel is bandlimited so that the transmitted symbol rate is fixed, this results in a lower information transmission rate. In the very common case that the high transmission rate is of interest (in contrast to minimizing transmission power), this reduction in effective information rate is unfortunate. Up until the early 1970s it was believed that coding would not greatly benefit channels needing a spectral efficiency — the number of bits transmitted per channel use — exceeding 1.

In 1976, a new method of coding was introduced by Ungerboeck [344, 346, 347, 345] which adds redundancy to the coded signal by increasing the number of symbols in the signal constellation employed in the modulation. If the average signal energy is fixed, having more signals in the signal constellation would tend to decrease the distance between points in the signal constellation. The key, therefore, is to combine the coding and modulation into a single unit which transmits only constrained *sequences* of symbols, and to employ a sequence detector (i.e., Viterbi algorithm) to detect the sequence. The combination of constrained symbol sequences and larger signal constellation gives rise to what is known as *trellis coded modulation*, or TCM.

13.2 Background on Signal Constellations

Because TCM is built upon signal constellations, we briefly review concepts related to signal constellations. For now, we restrict our attention to one- and two-dimensional signal constellations. (A review of the communications concepts in Section 1.4 may prove helpful.)

A *signal constellation* $\mathcal{S}$ is a discrete set of points, typically a subset of the real line $\mathbb{R}$ or the plane $\mathbb{R}^2$ (sometimes regarded as the complex plane). A one-dimensional constellation is used in what is often called *amplitude shift keying* (ASK). A one-dimensional constellation with two points $\pm\sqrt{E_b}$ is more frequently called BPSK (binary phase-shift keying). A two-dimensional constellation with all the points lying on a circle is referred as phase-shift keying (PSK). The constellation is frequently expressed in terms of the number of points, such as 4-PSK, 8-PSK, or 16-PSK. QPSK — quaternary PSK — is a synonym for 4-PSK. Figure 13.1 shows examples of PSK constellations, scaled so that they all have the same signal energy E_s. The minimum distance between signal points is denoted as d_0. A two-dimensional constellation with points on a square grid is frequently referred to as quadrature-amplitude modulation (QAM). Figure 13.2 shows overlays of examples of QAM constellations, where the minimum distance between points is called d_0. (The 32-point and 128-point constellations are referred to as cross constellations, since the points are arranged in a cross; this reduces the average energy compared to rectangular constellations.) Other