MIT 6.02 **DRAFT** *Lecture Notes*
Last update: February 24, 2012
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 7
Convolutional Codes: Construction and Encoding

This chapter introduces a widely used class of codes, called **convolutional codes**, which are used in a variety of systems including today's popular wireless standards (such as 802.11) and in satellite communications. They are also used as a building block in more powerful modern codes, such as turbo codes, which are used in wide-area cellular wireless network standards such as 3G, LTE, and 4G. Convolutional codes are beautiful because they are intuitive, one can understand them in many different ways, and there is a way to decode them so as to recover the *most likely* message from among the set of all possible transmitted messages. This chapter discusses the encoding of convolutional codes; the next one discusses how to decode convolutional codes efficiently.

Like the block codes discussed in the previous chapter, convolutional codes involve the computation of parity bits from message bits and their transmission, and they are also linear codes. Unlike block codes in systematic form, however, the sender does not send the message bits followed by (or interspersed with) the parity bits; in a convolutional code, the sender *sends only the parity bits*. These codes were invented by Peter Elias '44, an MIT EECS faculty member, in the mid-1950s. For several years, it was not known just how powerful these codes are and how best to decode them. The answers to these questions started emerging in the 1960s, with the work of people like Andrew Viterbi '57, G. David Forney (SM '65, Sc.D. '67, and MIT EECS faculty member), Jim Omura SB '63, and many others.

■ 7.1 Convolutional Code Construction

The encoder uses a *sliding window* to calculate $r > 1$ parity bits by combining various subsets of bits in the window. The combining is a simple addition in $\mathbb{F}_2$, as in the previous chapter (i.e., modulo 2 addition, or equivalently, an exclusive-or operation). Unlike a block code, however, the windows overlap and slide by 1, as shown in Figure 7-1. The size of the window, in bits, is called the code's **constraint length**. The longer the constraint length, the larger the number of parity bits that are influenced by any given message bit. Because the parity bits are the only bits sent over the channel, a larger constraint length generally

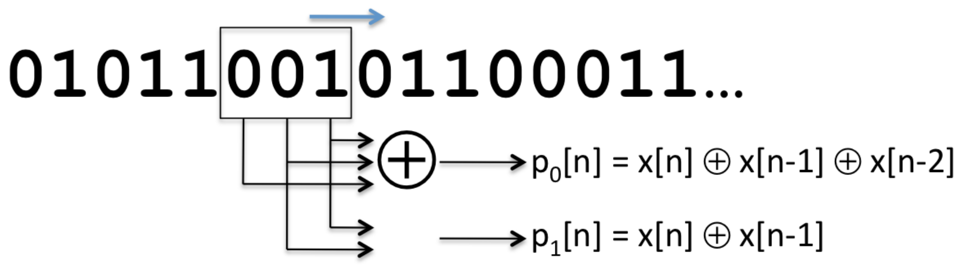

0101100101100011...
$p_0[n] = x[n] \oplus x[n-1] \oplus x[n-2]$
$p_1[n] = x[n] \oplus x[n-1]$

Figure 7-1: An example of a convolutional code with two parity bits per message bit and a constraint length (shown in the rectangular window) of three. I.e., $r = 2, K = 3$**.**

implies a greater resilience to bit errors. The trade-off, though, is that it will take considerably longer to decode codes of long constraint length (we will see in the next chapter that the complexity of decoding is exponential in the constraint length), so one cannot increase the constraint length arbitrarily and expect fast decoding.

If a convolutional code produces r parity bits per window and slides the window forward by one bit at a time, its rate (when calculated over long messages) is $1/r$. The greater the value of r, the higher the resilience of bit errors, but the trade-off is that a proportionally higher amount of communication bandwidth is devoted to coding overhead. In practice, we would like to pick r and the constraint length to be as small as possible while providing a low enough resulting probability of a bit error.

In 6.02, we will use K (upper case) to refer to the constraint length, a somewhat unfortunate choice because we have used k (lower case) in previous chapters to refer to the number of message bits that get encoded to produce coded bits. Although "L" might be a better way to refer to the constraint length, we'll use K because many papers and documents in the field use K (in fact, many papers use k in lower case, which is especially confusing). Because we will rarely refer to a "block" of size k while talking about convolutional codes, we hope that this notation won't cause confusion.

Armed with this notation, we can describe the encoding process succinctly. The encoder looks at K bits at a time and produces r parity bits according to carefully chosen functions that operate over various subsets of the K bits.[1] One example is shown in Figure 7-1, which shows a scheme with $K = 3$ and $r = 2$ (the rate of this code, $1/r = 1/2$). The encoder spits out r bits, which are sent sequentially, slides the window by 1 to the right, and then repeats the process. That's essentially it.

At the transmitter, the two princial remaining details that we must describe are:

1. What are good parity functions and how can we represent them conveniently?
2. How can we implement the encoder efficiently?

The rest of this chapter will discuss these issues, and also explain why these codes are called "convolutional".

[1]By convention, we will assume that each message has $K - 1$ "0" bits padded in front, so that the initial conditions work out properly.

7.2 Parity Equations

The example in Figure 7-1 shows one example of a set of *parity equations*, which govern the way in which parity bits are produced from the sequence of message bits, X. In this example, the equations are as follows (all additions are in $\mathbb{F}_2$)):

$$\begin{aligned} p_0[n] &= x[n] + x[n-1] + x[n-2] \\ p_1[n] &= x[n] + x[n-1] \end{aligned} \tag{7.1}$$

The rate of this code is $1/2$.

An example of parity equations for a rate $1/3$ code is

$$\begin{aligned} p_0[n] &= x[n] + x[n-1] + x[n-2] \\ p_1[n] &= x[n] + x[n-1] \\ p_2[n] &= x[n] + x[n-2] \end{aligned} \tag{7.2}$$

In general, one can view each parity equation as being produced by combining the message bits, X, and a **generator polynomial**, g. In the first example above, the generator polynomial coefficients are $(1,1,1)$ and $(1,1,0)$, while in the second, they are $(1,1,1),(1,1,0)$, and $(1,0,1)$.

We denote by g_i the K-element generator polynomial for parity bit p_i. We can then write $p_i[n]$ as follows:

$$p_i[n] = (\sum_{j=0}^{k-1} g_i[j]x[n-j]) \bmod 2. \tag{7.3}$$

The form of the above equation is a *convolution* of g and x—hence the term "convolutional code". The number of generator polynomials is equal to the number of generated parity bits, r, in each sliding window. The rate of the code is $1/r$ if the encoder slides the window one bit at a time.

7.2.1 An Example

Let's consider the two generator polynomials of Equations 7.1 (Figure 7-1). Here, the generator polynomials are

$$\begin{aligned} g_0 &= 1,1,1 \\ g_1 &= 1,1,0 \end{aligned} \tag{7.4}$$

If the message sequence, $X = [1,0,1,1,\ldots]$ (as usual, $x[n] = 0\ \forall n < 0$), then the parity bits from Equations 7.1 work out to be

$$\begin{aligned}
p_0[0] &= (1+0+0) = 1\\
p_1[0] &= (1+0) = 1\\
p_0[1] &= (0+1+0) = 1\\
p_1[1] &= (0+1) = 1\\
p_0[2] &= (1+0+1) = 0\\
p_1[2] &= (1+0) = 1\\
p_0[3] &= (1+1+0) = 0\\
p_1[3] &= (1+1) = 0.
\end{aligned} \tag{7.5}$$

Therefore, the bits transmitted over the channel are $[1,1,1,1,0,0,0,0,\ldots]$.

There are several generator polynomials, but understanding how to construct good ones is outside the scope of 6.02. Some examples (found by J. Busgang) are shown in Table 7-1.

Constraint length	g_0	g_1
3	110	111
4	1101	1110
5	11010	11101
6	110101	111011
7	110101	110101
8	110111	1110011
9	110111	111001101
10	110111001	1110011001

Table 7-1: Examples of generator polynomials for rate $1/2$ convolutional codes with different constraint lengths.

■ 7.3 Two Views of the Convolutional Encoder

We now describe two views of the convolutional encoder, which we will find useful in better understanding convolutional codes and in implementing the encoding and decoding procedures. The first view is in terms of **shift registers**, where one can construct the mechanism using shift registers that are connected together. This view is useful in developing hardware encoders. The second is in terms of a **state machine**, which corresponds to a view of the encoder as a set of states with well-defined transitions between them. The state machine view will turn out to be extremely useful in figuring out how to decode a set of parity bits to reconstruct the original message bits.

■ 7.3.1 Shift-Register View

Figure 7-2 shows the same encoder as Figure 7-1 and Equations (7.1) in the form of a block diagram made up of shift registers. The $x[n-i]$ values (here there are two) are referred to

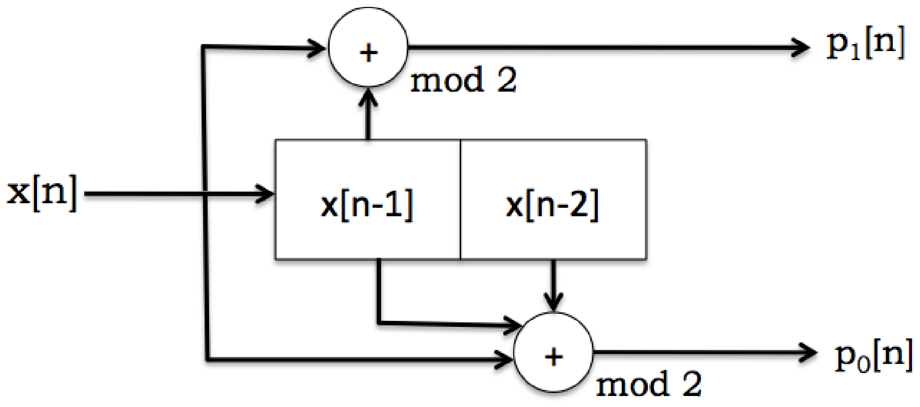


Figure 7-2: Block diagram view of convolutional coding with shift registers.

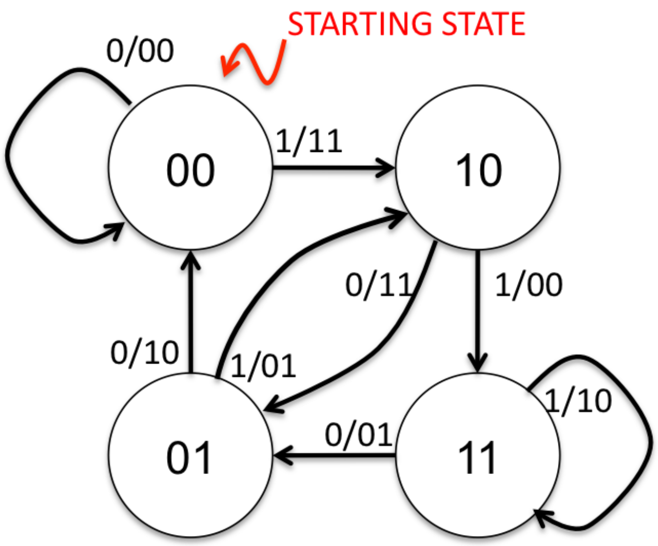


Figure 7-3: State-machine view of convolutional coding.

as the *state* of the encoder. This block diagram takes message bits in one bit at a time, and spits out parity bits (two per input bit, in this case).

Input message bits, $x[n]$, arrive from the left. The block diagram calculates the parity bits using the incoming bits and the state of the encoder (the $k-1$ previous bits; two in this example). After the r parity bits are produced, the state of the encoder shifts by 1, with $x[n]$ taking the place of $x[n-1]$, $x[n-1]$ taking the place of $x[n-2]$, and so on, with $x[n-K+1]$ being discarded. This block diagram is directly amenable to a hardware implementation using shift registers.

■ 7.3.2 State-Machine View

Another useful view of convolutional codes is as a state machine, which is shown in Figure 7-3 for the same example that we have used throughout this chapter (Figure 7-1).

An important point to note: the state machine for a convolutional code is *identical* for all codes with a given constraint length, K, and the number of states is always 2^{K-1}. Only the p_i labels change depending on the number of generator polynomials and the values of their coefficients. Each state is labeled with $x[n-1]x[n-2]\dots x[n-K+1]$. Each arc is labeled with $x[n]/p_0p_1\dots$. In this example, if the message is 101100, the transmitted bits are 11 11 01 00 01 10.

This state-machine view is an elegant way to explain what the transmitter does, and also what the receiver ought to do to decode the message, as we now explain. The transmitter begins in the initial state (labeled "STARTING STATE" in Figure 7-3) and processes the message one bit at a time. For each message bit, it makes the state transition from the current state to the new one depending on the value of the input bit, and sends the parity bits that are on the corresponding arc.

The receiver, of course, does not have direct knowledge of the transmitter's state transitions. It only sees the received sequence of parity bits, with possible bit errors. Its task is to determine the **best possible sequence of transmitter states that could have produced the parity bit sequence**. This task is the essence of the decoding process, which we introduce next, and study in more detail in the next chapter.

■ 7.4 The Decoding Problem

As mentioned above, the receiver should determine the "best possible" sequence of transmitter states. There are many ways of defining "best", but one that is especially appealing is the *most likely* sequence of states (i.e., message bits) that must have been traversed (sent) by the transmitter. A decoder that is able to infer the most likely sequence the *maximum-likelihood* (ML) decoder for the convolutional code.

In Section 6.2, we established that the ML decoder for "hard decoding", in which the distance between the received word and each valid codeword is the Hamming distance, may be found by computing the valid codeword with smallest Hamming distance, and returning the message that would have generated that codeword. The same idea holds for convolutional codes. (Note that this property holds whether the code is either block or convolutional, and whether it is linear or not.)

A simple numerical example may be useful. Suppose that bit errors are independent and identically distributed with an error probability of 0.001 (i.e., the channel is a BSC with $\varepsilon = 0.001$), and that the receiver digitizes a sequence of analog samples into the bits 1101001. Is the sender more likely to have sent 1100111 or 1100001? The first has a Hamming distance of 3, and the probability of receiving that sequence is $(0.999)^4(0.001)^3 = 9.9 \times 10^{-10}$. The second choice has a Hamming distance of 1 and a probability of $(0.999)^6(0.001)^1 = 9.9 \times 10^{-4}$, which is *six orders of magnitude higher* and is overwhelmingly more likely.

Thus, the most likely sequence of parity bits that was transmitted must be the one with the smallest Hamming distance from the sequence of parity bits received. Given a choice of possible transmitted messages, the decoder should pick the one with the smallest such Hamming distance. For example, see Figure 7-4, which shows a convolutional code with $K = 3$ and rate $1/2$. If the receiver gets 111011000110, then some errors have occurred, because no valid transmitted sequence matches the received one. The last column in the example shows d, the Hamming distance to all the possible transmitted sequences, with

Msg	*Xmit**	*Rcvd*	*d*
0000	000000000000	111011000110	7
0001	000000111110		8
0010	000011111000		8
0011	000011010110		4
0100	001111100000		6
0101	001111011110		5
0110	001101001000		7
0111	001100100110		6
1000	111110000000		4
1001	111110111110		5
1010	111101111000		7
1011	111101000110		2
1100	110001100000		5
1101	110001011110		4
1110	110010011000		6
1111	110010100110		3

Most likely: 1011

Figure 7-4: When the probability of bit error is less than 1/2, maximum-likelihood decoding boils down to finding the message whose parity bit sequence, when transmitted, has the smallest Hamming distance to the received sequence. Ties may be broken arbitrarily. Unfortunately, for an N-bit transmit sequence, there are 2^N possibilities, which makes it hugely intractable to simply go through in sequence because of the sheer number. For instance, when $N = 256$ bits (a really small packet), the number of possibilities rivals the number of atoms in the universe!

the smallest one circled. To determine the most-likely 4-bit message that led to the parity sequence received, the receiver could look for the message whose transmitted parity bits have smallest Hamming distance from the received bits. (If there are ties for the smallest, we can break them arbitrarily, because all these possibilities have the same resulting post-coded BER.)

Determining the nearest valid codeword to a received word is easier said than done for convolutional codes. For block codes, we found that comparing against each valid codeword would take time exponential in k, the number of valid codewords for an (n, k) block code. We then showed how syndrome decoding takes advantage of the linearity property to devise an efficient polynomial-time decoder for block codes, whose time complexity was roughly $O(n^t)$, where t is the number of errors that the linear block code can correct.

For convolutional codes, syndrome decoding in the form we described is impossible because n is *infinite* (or at least as long as the number of parity streams times the length of the entire message times, which could be arbitrarily long)! The straightforward approach

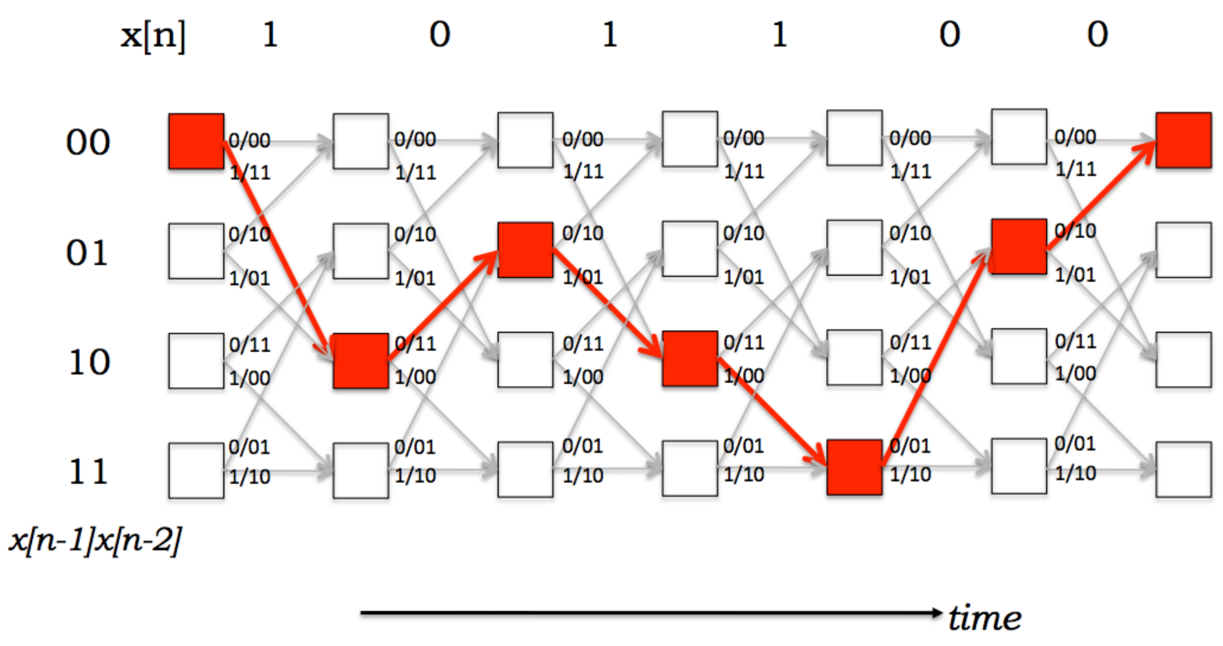


Figure 7-5: The trellis is a convenient way of viewing the decoding task and understanding the time evolution of the state machine.

of simply going through the list of possible transmit sequences and comparing Hamming distances is horribly intractable. We need a better plan for the receiver to navigate this unbelievable large space of possibilities and quickly determine the valid message with smallest Hamming distance. We will study a powerful and widely applicable method for solving this problem, called *Viterbi decoding*, in the next chapter. This decoding method uses a special structure called the **trellis**, which we describe next.

■ 7.5 The Trellis

The trellis is a structure derived from the state machine that will allow us to develop an efficient way to decode convolutional codes. The state machine view shows what happens at each instant when the sender has a message bit to process, but doesn't show how the system evolves in time. The *trellis* is a structure that makes the time evolution explicit. An example is shown in Figure 7-5. Each column of the trellis has the set of states; each state in a column is connected to two states in the next column—the same two states in the state diagram. The top link from each state in a column of the trellis shows what gets transmitted on a "0", while the bottom shows what gets transmitted on a "1". The picture shows the links between states that are traversed in the trellis given the message 101100.

We can now think about what the decoder needs to do in terms of this trellis. It gets a sequence of parity bits, and needs to determine the best path through the trellis—that is, the sequence of states in the trellis that can explain the observed, and possibly corrupted, sequence of received parity bits.

The Viterbi decoder finds a **maximum-likelihood path** through the trellis. We will study it in the next chapter.

Problems and exercises on convolutional coding are at the end of the next chapter, after we discuss the decoding process.

MIT 6.02 **DRAFT** *Lecture Notes*
Last update: February 29, 2012
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 8
Viterbi Decoding of Convolutional Codes

This chapter describes an elegant and efficient method to decode convolutional codes, whose construction and encoding we described in the previous chapter. This decoding method avoids explicitly enumerating the 2^N possible combinations of N-bit parity bit sequences. This method was invented by Andrew Viterbi '57 and bears his name.

■ 8.1 The Problem

At the receiver, we have a sequence of voltage samples corresponding to the parity bits that the transmitter has sent. For simplicity, and without loss of generality, we will assume that the receiver picks a suitable sample for the bit, or averages the set of samples corresponding to a bit, digitizes that value to a "0" or "1" by comparing to the threshold voltage (the demapping step), and propagates that bit decision to the decoder.

Thus, we have a *received bit sequence*, which for a convolutionally-coded stream corresponds to the stream of parity bits. If we decode this received bit sequence with no other information from the receiver's sampling and demapper, then the decoding process is termed **hard-decision decoding** ("hard decoding"). If, instead (or in addition), the decoder is given the stream of voltage samples and uses that "analog" information (in digitized form, using an analog-to-digital conversion) in decoding the data, we term the process **soft-decision decoding** ("soft decoding").

The Viterbi decoder can be used in either case. Intuitively, because hard-decision decoding makes an early decision regarding whether a bit is 0 or 1, it throws away information in the digitizing process. It might make a wrong decision, especially for voltages near the threshold, introducing a greater number of bit errors in the received bit sequence. Although it still produces the most likely transmitted sequence *given* the received bit sequence, by introducing additional errors in the early digitization, the overall reduction in the probability of bit error will be smaller than with soft decision decoding. But it is conceptually easier to understand hard decoding, so we will start with that, before going on to soft decoding.

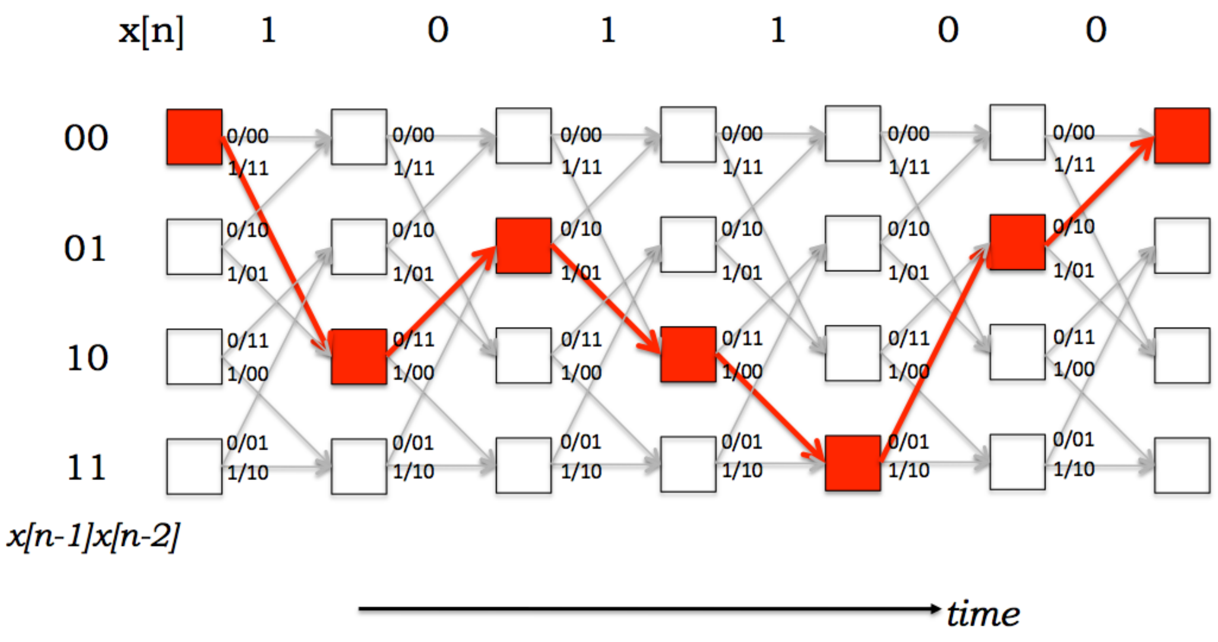


Figure 8-1: The trellis is a convenient way of viewing the decoding task and understanding the time evolution of the state machine.

As mentioned in the previous chapter, the trellis provides a good framework for understanding the decoding procedure for convolutional codes (Figure 8-1). Suppose we have the entire trellis in front of us for a code, and now receive a sequence of digitized bits (or voltage samples). If there are no errors, then there will be some path through the states of the trellis that would exactly match the received sequence. That path (specifically, the concatenation of the parity bits "spit out" on the traversed edges) corresponds to the transmitted parity bits. From there, getting to the original encoded message is easy because the top arc emanating from each node in the trellis corresponds to a "0" bit and the bottom arrow corresponds to a "1" bit.

When there are bit errors, what can we do? As explained earlier, finding the *most likely* transmitted message sequence is appealing because it minimizes the probability of a bit error in the decoding. If we can come up with a way to capture the errors introduced by going from one state to the next, then we can accumulate those errors along a path and come up with an estimate of the total number of errors along the path. Then, the path with the smallest such accumulation of errors is the path we want, and the transmitted message sequence can be easily determined by the concatenation of states explained above.

To solve this problem, we need a way to capture any errors that occur in going through the states of the trellis, and a way to navigate the trellis without actually materializing the entire trellis (i.e., without enumerating all possible paths through it and then finding the one with smallest accumulated error). The Viterbi decoder solves these problems. It is an example of a more general approach to solving optimization problems, called *dynamic programming*. Later in the course, we will apply similar concepts in network routing, an unrelated problem, to find good paths in multi-hop networks.

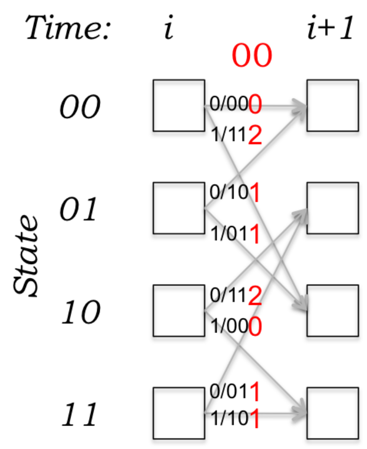


Figure 8-2: The branch metric for hard decision decoding. In this example, the receiver gets the parity bits 00.

■ 8.2 The Viterbi Decoder

The decoding algorithm uses two metrics: the **branch metric** (BM) and the **path metric** (PM). The branch metric is a measure of the "distance" between what was transmitted and what was received, and is defined for each arc in the trellis. In hard decision decoding, where we are given a sequence of digitized parity bits, the branch metric is the *Hamming distance* between the expected parity bits and the received ones. An example is shown in Figure 8-2, where the received bits are 00. For each state transition, the number on the arc shows the branch metric for that transition. Two of the branch metrics are 0, corresponding to the only states and transitions where the corresponding Hamming distance is 0. The other non-zero branch metrics correspond to cases when there are bit errors.

The path metric is a value associated with a state in the trellis (i.e., a value associated with each node). For hard decision decoding, it corresponds to the Hamming distance with respect to the received parity bit sequence over the most likely path from the initial state to the current state in the trellis. By "most likely", we mean the path with smallest Hamming distance between the initial state and the current state, measured over all possible paths between the two states. The path with the smallest Hamming distance minimizes the total number of bit errors, and is most likely when the BER is low.

The key insight in the Viterbi algorithm is that the receiver can compute the path metric for a (state, time) pair incrementally using the path metrics of previously computed states and the branch metrics.

■ 8.2.1 Computing the Path Metric

Suppose the receiver has computed the path metric $\text{PM}[s,i]$ for each state s at time step i (recall that there are 2^{K-1} states, where K is the constraint length of the convolutional code). In hard decision decoding, the value of $\text{PM}[s,i]$ is the total number of bit errors detected when comparing the received parity bits to the most likely transmitted message, considering all messages that could have been sent by the transmitter until time step i (starting from state "00", which we will take to be the starting state always, by convention).

Among all the possible states at time step i, the most likely state is the one with the smallest path metric. If there is more than one such state, they are all equally good possibilities.

Now, how do we determine the path metric at time step $i+1$, $\text{PM}[s, i+1]$, for each state s? To answer this question, first observe that if the transmitter is at state s at time step $i+1$, then *it must have been in only one of two possible states at time step i*. These two *predecessor states*, labeled α and β, are always the same for a given state. In fact, they depend only on the constraint length of the code and not on the parity functions. Figure 8-2 shows the predecessor states for each state (the other end of each arrow). For instance, for state 00, $\alpha = 00$ and $\beta = 01$; for state 01, $\alpha = 10$ and $\beta = 11$.

Any message sequence that leaves the transmitter in state s at time $i+1$ *must have* left the transmitter in state α or state β at time i. For example, in Figure 8-2, to arrive in state '01' at time $i+1$, one of the following two properties *must hold*:

1. The transmitter was in state '10' at time i and the i^{th} message bit was a 0. If that is the case, then the transmitter sent '11' as the parity bits and there were two bit errors, because we received the bits 00. Then, the path metric of the new state, PM['01', $i+1$] is equal to PM['10', i] + 2, because the new state is '01' and the corresponding path metric is larger by 2 because there are 2 errors.
2. The other (mutually exclusive) possibility is that the transmitter was in state '11' at time i and the i^{th} message bit was a 0. If that is the case, then the transmitter sent 01 as the parity bits and there was one bit error, because we received 00. The path metric of the new state, PM['01', $i+1$] is equal to PM['11', i] + 1.

Formalizing the above intuition, we can see that

$$\text{PM}[s, i+1] = \min(\text{PM}[\alpha, i] + \text{BM}[\alpha \rightarrow s], \text{PM}[\beta, i] + \text{BM}[\beta \rightarrow s]), \tag{8.1}$$

where α and β are the two predecessor states.

In the decoding algorithm, it is important to remember which arc corresponds to the minimum, because we need to traverse this path from the final state to the initial one keeping track of the arcs we used, and then finally *reverse* the order of the bits to produce the most likely message.

■ 8.2.2 Finding the Most Likely Path

We can now describe how the decoder finds the maximum-likelihood path. Initially, state '00' has a cost of 0 and the other $2^{K-1} - 1$ states have a cost of ∞.

The main loop of the algorithm consists of two main steps: first, calculating the branch metric for the next set of parity bits, and second, computing the path metric for the next column. The path metric computation may be thought of as an *add-compare-select* procedure:

1. *Add* the branch metric to the path metric for the old state.
2. *Compare* the sums for paths arriving at the new state (there are only two such paths to compare at each new state because there are only two incoming arcs from the previous column).
3. *Select* the path with the smallest value, breaking ties arbitrarily. This path corresponds to the one with fewest errors.

Figure 8-3 shows the decoding algorithm in action from one time step to the next. This example shows a received bit sequence of 11 10 11 00 01 10 and how the receiver processes it. The fourth picture from the top shows all four states with the same path metric. At this stage, any of these four states and the paths leading up to them are most likely transmitted bit sequences (they all have a Hamming distance of 2). The bottom-most picture shows the same situation with only the *survivor paths* shown. A survivor path is one that has a chance of being the maximum-likelihood path; there are many other paths that can be pruned away because there is no way in which they can be most likely. The reason why the Viterbi decoder is practical is that the number of survivor paths is much, much smaller than the total number of paths in the trellis.

Another important point about the Viterbi decoder is that *future knowledge* will help it break any ties, and in fact may even cause paths that were considered "most likely" at a certain time step to change. Figure 8-4 continues the example in Figure 8-3, proceeding until all the received parity bits are decoded to produce the most likely transmitted message, which has two bit errors.

■ 8.3 Soft-Decision Decoding

Hard decision decoding digitizes the received voltage signals by comparing it to a threshold, *before* passing it to the decoder. As a result, we lose information: if the voltage was 0.500001, the confidence in the digitization is surely much lower than if the voltage was 0.999999. Both are treated as "1", and the decoder now treats them the same way, even though it is overwhelmingly more likely that 0.999999 is a "1" compared to the other value.

Soft-decision decoding (also sometimes known as "soft input Viterbi decoding") builds on this observation. It *does not digitize the incoming samples prior to decoding*. Rather, it uses a continuous function of the analog sample as the input to the decoder. For example, if the expected parity bit is 0 and the received voltage is 0.3 V, we might use 0.3 (or 0.3^2, or some such function) as the value of the "bit" instead of digitizing it.

For technical reasons that will become apparent later, an attractive soft decision metric is the *square* of the difference between the received voltage and the expected one. If the convolutional code produces p parity bits, and the p corresponding analog samples are $v = v_1, v_2, \ldots, v_p$, one can construct a soft decision branch metric as follows

$$\text{BM}_{\text{soft}}[u, v] = \sum_{i=1}^{p} (u_i - v_i)^2, \tag{8.2}$$

where $u = u_1, u_2, \ldots, u_p$ are the *expected* p parity bits (each a 0 or 1). Figure 8-5 shows the soft decision branch metric for $p = 2$ when u is 00.

With soft decision decoding, the decoding algorithm is identical to the one previously described for hard decision decoding, except that the branch metric is no longer an integer Hamming distance but a positive real number (if the voltages are all between 0 and 1, then the branch metric is between 0 and 1 as well).

It turns out that this soft decision metric is closely related to the *probability of the decoding being correct* when the channel experiences additive Gaussian noise. First, let's look at the simple case of 1 parity bit (the more general case is a straightforward extension). Suppose the receiver gets the i^{th} parity bit as v_i volts. (In hard decision decoding, it would decode $-$ as 0 or 1 depending on whether v_i was smaller or larger than 0.5.) What is the probability that v_i would have been received given that bit u_i (either 0 or 1) was sent? With zero-mean additive Gaussian noise, the PDF of this event is given by

$$f(v_i|u_i) = \frac{e^{-d_i^2/2\sigma^2}}{\sqrt{2\pi\sigma^2}}, \tag{8.3}$$

where $d_i = v_i^2$ if $u_i = 0$ and $d_i = (v_i - 1)^2$ if $u_i = 1$.

The log likelihood of this PDF is proportional to $-d_i^2$. Moreover, along a path, the PDF of the sequence $V = v_1, v_2, \ldots, v_p$ being received given that a code word $U = u_i, u_2, \ldots, u_p$ was sent, is given by the product of a number of terms each resembling Eq. (8.3). The logarithm of this PDF for the path is equal to the sum of the individual log likelihoods, and is proportional to $-\sum_i d_i^2$. But that's precisely the negative of the branch metric we defined in Eq. (8.2), which the Viterbi decoder minimizes along the different possible paths! Minimizing this path metric is identical to maximizing the log likelihood along the different paths, implying that the soft decision decoder produces the most likely path that is consistent with the received voltage sequence.

This direct relationship with the logarithm of the probability is the reason why we chose the sum of squares as the branch metric in Eq. (8.2). A different noise distribution (other than Gaussian) may entail a different soft decoding branch metric to obtain an analogous connection to the PDF of a correct decoding.

■ 8.4 Achieving Higher and Finer-Grained Rates: Puncturing

As described thus far, a convolutional code achieves a maximum rate of $1/r$, where r is the number of parity bit streams produced by the code. But what if we want a rate greater than $1/2$, or a rate between $1/r$ and $1/(r+1)$ for some r?

A general technique called **puncturing** gives us a way to do that. The idea is straightforward: the encoder does not send every parity bit produced on each stream, but "punctures" the stream sending only a subset of the bits that are agreed-upon between the encoder and decoder. For example, one might use a rate-1/2 code along with the puncturing schedule specified as a vector; for example, we might use the vector (101) on the first parity stream and (110) on the second. This notation means that the encoder sends the first and third bits but not the second bit on the first stream, and sends the first and second bits but not the third bit on the second stream. Thus, whereas the encoder would have sent two parity bits for every message bit without puncturing, it would now send four parity bits (instead of six) for every three message bits, giving a rate of 3/4.

In this example, suppose the sender in the rate-1/2 code, without puncturing, emitted bits $p_0[0]p_1[0]p_0[1]p_1[1]p_0[2]p_1[2]\ldots$. Then, with the puncturing schedule given, the bits emitted would be $p_0[0]p_1[0] - p_1[1]p_0[2] - \ldots$, where each $-$ refers to an omitted bit.

At the decoder, when using a punctured code, missing parity bits don't participate in the calculation of branch metrics. Otherwise, the procedure is the same as before. We can think of each missing parity bit as a blank ('$-$') and run the decoder by just skipping over the blanks.

■ 8.5 Encoder and Decoder Implementation Complexity

There are two important questions we must answer concerning the time and space complexity of the convolutional encoder and Viterbi decoder.

1. How much state and space does the encoder need?
2. How much time does the decoder take?

The first question is easy to answer: at the encoder, the amount of space is linear in K, the constraint length; the time required is linear in the message length, n. The encoder is much easier to implement than the Viterbi decoder. The decoding time depends both on K and the length of the coded (parity) bit stream (which is linear in n). At each time step, the decoder must compare the branch metrics over two state transitions into each state, for each of $2^(K-1)$ states. The number of comparisons required is 2^K in each step, giving us a total time complexity of $O(n \cdot 2^K)$ for decoding an n-bit message.

Moreover, as described thus far, we can decode the first bits of the message only at the very end. A little thought will show that although a little future knowledge is useful, it is unlikely that what happens at bit time 1000 will change our decoding decision for bit 1, if the constraint length is, say, 6. In fact, in practice the decoder starts to decode bits once it has reached a time step that is a small multiple of the constraint length; experimental data suggests that $5 \cdot K$ message bit times (or thereabouts) is a reasonable decoding window, regardless of how long the parity bit stream corresponding to the message is.

■ 8.6 Designing Good Convolutional Codes

At this stage, a natural question one might wonder about is, "What makes a set of parity equations a good convolutional code?" In other words, is there a systematic method to *generate* good convolutional codes? Or, given two convolutional codes, is there a way to analyze their generators and determine how they might perform relative to each other in their primary task, which is to enable communication over a noisy channel at as high a rate as they can?

In principle, many factors determine the effectiveness of a convolutional code. One would expect the ability of a convolutional code to correct errors depends on the constraint length, K, because the larger the constraint length, the greater the degree to which any given message bit contributes to some parity bit, and the greater the resilience to bit errors. One would also expect the resilience to errors to be higher as the number of generators (parity streams) increases, because that corresponds to a lower rate (more redundancy). And last but not least, the coefficients of the generators surely have a role to play in determining the code's effectiveness.

Fortunately, there is one metric, called the **free distance** of the convolutional code, which captures these different axes and is a primary determinant of the error-reducing capability of a convolutional code, when hard-decision decoding is used.

■ 8.6.1 Free Distance

Because convolutional codes are linear, everything we learned about linear codes applies here. In particular, the Hamming distance of any linear code, i.e., the minimum Hamming distance between any two valid codewords, is equal to the *number of ones in the smallest non-zero codeword with minimum weight*, where the weight of a codeword is the number of ones it contains.

In the context of convolutional codes, the smallest Hamming distance between any two valid codewords is called the *free distance*. Specifically, the free distance of a convolutional code is the difference in path metrics between the all-zeroes output and the path with the smallest non-zero path metric going from the initial 00 state to some future 00 state. Figure 8-6 illustrates this notion with an example. In this example, the free distance is 4, and it takes 8 output bits to get back to the correct state, so one would expect this code to be able to correct up to $\lfloor (4-1)/2 \rfloor = 1$ bit error in blocks of 8 bits, if the block starts at the first parity bit. In fact, this error correction power is essentially the same as an $(8, 4, 3)$ rectangular parity code. Note that the free distance in this example is 4, not 5: the smallest non-zero path metric between the initial 00 state and a future 00 state goes like this: $00 \rightarrow 10 \rightarrow 11 \rightarrow 01 \rightarrow 00$ and the corresponding path metrics increase as $0 \rightarrow 2 \rightarrow 2 \rightarrow 3 \rightarrow 4$. In the next section, we will find that a small change to the generator—replacing 110 with 101—makes a huge difference in the performance of the code.

Why do we define a "free distance", rather than just call it the Hamming distance, if it is defined the same way? The reason is that any code with Hamming distance D (whether linear or not) can correct all patterns of up to $\lfloor \frac{D-1}{2} \rfloor$ errors. If we just applied the same notion to convolutional codes, we will conclude that we can correct all single-bit errors in the example given, or in general, we can correct some fixed number of errors.

Now, convolutional coding produces an unbounded bit stream; these codes are markedly distinct from block codes in this regard. As a result, the $\lfloor \frac{D-1}{2} \rfloor$ formula is not too instructive because it doesn't capture the true error correction properties of the code. A convolutional code (with Viterbi decoding) can correct $t = \lfloor \frac{D-1}{2} \rfloor$ errors as long as these errors are "far enough apart". So the notion we use is the free distance because, in a sense, errors can keep occurring and as long as no more than t of them occur in a closely spaced burst, the decoder can correct them all.

■ 8.6.2 Selecting Good Convolutional Codes

The free distance concept also provides a way to construct good convolutional codes. Given a decoding budget (e.g., hardware resources), one first determines an appropriate bound on K. Then, one picks an upper bound on r depending on the maximum rate. Given a specific K and r, there is a finite number of generators that are feasible. One can write a program to exhaustively go through all feasible combinations of generators, compute the free distance, and pick the code (or codes) with the largest free distance. The convolutional code is specified completely by specifying the generators (both K and r are implied if one lists the set of generators).

■ 8.7 Comparing the Error-Correction Performance of Codes

This section discusses how to compare the error-correction performance of different codes and discusses simulation results obtained by implementing different codes and evaluating them under controlled conditions. We have two goals in this section: first, to describe the "best practices" in comparing codes and discuss common pitfalls, and second, to compare some specific convolutional and block codes and discuss the reasons why some codes perform better than others.

There are two metrics of interest. The first is the *bit error rate* (BER) *after decoding*, which is sometimes also known as the probability of decoding error. The second is the *rate* achieved by the code. For both metrics, we are interested in how they vary as a function of the channel's parameters, such as the value of ε in a BSC (i.e., the channel's underlying bit error probability) or the degree of noise on the channel (for a channel with additive Gaussian noise, which we will describe in detail in the next chapter).

Here, we focus only on the post-decoding BER of a code.

■ 8.7.1 Post-decoding BER over the BSC

For the BSC, the variable is ε, and one can ask how different codes perform (in terms of the BER) as we vary ε. Figure 8-7 shows the post-decoding BER of a few different linear block codes and convolutional codes as a function of the BSC error rate, ε. From this graph, it would appear that the rate-1/3 repetition code $(3, 1)$ with a Hamming distance of 3 is the most robust code at high BSC error probabilities (right-side of the picture), and that the two rate-1/2 convolutional codes are very good ones at other BERs. It would also appear from this curve that the $(7, 4)$ and $(15, 11)$ Hamming codes are inferior to the other codes.

The problem with these conclusions is that they don't take the *rate* of the code into account; some of these codes incur much higher overhead than the others. As such, on a curve such as Figure 8-7 that plots the post-decoding BER against the BSC error probability, it is sensible only to compare codes of the same rate. Thus, one can compare the $(8, 4)$ block code to the three other convolutional code, and form the following conclusions:

1. The two best convolutional codes, $(3,(7, 5))$ (i.e., with generators $(111, 101)$) and $(4,(14, 13))$ (i.e., with generators $(1110, 1101)$), perform the best. Both these codes handily beat the third convolutional code, $(3,(7, 6))$, which we picked from Bussgang's paper on generating good convolutional codes.[1]

   The reason for the superior performance of the $(3,(7, 5))$ and $(4,(14, 13))$ codes is that they have a greater free distance (5 and 6 respectively) than the $(3,(7, 6))$ code (whose free distance is 4). The greater free distance allows for a larger number of closely-spaced errors to be corrected.

2. Interestingly, these results show that the $(3,(7, 5))$ code with free distance 5 is stronger than the $(4,(14, 13))$ code with free distance 6. The reason is that the number of trellis edges to go from state 00 back to state 00 in the $(3,(7, 5))$ case is only 3, corresponding to a group of 6 consecutive coded bits. The relevant state transitions are $00 \rightarrow 10 \rightarrow 01 \rightarrow 00$ and the corresponding path metrics are $0 \rightarrow 2 \rightarrow 3 \rightarrow 5$. In contrast, the $(1110, 1101)$ code has a slightly bigger free distance, but it takes 7 trellis edges to achieve that ($000 \rightarrow 100 \rightarrow 010 \rightarrow 001 \rightarrow 000$), meaning that the code can correct up to 2 bit errors in sliding windows of length $2 \cdot 4 = 8$ bits. Moreover, an increase in the free distance from 5 to 6 (an even number) does not improve the error-*correcting* power of the code.

[1]Julian Bussgang, "Some Properties of Binary Convolutional Code Generators," IEEE Transactions on Information Theory, pp. 90–100, Jan. 1965.

3. The post-decoding BER is roughly the same for the $(8, 4)$ rectangular parity code and the $(3, (111, 110))$ convolutional code. The reason is that the free distance of the $K = 3$ convolutional code is 4, which means it can correct one bit error over blocks that are similar in length to the rectangular parity code we are comparing with. Intuitively, both schemes essentially produce parity bits that are built from similar amounts of history. In the rectangular parity case, the row parity bit comes from two successive message bits, while the column parity comes from two message bits with one skipped in between. But we also send the message bits, so we're mimicking a similar constraint length (amount of memory) to the $K = 3$ convolutional code. The bottom line is that $(3, (111, 110))$ is not such a good convolutional code.

4. The $(7, 4)$ Hamming code performs similarly to the $(8, 4)$ rectangular parity code, but it has a higher code rate (4/7 versus 1/2), which means it provides the same correction capabilities with lower overhead. One may therefore conclude that it is a better code than the $(8, 4)$ rectangular parity code.

But how does one go about comparing the post-decoding BER of codes with different rates? We need a way to capture the different amounts of redundancy exhibited by codes of different rates. To do that, we need to change the model to account for what happens at the physical (analog) level. A standard way of handling this issue is to use the **signal-to-noise ratio (SNR)** as the control variable (on the x-axis) and introduce **Gaussian noise** to perturb the signals sent over the channel. The next chapter studies this noise model in detail, but here we describe the basic intuition and results obtained when comparing the performance of codes under this model. This model is also essential to understand the benefits of soft-decision decoding, because soft decoding uses the received voltage samples directly as input to the decoder without first digitizing each sample. The question is how much gain we observe by doing soft-decision decoding compared to hard-decision decoding.

■ 8.7.2 Gaussian Noise Model and the E_b/N_0 Concept

Consider a message k bits long. We have two codes: C_1 has rate k/n_1 and C_2 has rate k/n_2, and suppose $n_2 > n_1$. Hence, for the k-bit message, when encoded with C_1, we transmit n_1 bits, and when encoded with C_2, we transmit n_2 bits. Clearly, using C_2 consumes more resources because it uses the channel more often than C_1.

An elegant way to account for the greater resource consumption of C_1 is to run an experiment where each "1" bit is mapped to a certain voltage level, V_1, and each "0" is mapped to a voltage V_0. For reasons that will become apparent in the next chapter, what matters for decoding is the difference in separation between the voltages, $V_1 - V_0$, and not their actual values, so we can assume that the two voltages are centered about 0. For convenience, assume $V_1 = \sqrt{E_s}$ and $V_0 = -\sqrt{E_s}$, where E_s is the *energy per sample*. The energy, or power, is proportional to the square of the voltage of used.

Now, when we use code C_1, k message bits get tranformed to n_1 coded bits. Assuming that each coded bit is sent as one voltage sample (for simplicity), the *energy per bit* is equal to $n_1/k \cdot E_s$. Similarly, for code C_2, it is equal to $n_2/k \cdot E_s$. Each voltage sample in the additive Gaussian noise channel model (see the next chapter) is perturbed according to a Gaussian distribution with some variance; the variance is the amount of noise (the greater the variance, the greater the noise, and the greater the bit-error probability of the equivalent BSC). Hence, the correct "scaled" x-axis for comparing the post-decoding BER of codes of different rates is E_b/N_0, the ratio of the energy-per-message-bit to the channel Gaussian noise.

Figure 8-8 shows some representative performance results of experiments done over a simulated Gaussian channel for different values of E_b/N_0. Each data point in the experiment is the result of simulating about 2 million message bits being encoded and transmitted over a noisy channel. The top-most curve shows the uncoded probability of bit error. The x axis plots the E_b/N_0 on the decibel (dB) scale, defined in Chapter 9 (lower noise is toward the right). The y axis shows the probability of a decoding error on a *log scale*.

Some observations from these results are noteworthy:

1. Good convolutional codes are noticeably superior to the Hamming and rectangular parity codes.

2. Soft-decision decoding is a significant win over hard-decision decoding; for the same post-decoding BER, soft decoding has a 2 *to* 2.3 *db gain*; i.e., with hard decoding, you would have to increase the signal-to-noise ratio by that amount (which is a factor of $1.6\times$, as explained in Chapter 9) to achieve the same post-decoding BER.

■ 8.8 Summary

From its relatively modest, though hugely impactful, beginnings as a method to decode convolutional codes, Viterbi decoding has become one of the most widely used algorithms in a wide range of fields and engineering systems. Modern disk drives with "PRML" technology to speed-up accesses, speech recognition systems, natural language systems, and a variety of communication networks use this scheme or its variants.

In fact, a more modern view of the soft decision decoding technique described in this lecture is to think of the procedure as finding the most likely set of traversed states in a *Hidden Markov Model* (HMM). Some underlying phenomenon is modeled as a Markov state machine with probabilistic transitions between its states; we see noisy observations from each state, and would like to piece together the observations to determine the most likely sequence of states traversed. It turns out that the Viterbi decoder is an excellent starting point to solve this class of problems (and sometimes the complete solution).

On the other hand, despite its undeniable success, Viterbi decoding isn't the only way to decode convolutional codes. For one thing, its computational complexity is exponential in the constraint length, K, because it does require each of these states to be enumerated. When K is large, one may use other decoding methods such as BCJR or Fano's sequential decoding scheme, for instance.

Convolutional codes themselves are very popular over both wired and wireless links. They are sometimes used as the "inner code" with an outer block error correcting code, but they may also be used with just an outer error detection code. They are also used as a component in more powerful codes like turbo codes, which are currently one of the highest-performing codes used in practice.

Problems and Exercises

1. Please check out and solve the online problems on error correction codes at `http://web.mit.edu/6.02/www/currentsemester/handouts/tutprobs/ecc.html`

2. Consider a convolutional code whose parity equations are

$$\begin{aligned} p_0[n] &= x[n] + x[n-1] + x[n-3] \\ p_1[n] &= x[n] + x[n-1] + x[n-2] \\ p_2[n] &= x[n] + x[n-2] + x[n-3] \end{aligned}$$

   (a) What is the rate of this code? How many states are in the state machine representation of this code?

   (b) Suppose the decoder reaches the state "110" during the forward pass of the Viterbi algorithm with this convolutional code.

      i. How many predecessor states (i.e., immediately preceding states) does state "110" have?

      ii. What are the bit-sequence representations of the predecessor states of state "110"?

      iii. What are the expected parity bits for the transitions from each of these predecessor states to state "110"? Specify each predecessor state and the expected parity bits associated with the corresponding transition below.

   (c) To increase the rate of the given code, Lem E. Tweakit punctures the p_0 parity stream using the vector (1 0 1 1 0), which means that every second and fifth bit produced on the stream are *not sent*. In addition, she punctures the p_1 parity stream using the vector (1 1 0 1 1). She sends the p_2 parity stream unchanged. What is the rate of the punctured code?

3. Let `conv_encode(x)` be the resulting bit-stream after encoding bit-string x with a convolutional code, C. Similarly, let `conv_decode(y)` be the result of decoding y to produce the maximum-likelihood estimate of the encoded message. Suppose we send a message M using code C over some channel. Let P = `conv_encode(M)` and let R be the result of sending P over the channel and digitizing the received samples at the receiver (i.e., R is another bit-stream). Suppose we use Viterbi decoding on R, knowing C, and find that the maximum-likelihood estimate of M is $\hat{M}$. During the decoding, we find that the minimum path metric among all the states in the final stage of the trellis is D_{min}.

   D_{min} is the Hamming distance between __________ and __________. Fill in the blanks, explaining your answer.

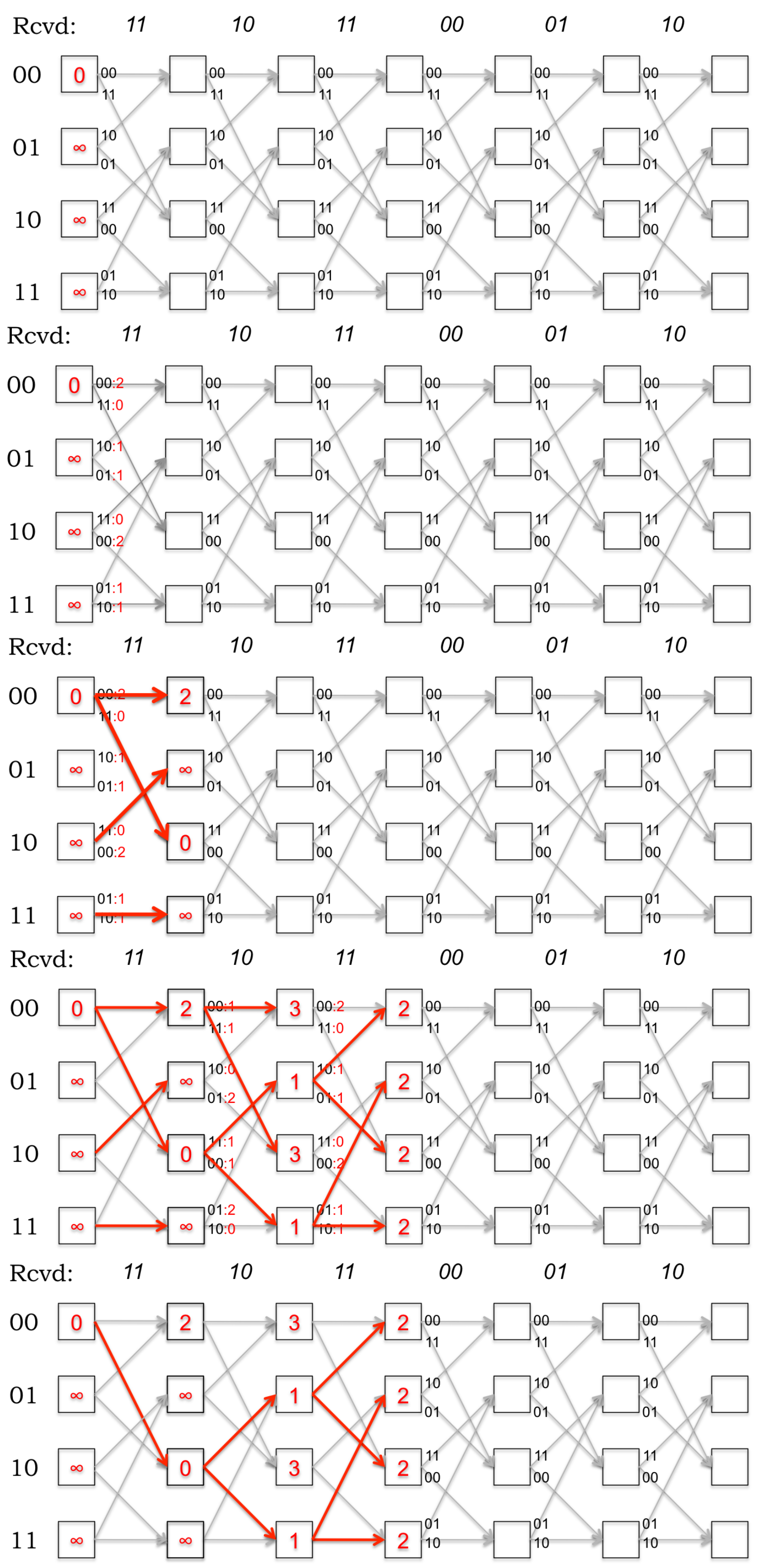


Figure 8-3: The Viterbi decoder in action. This picture shows four time steps. The bottom-most picture is the same as the one just before it, but with only the survivor paths shown.

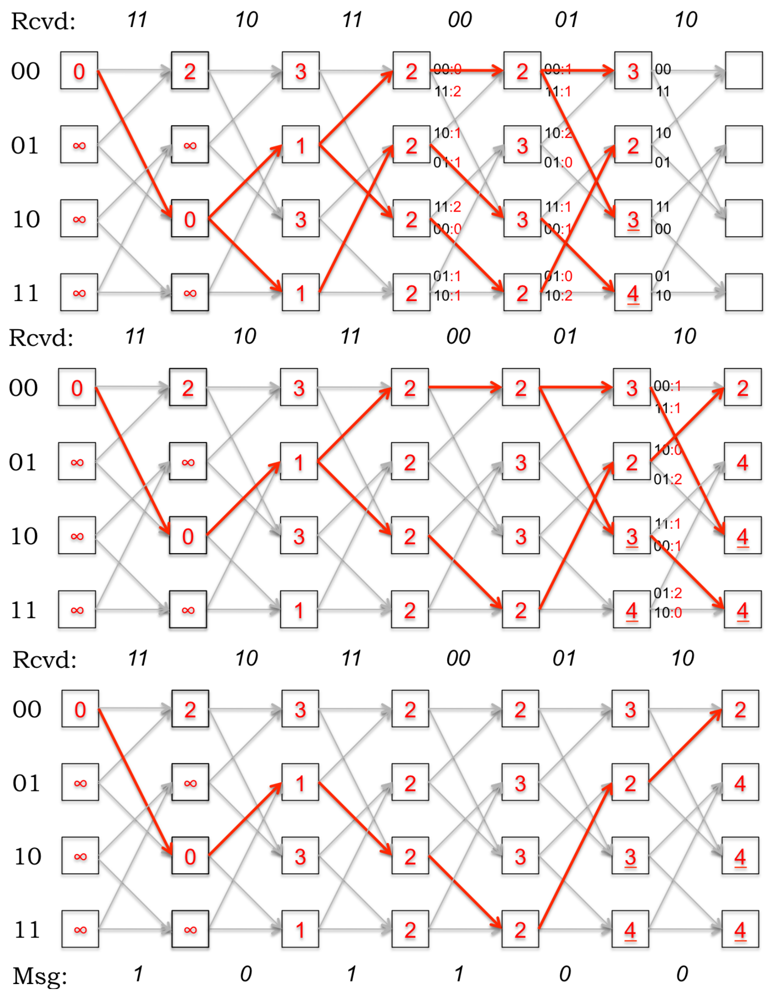


Figure 8-4: The Viterbi decoder in action (continued from Figure 8-3. The decoded message is shown. To produce this message, start from the final state with smallest path metric and work backwards, and then reverse the bits. At each state during the forward pass, it is important to remeber the arc that got us to this state, so that the backward pass can be done properly.

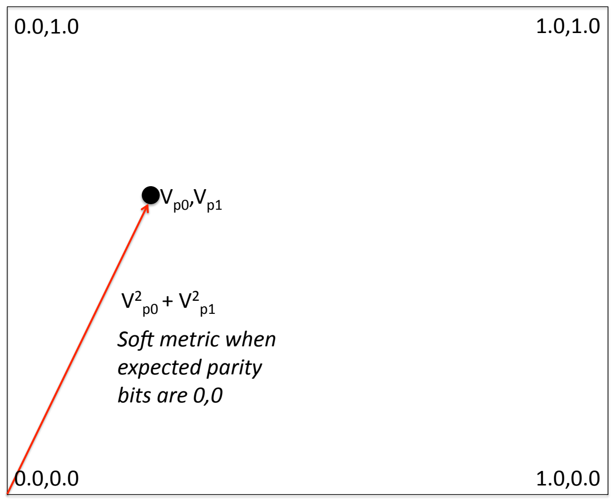
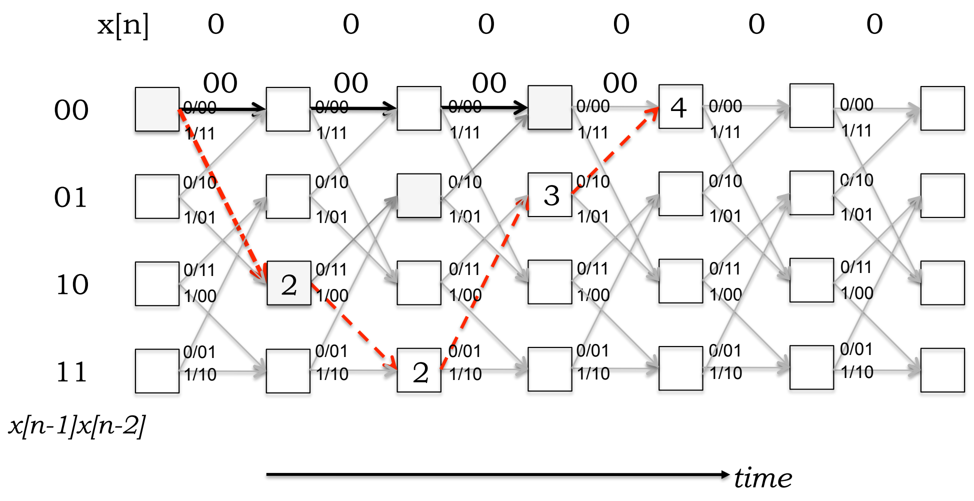


Figure 8-5: Branch metric for soft decision decoding.



The free distance is the difference in path metrics between the all-zeroes output and the path with the smallest non-zero path metric going from the initial 00 state to some future 00 state. It is 4 in this example. The path 00 → 10 →01 → 00 has a shorter length, but a higher path metric (of 5), so it is not the free distance.

Figure 8-6: The free distance of a convolutional code.

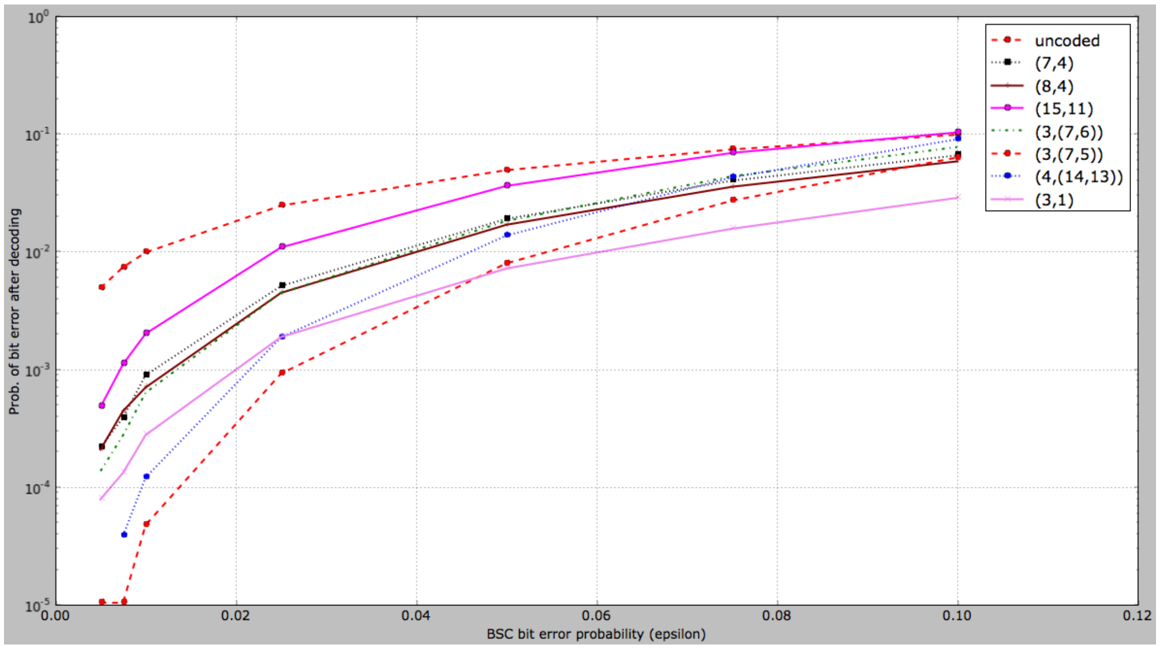


Figure 8-7: Post-decoding BER v. BSC error probability ε for different codes. Note that not all codes have the same rate, so this comparison is misleading. One should only compare curves of the same rate on a BER v. BSC error probability curve such as this one; comparisons between codes of different rates on the x-axis given aren't meaningful because they don't account for the different overhead amounts.

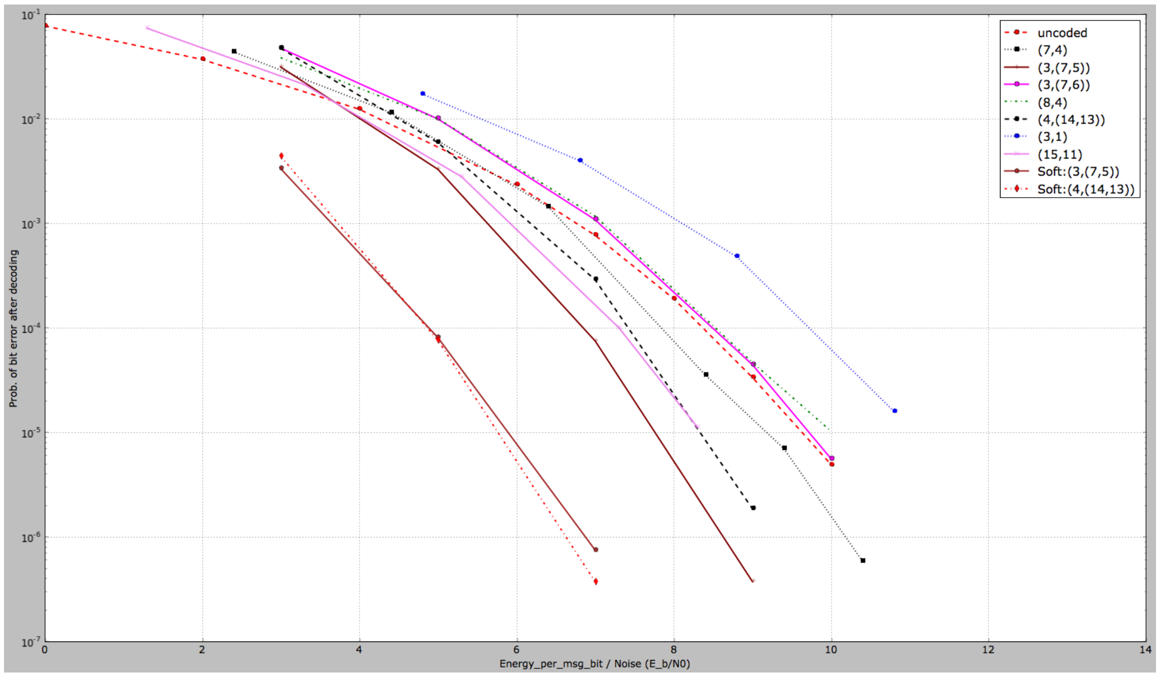


Figure 8-8: Post-decoding BER of a few different linear block codes and convolutional codes as a function of E_b/N_0 in the additive Gaussian noise channel model.