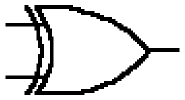
Convolutional (Viterbi) Encoding

Convolutional encoding of data is accomplished using a shift register and associated combinatorial logic that performs modulo-two addition. (A shift register is merely a chain of flip-flops wherein the output of the nth flip-flop is tied to the input of the (n+1)th flip-flop. Every time the active edge of the clock occurs, the input to the flip-flop is clocked through to the output, and thus the data are shifted over one stage.) The combinatorial logic is often in the form of cascaded exclusive-or gates. As a reminder, exclusive-or gates are two-input, one-output gates often represented by the logic symbol shown below,

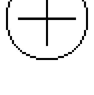


that implement the following truth-table:

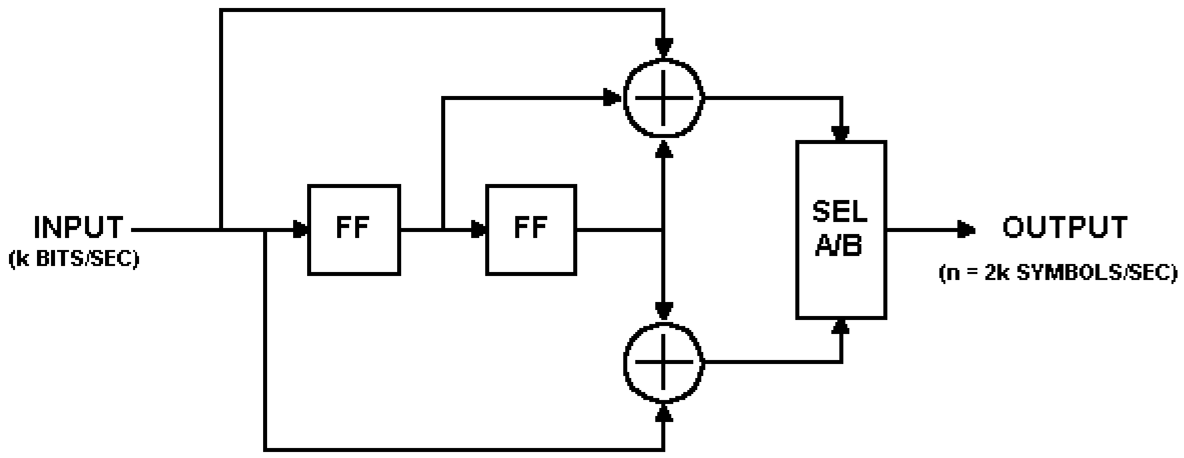
Input A	Input B	Output (A xor B)
0	0	0
0	1	1
1	0	1
1	1	0

The exclusive-or gate performs modulo-two addition of its inputs. When you cascade q two-input exclusive-or gates, with the output of the first one feeding one of the inputs of the second one, the output of the second one feeding one of the inputs of the third one, etc., the output of the last one in the chain is the modulo-two sum of the q + 1 inputs.

Another way to illustrate the modulo-two adder, and the way that is most commonly used in textbooks, is as a circle with a + symbol inside, thus:



Now that we have the two basic components of the convolutional encoder (flip-flops comprising the shift register and exclusive-or gates comprising the associated modulo-two adders) defined, let's look at a picture of a convolutional encoder for a rate 1/2, K = 3, m = 2 code:



In this encoder, data bits are provided at a rate of k bits per second. Channel symbols are output at a rate of n = 2k symbols per second. The input bit is stable during the encoder cycle. The encoder cycle starts when an input clock edge occurs. When the input clock edge occurs, the output of the left-hand flip-flop is clocked into the right-hand flip-flop, the previous input bit is clocked into the left-hand flip-flop, and a new input bit becomes available. Then the outputs of the upper and lower modulo-two adders become stable. The output selector (SEL A/B block) cycles through two states-in the first state, it selects and outputs the output of the upper modulo-two adder; in the second state, it selects and outputs the output of the lower modulo-two adder.

The encoder shown above encodes the K = 3, (7, 5) convolutional code. The octal numbers 7 and 5 represent the code generator polynomials, which when read in binary (111_2 and 101_2) correspond to the shift register connections to the upper and lower modulo-two adders, respectively. This code has been determined to be the "best" code for rate 1/2, K = 3. It is the code I will use for the remaining discussion and examples, for reasons that will become readily apparent when we get into the Viterbi decoder algorithm.

Let's look at an example input data stream, and the corresponding output data stream:

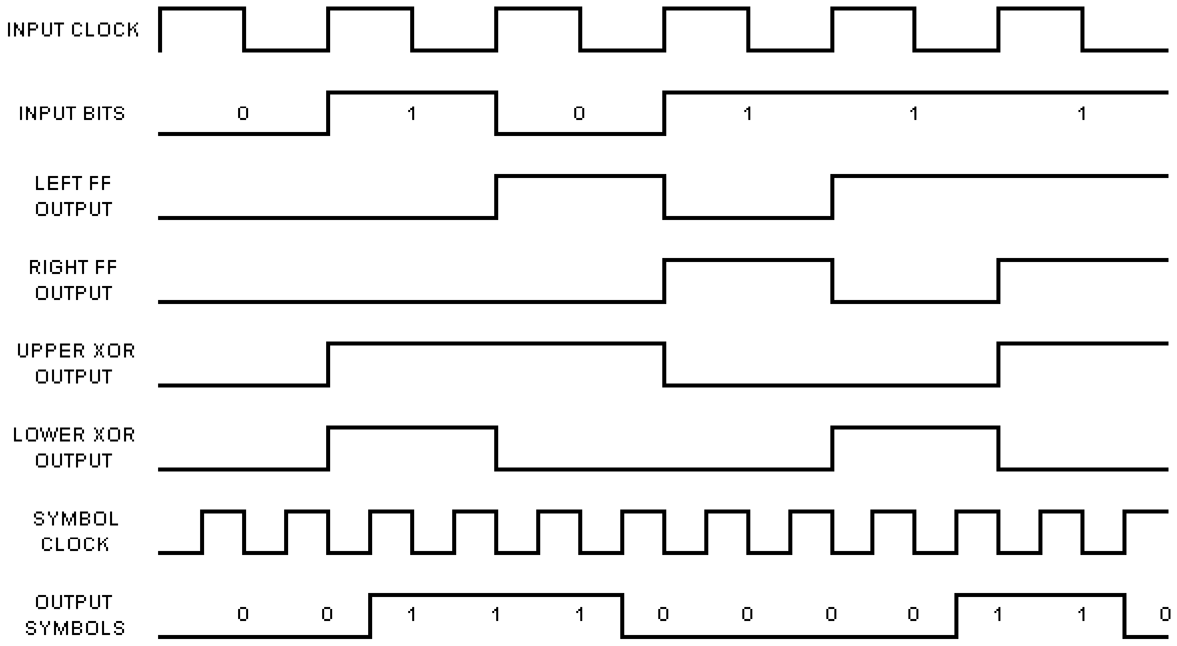
Let the input sequence be 010111001010001_2.

Assume that the outputs of both of the flip-flops in the shift register are initially cleared, i.e. their outputs are zeroes. The first clock cycle makes the first input bit, a zero, available to the encoder. The flip-flop outputs are both zeroes. The inputs to the modulo-two adders are all zeroes, so the output of the encoder is 00_2.

The second clock cycle makes the second input bit available to the encoder. The left-hand flip-flop clocks in the previous bit, which was a zero, and the right-hand flip-flop clocks in the zero output by the left-hand flip-flop. The inputs to the top modulo-two adder are 100_2, so the output is a one. The inputs to the bottom modulo-two adder are 10_2, so the output is also a one. So the encoder outputs 11_2 for the channel symbols.

The third clock cycle makes the third input bit, a zero, available to the encoder. The left-hand flip-flop clocks in the previous bit, which was a one, and the right-hand flip-flop clocks in the zero from two bit-times ago. The inputs to the top modulo-two adder are 010_2, so the output is a one. The inputs to the bottom modulo-two adder are 00_2, so the output is zero. So the encoder outputs 10_2 for the channel symbols.

And so on. The timing diagram shown below illustrates the process:



After all of the inputs have been presented to the encoder, the output sequence will be:

$00\ 11\ 10\ 00\ 01\ 10\ 01\ 11\ 11\ 10\ 00\ 10\ 11\ 00\ 11_2$.

Notice that I have paired the encoder outputs-the first bit in each pair is the output of the upper modulo-two adder; the second bit in each pair is the output of the lower modulo-two adder.

You can see from the structure of the rate 1/2 K = 3 convolutional encoder and from the example given above that each input bit has an effect on three successive pairs of output symbols. That is an extremely important point and that is what gives the convolutional code its error-correcting power. The reason why will become evident when we get into the Viterbi decoder algorithm.

Now if we are only going to send the 15 data bits given above, in order for the last bit to affect three pairs of output symbols, we need to output two more pairs of symbols. This is accomplished in our example encoder by clocking the convolutional encoder flip-flops two (= m) more times, while holding the input at zero. This is called "flushing" the encoder, and results in two more pairs of output symbols. The final binary output of the encoder is thus 00 11 10 00 01 10 01 11 11 10 00 10 11 00 11 10 11_2. If we don't perform the flushing operation, the last m bits of the message have less error-correction capability than the first through (m - 1)th bits had. This is a pretty important thing to remember if you're going to use this FEC technique in a burst-mode environment. So's the step of clearing the shift register at the beginning of each burst. The encoder must start in a known state and end in a known state for the decoder to be able to reconstruct the input data sequence properly.

Now, let's look at the encoder from another perspective. You can think of the encoder as a simple state machine. The example encoder has two bits of memory, so there are four possible states. Let's give the left-hand flip-flop a binary weight of 2^1, and the right-hand flip-flop a binary weight of 2^0. Initially, the encoder is in the all-zeroes state. If the first input bit is a zero, the encoder stays in the all zeroes state at the next clock edge. But if the input bit is a one, the encoder transitions to the 10_2 state at the next clock edge. Then, if the next input bit is zero, the encoder transitions to the 01_2 state, otherwise, it transitions to the 11_2 state. The following table gives the next state given the current state and the input, with the states given in binary:

	Next State, if	
Current State	**Input = 0:**	**Input = 1:**
00	00	10
01	00	10
10	01	11
11	01	11

The above table is often called a state transition table. We'll refer to it as the `next state` table. Now let us look at a table that lists the channel output symbols, given the current state and the input data, which we'll refer to as the `output` table:

	Output Symbols, if	
Current State	**Input = 0:**	**Input = 1:**
00	00	11
01	11	00
10	10	01
11	01	10

You should now see that with these two tables, you can completely describe the behavior of the example rate 1/2, K = 3 convolutional encoder. Note that both of these tables have $2^{(K - 1)}$ rows, and 2^k columns, where K is the constraint length and k is the number of bits input to the encoder for each cycle. These two tables will come in handy when we start discussing the Viterbi decoder algorithm.

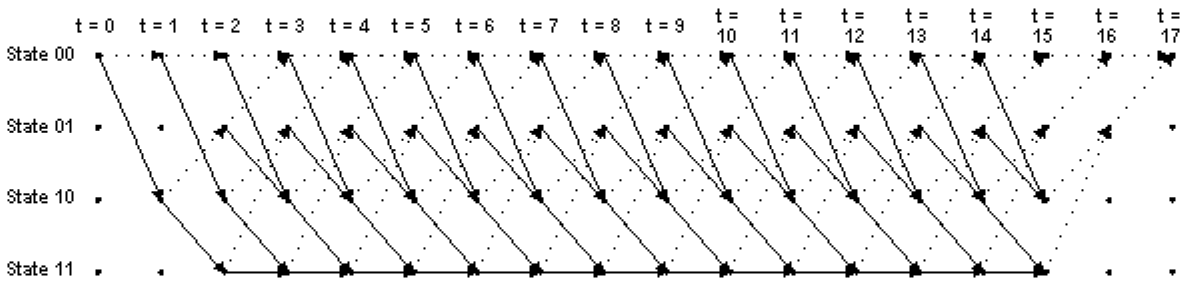
Mapping the Channel Symbols to Signal Levels

Mapping the one/zero output of the convolutional encoder onto an antipodal baseband signaling scheme is simply a matter of translating zeroes to +1s and ones to -1s. This can be accomplished by performing the operation $y = 1 - 2x$ on each convolutional encoder output symbol

Description of the Algorithms (Part 2)

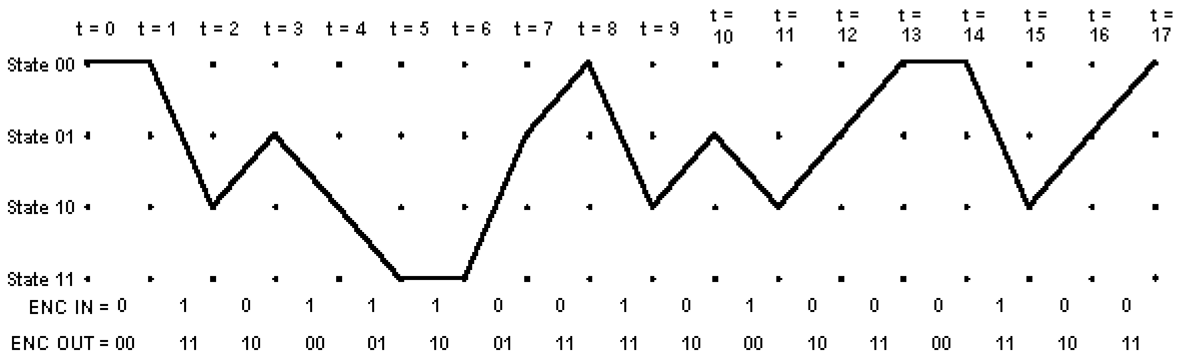
Performing Viterbi Decoding

The Viterbi decoder itself is the primary focus of this tutorial. Perhaps the single most important concept to aid in understanding the Viterbi algorithm is the trellis diagram. The figure below shows the trellis diagram for our example rate 1/2 K = 3 convolutional encoder, for a 15-bit message:

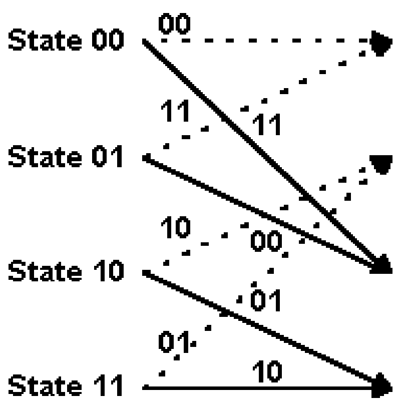


The four possible states of the encoder are depicted as four rows of horizontal dots. There is one column of four dots for the initial state of the encoder and one for each time instant during the message. For a 15-bit message with two encoder memory flushing bits, there are 17 time instants in addition to t = 0, which represents the initial condition of the encoder. The solid lines connecting dots in the diagram represent state transitions when the input bit is a one. The dotted lines represent state transitions when the input bit is a zero. Notice the correspondence between the arrows in the trellis diagram and the state transition table discussed above. Also notice that since the initial condition of the encoder is State 00_2, and the two memory flushing bits are zeroes, the arrows start out at State 00_2 and end up at the same state.

The following diagram shows the states of the trellis that are actually reached during the encoding of our example 15-bit message:

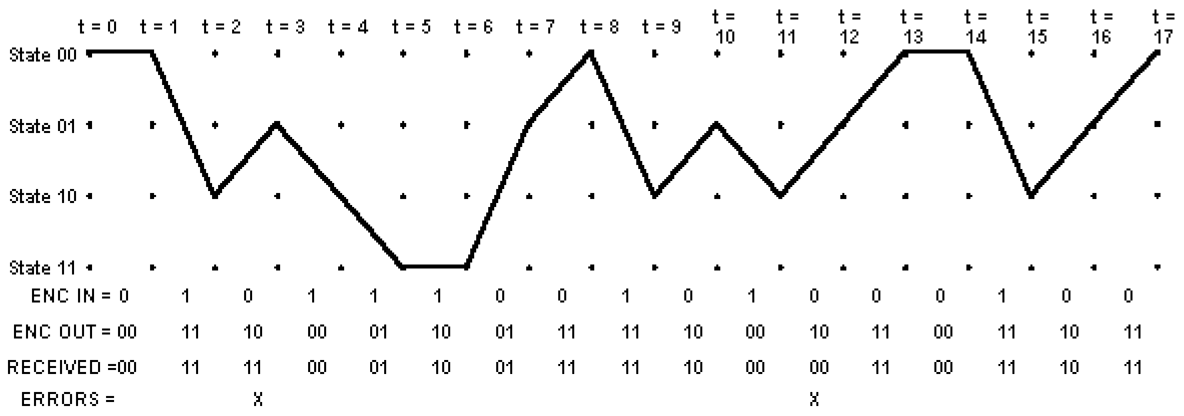


The encoder input bits and output symbols are shown at the bottom of the diagram. Notice the correspondence between the encoder output symbols and the output table discussed above. Let's look at that in more detail, using the expanded version of the transition between one time instant to the next shown below:



The two-bit numbers labeling the lines are the corresponding convolutional encoder channel symbol outputs. Remember that dotted lines represent cases where the encoder input is a zero, and solid lines represent cases where the encoder input is a one. (In the figure above, the two-bit binary numbers labeling dotted lines are on the left, and the two-bit binary numbers labeling solid lines are on the right.)

OK, now let's start looking at how the Viterbi decoding algorithm actually works. For our example, we're going to use hard-decision symbol inputs to keep things simple. (The example source code uses soft-decision inputs to achieve better performance.) Suppose we receive the above encoded message with a couple of bit errors:

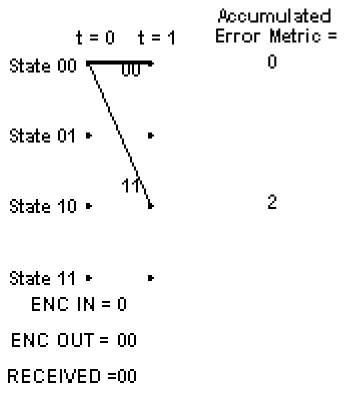


ENC IN = 0	1	0	1	1	1	0	0	1	0	1	0	0	0	1	0	0
ENC OUT = 00	11	10	00	01	10	01	11	11	10	00	10	11	00	11	10	11
RECEIVED =00	11	11	00	01	10	01	11	11	10	00	00	11	00	11	10	11
ERRORS =		X									X					

Each time we receive a pair of channel symbols, we're going to compute a metric to measure the "distance" between what we received and all of the possible channel symbol pairs we could have received. Going from t = 0 to t = 1, there are only two possible channel symbol pairs we could have received: 00_2, and 11_2. That's because we know the convolutional encoder was initialized to the all-zeroes state, and given one input bit = one or zero, there are only two states we could transition to and two possible outputs of the encoder. These possible outputs of the encoder are 00_2 and 11_2.

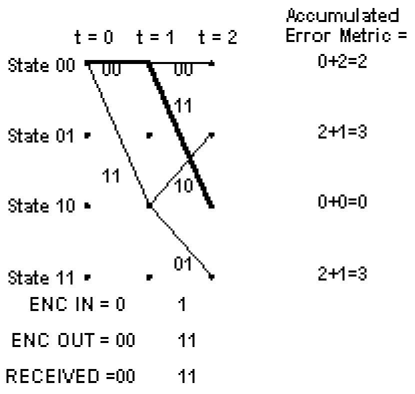
The metric we're going to use for now is the Hamming distance between the received channel symbol pair and the possible channel symbol pairs. The Hamming distance is computed by simply counting how many bits are different between the received channel symbol pair and the possible channel symbol pairs. The results can only be zero, one, or two. The Hamming distance (or other metric) values we compute at each time instant for the paths between the states at the previous time instant and the states at the current time instant are called branch metrics. For the first time instant, we're going to save these results as "accumulated error metric" values, associated with states. For the second time instant on, the accumulated error metrics will be computed by adding the previous accumulated error metrics to the current branch metrics.

At t = 1, we received 00_2. The only possible channel symbol pairs we could have received are 00_2 and 11_2. The Hamming distance between 00_2 and 00_2 is zero. The Hamming distance between 00_2 and 11_2 is two. Therefore, the branch metric value for the branch from State 00_2 to State 00_2 is zero, and for the branch from State 00_2 to State 10_2 it's two. Since the previous accumulated error metric values are equal to zero, the accumulated metric values for State 00_2 and for State 10_2 are equal to the branch metric values. The accumulated error metric values for the other two states are undefined. The figure below illustrates the results at t = 1:



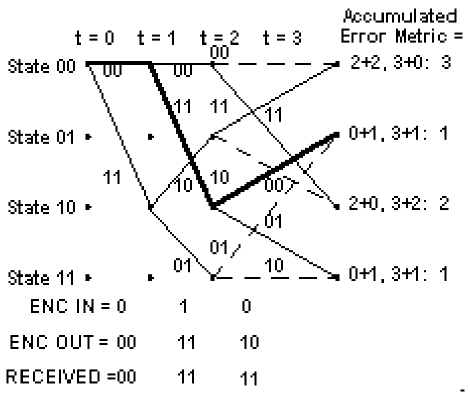
Note that the solid lines between states at t = 1 and the state at t = 0 illustrate the predecessor-successor relationship between the states at t = 1 and the state at t = 0 respectively. This information is shown graphically in the figure, but is stored numerically in the actual implementation. To be more specific, or maybe clear is a better word, at each time instant t, we will store the number of the predecessor state that led to each of the current states at t.

Now let's look what happens at t = 2. We received a 11_2 channel symbol pair. The possible channel symbol pairs we could have received in going from t = 1 to t = 2 are 00_2 going from State 00_2 to State 00_2, 11_2 going from State 00_2 to State 10_2, 10_2 going from State 10_2 to State 01_2, and 01_2 going from State 10_2 to State 11_2. The Hamming distance between 00_2 and 11_2 is two, between 11_2 and 11_2 is zero, and between 10_2 or 01_2 and 11_2 is one. We add these branch metric values to the previous accumulated error metric values associated with each state that we came from to get to the current states. At t = 1, we could only be at State 00_2 or State 10_2. The accumulated error metric values associated with those states were 0 and 2 respectively. The figure below shows the calculation of the accumulated error metric associated with each state, at t = 2.



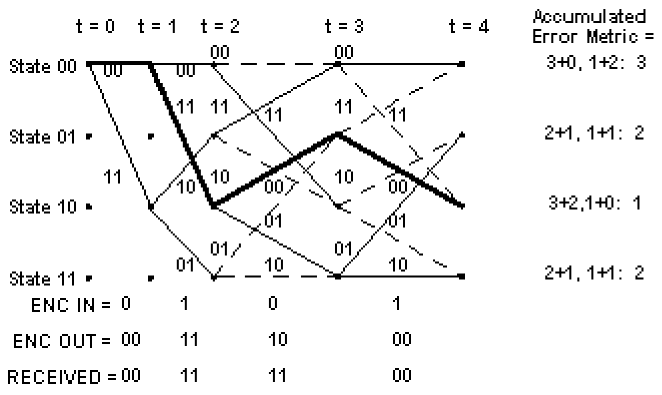
That's all the computation for t = 2. What we carry forward to t = 3 will be the accumulated error metrics for each state, and the predecessor states for each of the four states at t = 2, corresponding to the state relationships shown by the solid lines in the illustration of the trellis.

Now look at the figure for t = 3. Things get a bit more complicated here, since there are now two different ways that we could get from each of the four states that were valid at t = 2 to the four states that are valid at t = 3. So how do we handle that? The answer is, we compare the accumulated error metrics associated with each branch, and discard the larger one of each pair of branches leading into a given state. If the members of a pair of accumulated error metrics going into a particular state are equal, we just save that value. The other thing that's affected is the predecessor-successor history we're keeping. For each state, the predecessor that survives is the one with the lower branch metric. If the two accumulated error metrics are equal, some people use a fair coin toss to choose the surviving predecessor state. Others simply pick one of them consistently, i.e. the upper branch or the lower branch. It probably doesn't matter which method you use. The operation of adding the previous accumulated error metrics to the new branch metrics, comparing the results, and selecting the smaller (smallest) accumulated error metric to be retained for the next time instant is called the add-compare-select operation. The figure below shows the results of processing t = 3:

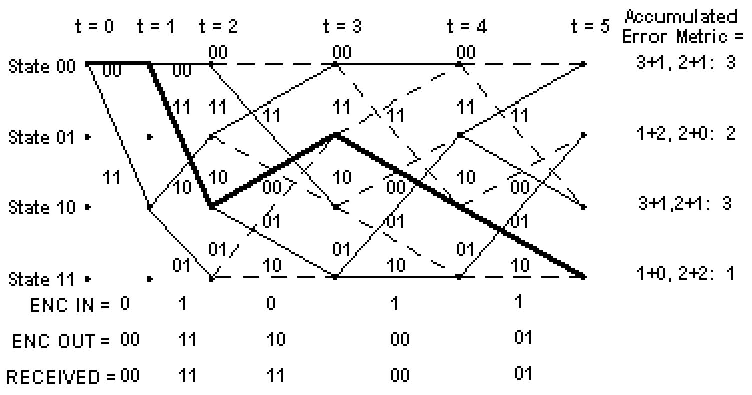


Note that the third channel symbol pair we received had a one-symbol error. The smallest accumulated error metric is a one, and there are two of these.

Let's see what happens now at t = 4. The processing is the same as it was for t = 3. The results are shown in the figure:

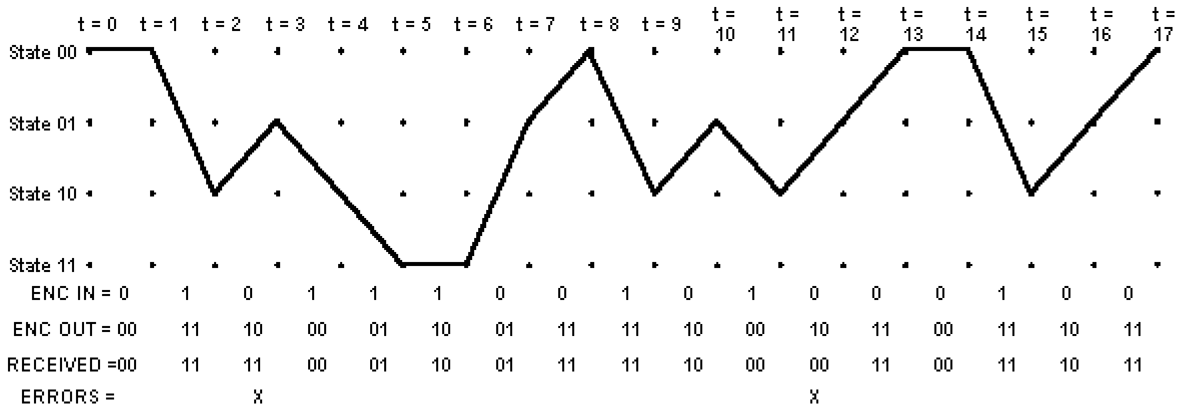


Notice that at t = 4, the path through the trellis of the actual transmitted message, shown in bold, is again associated with the smallest accumulated error metric. Let's look at t = 5:



At t = 5, the path through the trellis corresponding to the actual message, shown in bold, is still associated with the smallest accumulated error metric. This is the thing that the Viterbi decoder exploits to recover the original message.

At t = 17, the trellis looks like this (clutter of the intermediate state history removed):



t = 0
t = 1
t = 2
t = 3
t = 4
t = 5
t = 6
t = 7
t = 8
t = 9
t = 10
t = 11
t = 12
t = 13
t = 14
t = 15
t = 16
t = 17
State 00
State 01
State 10
State 11
ENC IN = 0 1 0 1 1 1 0 0 1 0 1 0 0 0 1 0 0
ENC OUT = 00 11 10 00 01 10 01 11 11 10 00 10 11 00 11 10 11
RECEIVED =00 11 11 00 01 10 01 11 11 10 00 00 11 00 11 10 11
ERRORS = X X

The decoding process begins with building the accumulated error metric for some number of received channel symbol pairs, and the history of what states preceded the states at each time instant t with the smallest accumulated error metric. Once this information is built up, the Viterbi decoder is ready to recreate the sequence of bits that were input to the convolutional encoder when the message was encoded for transmission. This is accomplished by the following steps:

- First, select the initial state as the one having the smallest accumulated error metric and save the state number of that state.
- Iteratively perform the following step until the beginning of the trellis is reached: Working backward through the state history table, for the currently selected state, select a new state by looking in the state history table for the predecessor to the current state. Save this state number as the new currently selected state and continue. This step is called traceback.
- Now work forward through the list of selected states saved in the previous steps. Look up what input bit corresponds to a transition from each predecessor state to its successor state. That is the bit that must have been encoded by the convolutional encoder.

The following table shows the accumulated metric for the full 15-bit (plus two flushing bits) example message at each time t. We will use this table only to select the initial, first state for decoding:

t =	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
State 00_2		0	2	3	3	3	3	4	1	3	4	3	3	2	2	4	5	2
State 01_2			3	1	2	2	3	1	4	4	1	4	2	3	4	4	2	
State 10_2		2	0	2	1	3	3	4	3	1	4	1	4	3	3	2		
State 11_2			3	1	2	1	1	3	4	4	3	4	2	3	4	4		

It is interesting to note that for this hard-decision-input Viterbi decoder example, the smallest accumulated error metric in the final state indicates how many channel symbol errors occurred.

From this table, we can see that the initial, first state to select is state ‘0’.

The following state history table shows the surviving predecessor states for each state at each time t. We will use this table to perform the actual step-by-step traceback:

t =	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
State 00_2	0	0	0	1	0	1	1	0	1	0	0	1	0	1	0	0	0	1
State 01_2	0	0	2	2	3	3	2	3	3	2	2	3	2	3	2	2	2	0
State 10_2	0	0	0	0	1	1	1	0	1	0	0	1	1	0	1	0	0	0
State 11_2	0	0	2	2	3	2	3	2	3	2	2	3	2	3	2	2	0	0

The following table shows the states selected when tracing the path back through the survivor state table shown above:

t =	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	0	0	2	1	2	3	3	1	0	2	1	2	1	0	0	2	1	0

Using a table that maps state transitions to the inputs that caused them, we can now recreate the original message. Here is what this table looks like for our example rate 1/2 K = 3 convolutional code:

	Input was, Given Next State =			
Current State	**00_2 = 0**	**01_2 = 1**	**10_2 = 2**	**11_2 = 3**
00_2 = 0	0	x	1	x
01_2 = 1	0	x	1	x
10_2 = 2	x	0	x	1
11_2 = 3	x	0	x	1

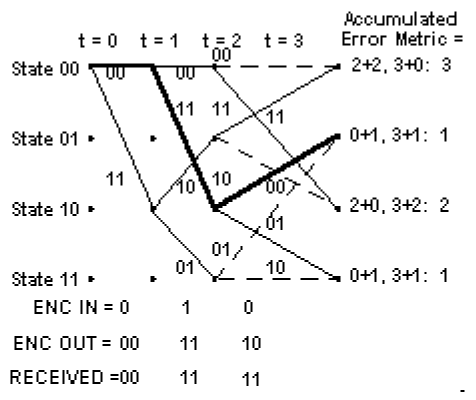
Note: In the above table, x denotes an impossible transition from one state to another state.

So now we have all the tools required to recreate the original message from the message we received:

t =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	1	0	1	1	1	0	0	1	0	1	0	0	0	1

The two flushing bits are discarded.

Here's an insight into how the traceback algorithm eventually finds its way onto the right path even if it started out choosing the wrong initial state. This could happen if more than one state had the smallest accumulated error metric, for example. I'll use the figure for the trellis at t = 3 again to illustrate this point:



See how at t = 3, both States 01_2 and 11_2 had an accumulated error metric of 1. The correct path goes to State 01_2 -notice that the bold line showing the actual message path goes into this state. But suppose we choose State 11_2 to start our traceback. The predecessor state for State 11_2 , which is State 10_2 , is the same as the predecessor state for State 01_2! This is because at t = 2, State 10_2 had the smallest accumulated error metric. So after a false start, we are almost immediately back on the correct path.

For the example 15-bit message, we built the trellis up for the entire message before starting traceback. For longer messages, or continuous data, this is neither practical or desirable, due to memory constraints and decoder delay. Research has shown that a traceback depth of K x 5 is sufficient for Viterbi decoding with the type of codes we have been discussing. Any deeper traceback increases decoding delay and decoder memory requirements, while not significantly improving the performance of the decoder. The exception is punctured codes, which I'll describe later. They require deeper traceback to reach their final performance limits.

To implement a Viterbi decoder in software, the first step is to build some data structures around which the decoder algorithm will be implemented. These data structures are best implemented as arrays. The primary six arrays that we need for the Viterbi decoder are as follows:

- A copy of the convolutional encoder `next state` table, the state transition table of the encoder. The dimensions of this table (rows x columns) are $2^{(K - 1)}$ x 2^k. This array needs to be initialized before starting the decoding process.
- A copy of the convolutional encoder `output` table. The dimensions of this table are $2^{(K - 1)}$ x 2^k. This array needs to be initialized before starting the decoding process.
- An array (table) showing for each convolutional encoder current state and next state, what input value (0 or 1) would produce the next state, given the current state. We'll call this array the `input` table. Its dimensions are $2^{(K - 1)}$ x $2^{(K - 1)}$. This array needs to be initialized before starting the decoding process.
- An array to store state predecessor history for each encoder state for up to K x 5 + 1 received channel symbol pairs. We'll call this table the `state history` table. The dimensions of this array are $2^{(K - 1)}$ x (K x 5 + 1). This array does not need to be initialized before starting the decoding process.
- An array to store the accumulated error metrics for each state computed using the add-compare-select operation. This array will be called the `accumulated error metric` array. The dimensions of this array are $2^{(K - 1)}$ x 2. This array does not need to be initialized before starting the decoding process.
- An array to store a list of states determined during traceback (term to be explained below). It is called the `state sequence` array. The dimensions of this array are (K x 5) + 1. This array does not need to be initialized before starting the decoding process.