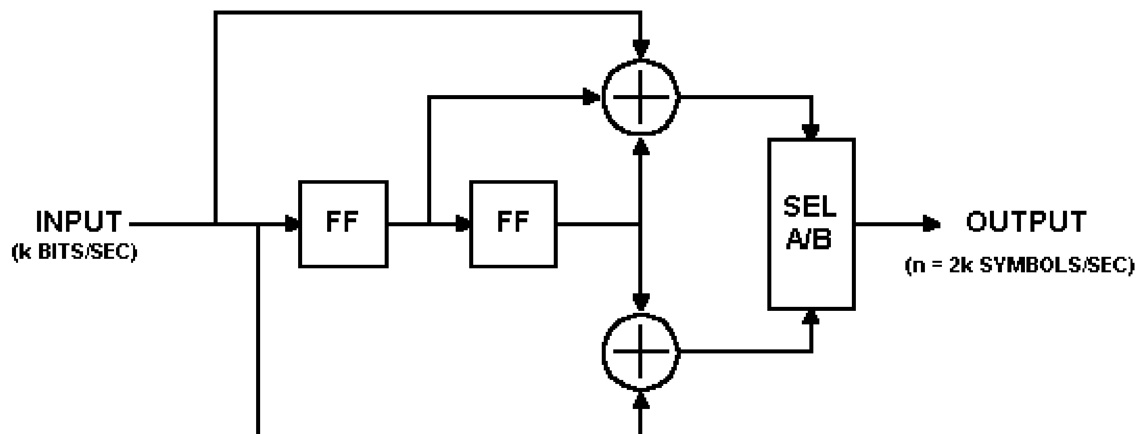
Convolutional coding and one decoding algorithm, the Viterbi Algorithm, are currently used in about one billion cellphones, which is probably the largest number in any application. However, the largest current consumer of Viterbi algorithm processor cycles is probably digital video broadcasting. A recent estimate at Qualcomm is that approximately 1015 bits per second are now being decoded by the Viterbi algorithm in digital TV sets around the world, every second of every day. Nowadays, they are also used in Bluetooth implementations.

The idea of channel coding is to improve the capacity of a channel by adding some carefully designed redundant information to the data being transmitted through the channel. Convolutional coding and block coding are the two major forms of channel coding. Convolutional codes operate on serial data, one or a few bits at a time. Block codes operate on relatively large (typically, up to a couple of hundred bytes) message blocks. For example, Reed Solomon code is a block coder.

The major difference of block coding and the convolutional coding is that block coding is memoryless. Given a string of k bits, a block coder outputs a unique n-bit data block. Convolutional codes do not map individual blocks of bits into blocks of codewords. Instead they accept a continuous stream of bits and map them into an output stream introducing redundancies in the process. The efficiency or data rate of a convolutional code is measured by the ratio of the number of bits in the input, k, and the number of bits in the output, n. In a convolutional code, there is some `memory' that remembers the stream of bits that flow by. This information is used to encode the following bits. The number of the preceding bits used in the encoding process is called the constraint length m (that is similar to the memory in the system). Typically the values of k, n, m are 1-2, 2-3, and 4-7 in commonly employed convolutional codes.

In the next we will talk about a convolutional code with a rate is k/n=1/2.



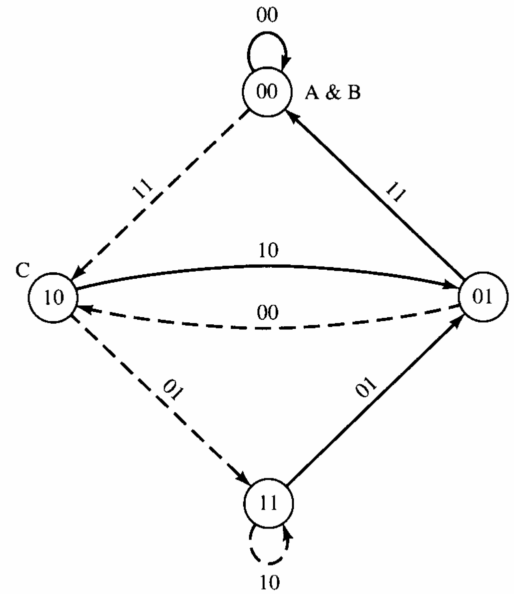
FF: shift register, at each clock tick the content in the shift register is shifted by 1 to the right. The new input comes in as the first bit and the last bit will be the output. A shift register can be considered as adding some delay in the input. The shift registers can be understood as the “memory” of the encoder. It remembers the bits earlier in the sequence

and performs operations with the following bits to output the final result. The shift registers are initialized as all 0's.

Recall that $\oplus$ is an XOR operator. $1\oplus1=0$, $1\oplus0=1$, $0\oplus0=0$.

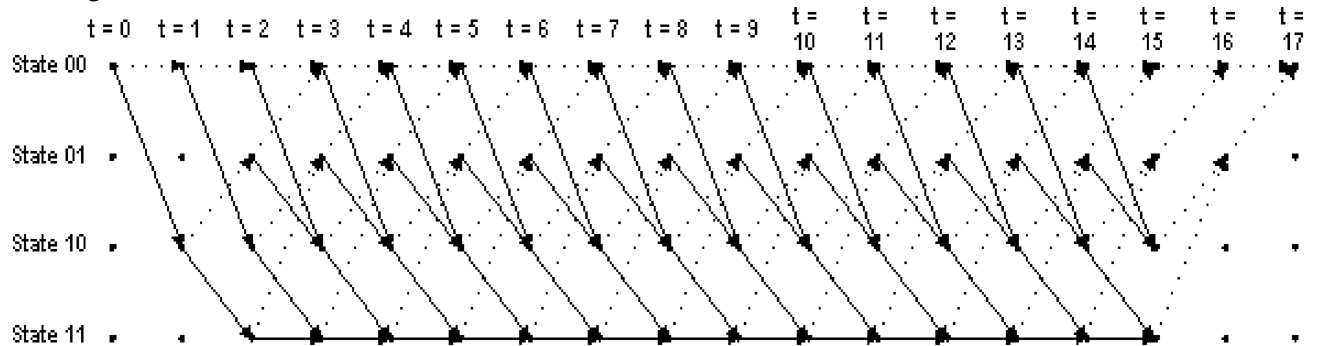
If we work on an input sequence 010111001010001, the output is 00 11 10 00 01 10 01 11 11 10 00 10 11 00 11_2.

This encoder can also be modeled by a finite state machine. Each state is labeled by two bits --- the states of the two shift registers. Each transition is labeled w/v1v2, where w represents the input bit and v1 and v2 represent the two output bits. In this case, we always have w = v1.



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

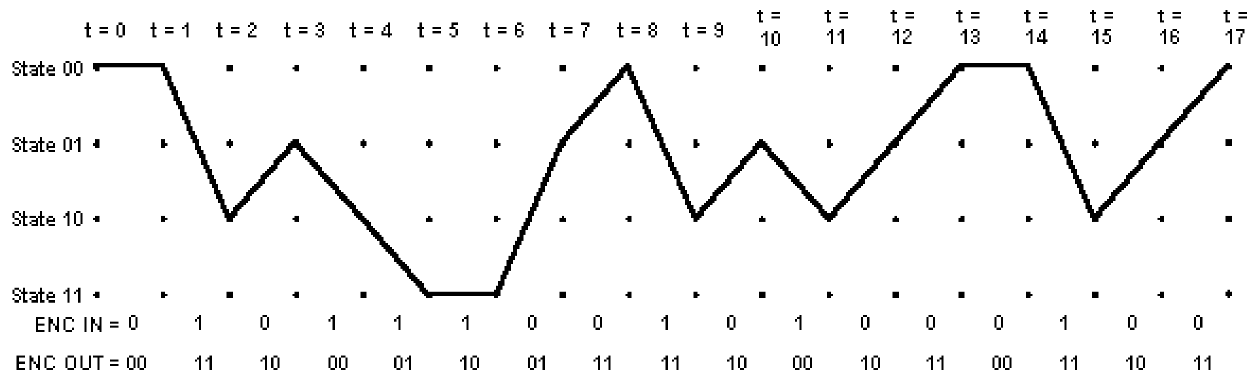
Now we are able to describe the decoding algorithm, mainly the Viterbi algorithm. Perhaps the single most important concept to aid in understanding the Viterbi algorithm is the trellis diagram. The figure below shows the trellis diagram for our example rate 1/2 K = 3 convolutional encoder, for a 15-bit 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. 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

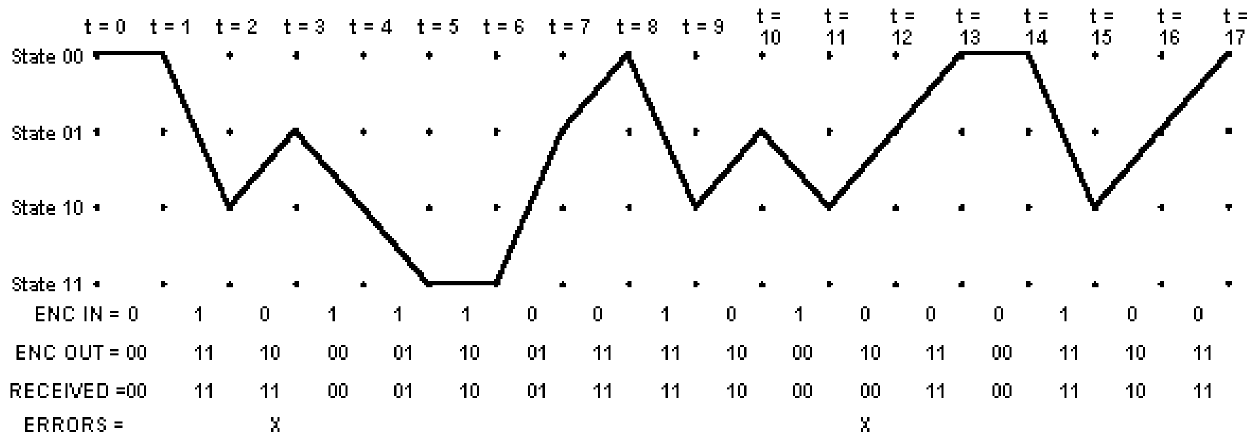
the input bit is a zero. Notice the correspondence between the arrows in the trellis diagram and finite state machine discussed above.

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.



Now let's start looking at how the Viterbi decoding algorithm actually works. Now observe that we have the encoded message (possibly with some error), and we want to recover the original data. In other words, we want to infer from the output through what sequence of transitions we get to the final state.

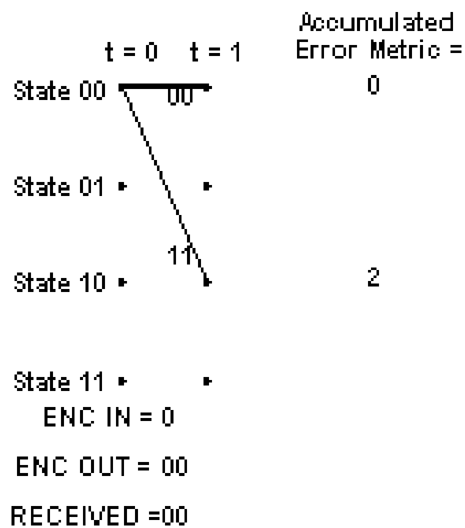
Suppose we receive the above encoded message with a couple of bit errors.



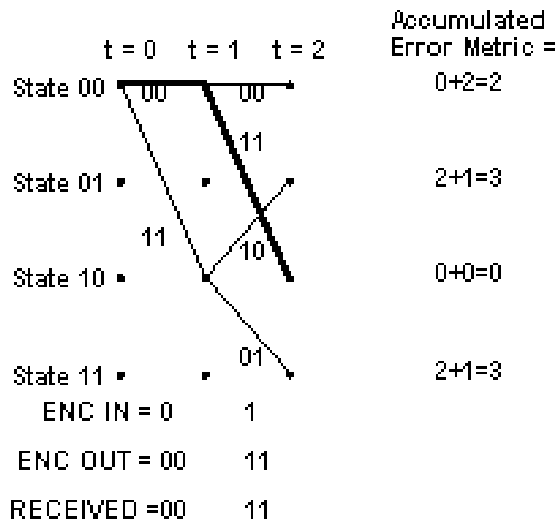
Each time we receive a pair of channel symbols, we're going to compute a metric to measure the "distance" between what we received and all of the possible channel symbol pairs we could have received. Going from t = 0 to t = 1, there are only two possible channel symbol pairs we could have received: 00_2, 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:

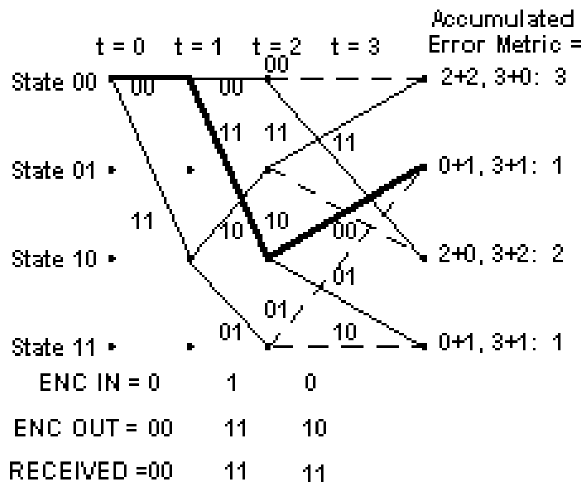


Now let's look what happens at t = 2. We received a 11_2 channel symbol pair. The possible channel symbol pairs we could have received in going from t = 1 to t = 2 are 00_2 going from State 00_2 to State 00_2, 11_2 going from State 00_2 to State 10_2, 10_2 going from State 10_2 to State 01_2, and 01_2 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.



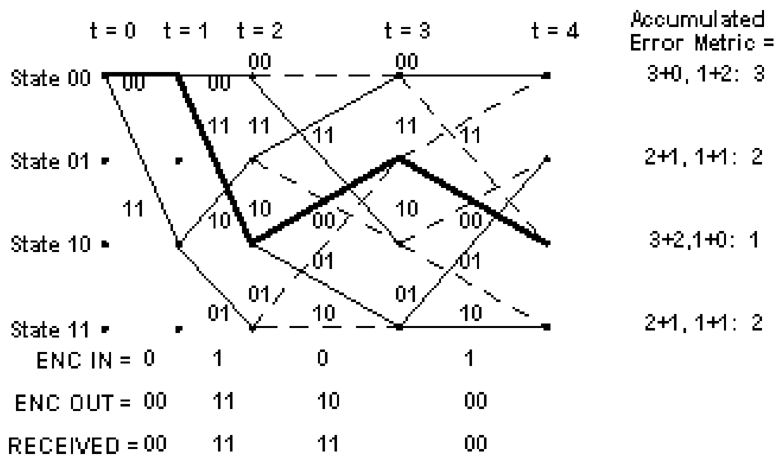
That's all the computation for t = 2. What we carry forward to t = 3 will be the accumulated error metrics for each state, and the predecessor states for each of the four states at t = 2, corresponding to the state relationships shown by the solid lines in the illustration of the trellis.

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:

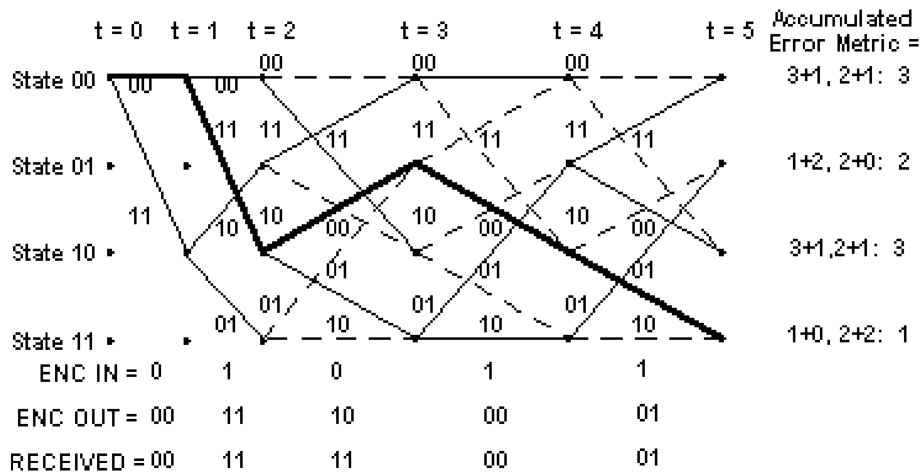


Note that the third channel symbol pair we received had a one-symbol error. The smallest accumulated error metric is a one, and there are two of these.

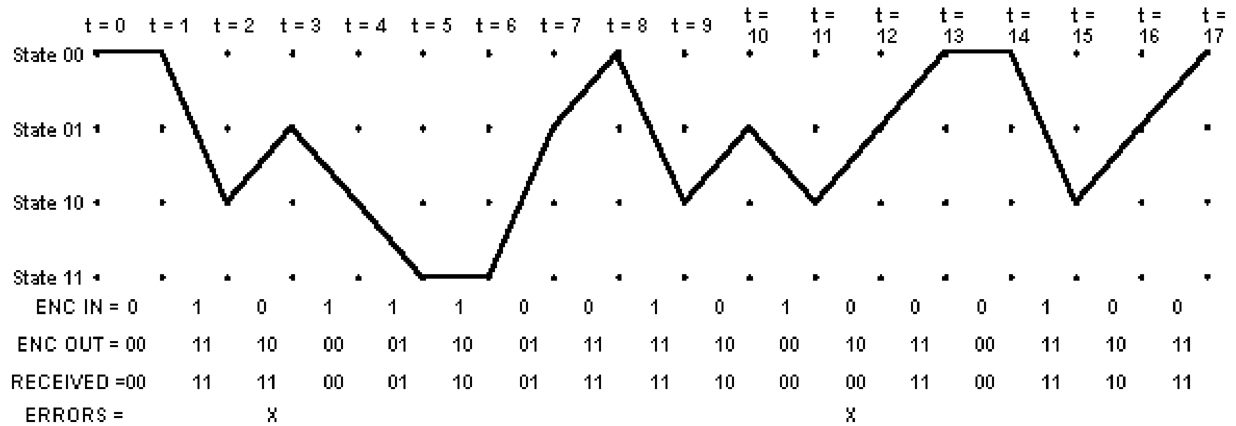
Let's see what happens now at t = 4. The processing is the same as it was for t = 3. The results are shown in the figure:



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 = 17, the trellis looks like this, with the clutter of the intermediate state history removed:



The observation here is that we have correctly decoded the original sequence. If we look back, the way to find the correct data is by taking the sequence of transitions such that the final output is close to the message received. In other words, we perform maximum likelihood decoding.

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 state 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 selected state, select a

new state which is listed in the state history table as being the predecessor to that state. Save the state number of each selected state. 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.

This is exactly dynamic programming.