6.02 - Viterbi Decoding
Lecture #7
Katrina LaCurts, lacurts@mit.edu

## Recap

On Monday, we started convolutional codes. These are codes that operate over a stream of message bits, and transmit only parity bits (not parity and also data). We discussed the encoding process, but left decoding for today.

Recall the three different views for convolutional codes
1. Shift-register view, which reflects how the encoding process is implemented in hardware.
2. State-machine view, which took the possible states that the shift-register could be in, and visualized how we move from one state to another, and what parity bits are outputted.
3. Trellis view, which expanded the state machine out over time.

Today, we will concentrate entirely on the trellis view, because it is most useful in the decoding process.

## Decoding

> Aside: Every time I use the word "decoding" in the first part of this lecture, I am really talking about "hard-decision decoding". You will understand this difference when I get to the second part of the lecture and explain to you what the contrasting "soft-decision decoding" is.

In the trellis view, the encoding process creates a path through the trellis corresponding to the codeword, and we send the corresponding parity bits.

At the other end, the receiver receives a sequence of parity bits; its job is to figure out what the corresponding codeword is, i.e., what path the transmitter took through the trellis.

Really, we are finding the *most likely* path that the transmitter took through the trellis, given that some of the parity bits may be in error.

On Monday we noted that finding the most likely path is equivalent to determining the codeword with minimal Hamming distance between itself and the received sequence of bits.

So imagine that we received the following sequence of parity bits:
  00 01 01 10 ..

We know we start at the [00] state.  The first two parity bits we received were 00.  So that's easy; we know we took the transition from [00] -> [00], and thus the first message bit is a zero.

The next two received parity bits are 01.  This is problematic!  First, the only transitions out of [00] correspond to parity bits 00 and 11; our received parity bits don't match up here.  Moreover, at this point, it seems like both of those transitions are equally as likely; the Hamming distance between 01 and 00 is the same as the Hamming distance between 01 and 11.

---

## Naive Decoding

---

Here is one way we could do the decoding:
- Enumerate all paths that start at the 00 state.
- Find the one that corresponds to a parity bit sequence that's closest in Hamming distance to the received parity bits

With L message bits, there are 2^L possible paths.  And we're dealing with a fairly large L here (on the order of hundreds/thousands); 2^L is not feasible for us.

---

## Viterbi Decoding: Intuition

---

Let's go back to our example.  We couldn't figure out what the second bit of the message should be, which left us with two states to consider: either [00] or [10].

We also know that the next two parity bits are (again) 01.  So imagine we're in state [00]: parity bits 01 don't correspond to either transition out of that state.

Now imagine we're in state [10].  Ah-ha!  01 does correspond to a transition out of that state.  So with this new information, it seems more likely that our path through the states was [00] -> [00] -> [10] (and then -> [11]) rather than [00] -> [00] -> [00] (and then either -> [00] or -> [10]).

This gives you some intuition for the Viterbi algorithm.  At various points, we may not be sure exactly what transition was used.  But as we move through the message, we can use additional parity bits to make

a pretty good guess as to what previous transitions -- that we were unsure about -- should have been.

# Viterbi Decoding

## Background

The algorithm we will use to decode this is called the Viterbi algorithm. It actually solves a larger problem than just convolutional codes. It is a "recursive optimal solution to the problem of estimating the state sequence of a discrete-time finite-state Markov process observed in memoryless noise." (Forney, Proc. IEEE, 1973)

In the context of convolutional codes, it's used in both CDMA and GSM cellular, dial-up modems, statellite, deep-space communications, and 802.11 wireless LANs.

More generally, it is used in speech recognition, speech synthesis, keyword spotting, computational linguistics, and bioinformatics.

> Aside: In a more general context, we would talk about the Viterbi algorithm as doing the following: given a set of observations about a system, it finds the most likely sequence of "hidden states" that led to these observations. For us, those observations are parity bits, and the hidden states are the message bits (which are hidden from the receiver). The algorithm becomes more general by assigning different probabilities to the transitions.
>
> This is not something we'll cover in 6.02, so for now it's fine to only understand Viterbi in the context of convolutional codes. But it's good to know that it's actually much more powerful.

The Viterbi algorithm is a dynamic programming algorithm, which means it's going to solve a smaller sub-problem at each step, and build these solutions up into our final solution.

> Aside: dynamic programming is also a much more general concept.

We will compute the most likely message sequence leading up to every intermediate state, as well as the associated cost. At the end, we'll figure out the allowed final state that has the lowest cost, and "trace back" from there. (We say "allowed" because we may have additional constraints, e.g., the final state has to be 00)

## Introduction

As discussed in class on Monday, every path through the trellis from the possible initial states to the possible final states corresponds to a valid codeword. Thus, every path through the trellis also corresponds to a transmitted message.

For a received codeword r, I can calculate the Hamming distance between r and any valid codeword. Thus, we have a notion of a distance for a particular path through the trellis: the distance of any path p, which corresponds to valid codeword c, is the Hamming distance between c and r. I'll call this distance the error metric for a path.

Now, suppose I show you two possible paths through the trellis. Path 1, corresponding to valid codeword c1, has error metric 5; path 2, corresponding to valid codeword c2, has error metric 3. Which codeword is more likely to be correct? c2. Thus, to figure out the transmitted message, we can just take a look at the message bits that correspond to the transitions of path 2.

The Viterbi algorithm will work in part by exploring many possible paths through the trellis. When we get to the end of the trellis, we will determine which has the minimum error metric, and that path will correspond to the most-likely codeword; from the path, we will be able to determine the (most-likely) message.

Part of what makes this algorithm efficient is that instead of exploring all possible paths through the trellis, it is able to eliminate certain paths as it moves through the trellis. In fact, it only keeps track of the optimal path to each state at each time as it moves through the trellis, thereby eliminating from consideration all but 2^(K-1) candidate paths at each time.

## Example: The beginning

We're going to deal with the following received message:

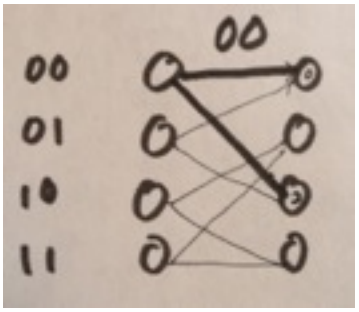
   00 01 01 10 11 00

And the same parity equations from Monday.

We know that the transmitter started at state [00]. Our first two received parity bits are 00.

If the transmitter had taken transition [00] -> [00], the Hamming distance would be 0. If the transmitter had taken transition [00] ->

[10], the Hamming distance would be 2.  So we are starting off on two possible paths.

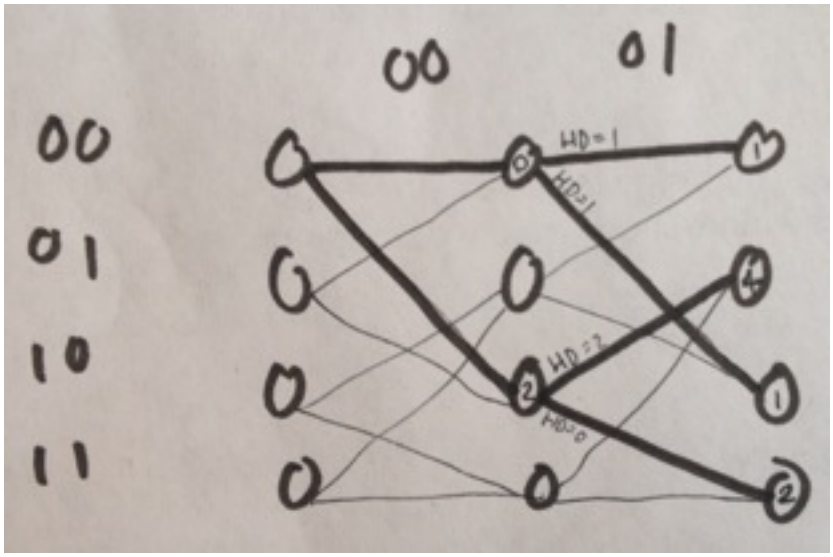



Notice that I am putting an error metric in the states.  This indicates the error metric for the path that ends at that particular state in this iteration.  You should immediately ask yourself: what happens if multiple paths end at the same state?  This is a question I will answer shortly.

Now let's continue.  My next two parity bits are 01.  From what I did in the last step, it seems that I could potentially be in one of two states right now: [00] or [10].

From [00], I can make one of two transitions:
  [00] -> [00]; parity bits 00 => Hamming distance = 1
  [00] -> [10]; parity bits 11 => Hamming distance = 1
From [10], I can make one of two transitions:
  [10] -> [01]; parity bits 10 => Hamming distance = 2
  [10] -> [11]; parity bits 01 => Hamming distance = 0

I want to figure out the error metric for this iteration, so I'm going to add the new Hamming distances to the error metrics that already exist.



## Branch Metric

Before we get to far into this example, I want to start attaching some more formal terms.

At each iteration, we have been computing the Hamming distance between the received parity bits at that iteration and the parity bits associated with different transitions (sometimes we call the transitions "arcs"). Formally, we are computing the "branch metric": the distance between the received parity bits and the expected parity bits for each transition.

The branch metric is not restricted to being the Hamming distance, and we will see an alternative choice later in lecture.

In general, the branch metric is proportional to the negative log likelihood, i.e., the negative log probability that we receive r given that x was sent. As you have seen, the larger the branch metric, the less probable it was that we made that transition.

## Example: Continuing on

Okay. So we have processed the received parity bits 00 01, and we have error metrics calculated through four different paths through the trellis. At this point, it is unclear what makes the Viterbi algorithm so efficient, as we are essentially considering all possible paths.

Our next two received parity bits are 01. We start by calculating all of the appropriate branch metrics:

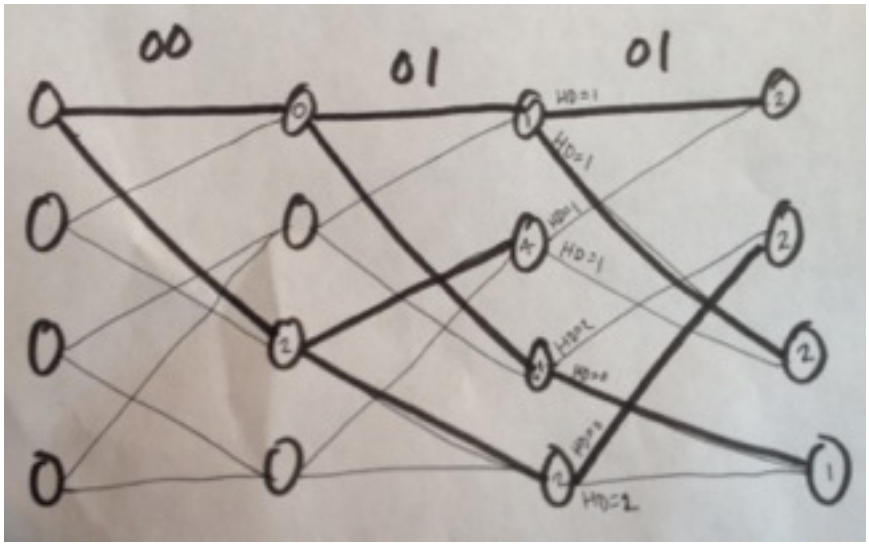
```
 [00] -> [00]: BM = 1          [10] -> [01]: BM = 2
 [00] -> [10]: BM = 1          [10] -> [11]: BM = 0
 [01] -> [00]: BM = 1          [11] -> [01]: BM = 0
 [01] -> [10]: BM = 1          [11] -> [11]: BM = 2
```

Now, if we took the branch from [00] -> [00], that would make our total error metric 2 (1 + 1 = 2). But wait! There are actually two paths that end at state [00]; the one from [00] -> [00], and the one from [01] -> [00].

If we took the other path, that would give us an error metric of

4+1=5.  So at this stage, which path is more likely?  The first.

We'll do the same at the other states (I'll abbreviate "error metric" as "EM")
  At [01]:  Taking [10] -> [01] yields EM = 3 (1+2); taking [11] -> [01] yields EM = 2 (2 + 0).  So we'll use [11] -> [01]
  At [10]:  Taking [00] -> [10] yields EM =2; taking [01] -> [10] yields EM=5
  At [11]: Taking [10] -> [11] yields EM=1; taking [11] -> [11] yields EM=4




Now you can see where the efficiency of the Viterbi algorithm comes in.  There are, in theory, eight potential paths through the trellis now (we're on the 3rd iteration), but we just eliminated some of them.

Why did we do that?  Well, suppose that a wizard appears, and let's you know that at iteration 3, you are supposed to be at state [00]. Originally, there were going to be two paths from the starting state to [00]: [00] -> [00] -> [00] -> [00] (EM=2), and [00] -> [10] -> [01] -> [00] (EM=5).  It is more likely that we took the first path; it has the smaller error metric.  And any paths that continue on from state [00] at iteration 3 will just increase that error metric.  So we don't even consider the second path anymore.

---
Path Metric
---

The final thing to formalize before we continue this discussion regards the path metric, which is related to the error metric I've been discussing, but is subtly different.

We keep track of the path metric for each state at each iteration, i.e., we have

> PM[s,i] for each state s of the 2^{K-1} transmitter states, and for each iteration i (0 <= i < L-1)

What the path metric is, is this:

> PM[s,i] = the smallest sum of branch metrics, minimized over all message sequences that place the transmitter in state s at time i.

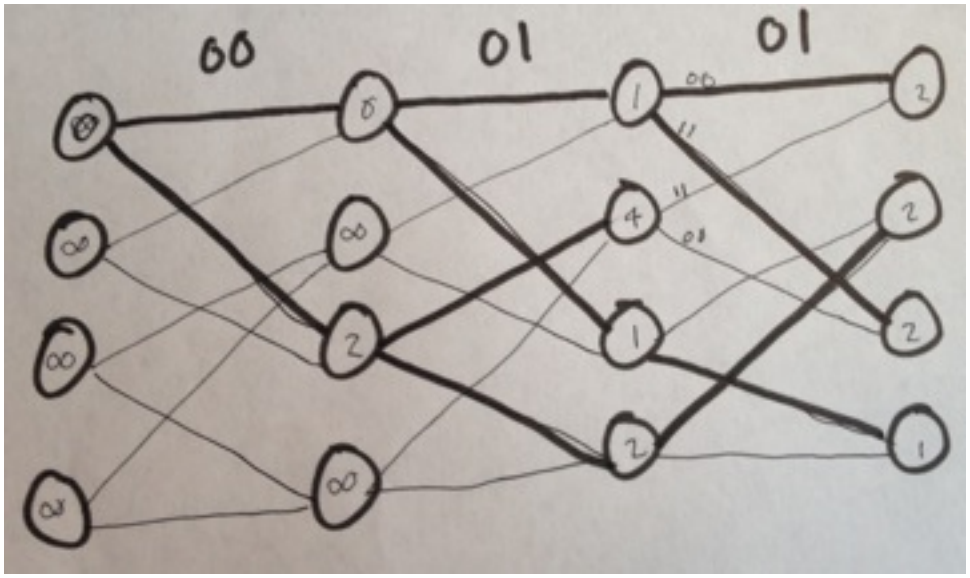
This definition is equivalent to the minimum error distance over all message sequences that place the transmitter in state s at time i.

> Aside: In a more perfect world, this quantity would be called the "optimal path metric", not the "path metric".

Up until this iteration, we only had one possible path to each state in each iteration; the error distance corresponded to the path metric, even though we hadn't introduced a notion of the best path.

We compute PM[s,i+1] using PM[s,i] and the BM for the incoming branches.

You can see at the beginning, we had some states without error distances filled in.  Formally, those are infinite; there was no path that placed the transmitter in those states at those times.



## The Algorithm, Formally

At time i+1, the following is known: PM[s,i] for all states s.

For each state s, at time i+1:

- Determine a and b, the two states that the transmitter could've been in at time i to get to s. These are the "predecessor states" of s.

  Ex: for state [00], the predecessors are a=[00] and b=[01]

  Any message sequences that leaves the trellis in state s at time i+1 must have left the trellis in state a or b at time i.

- Calculate the branch metrics for these two transitions; this gives us BM_a, BM_b

  Ex: if the received parity bits are 01 and we're considering s=[00], then BM_a = 1 and BM_b = 1.

- Determine PM[s,i] = min(PM[a,i] + BM_a, PM[b,i] + BM_b

  Ex: (from the diagram) PM[[00],3] = min(PM[[00],2] + 1, PM[[01],2] + 1) = min(1 + 1, 4 + 1) = min(2, 5) = 2.

  (If there were a tie, we'd break it arbitrarily)

- Keep track of the predecessor state (equivalently, the arc) used to get to s in iteration i.

  Ex: this was [00] -> [00] for our example.

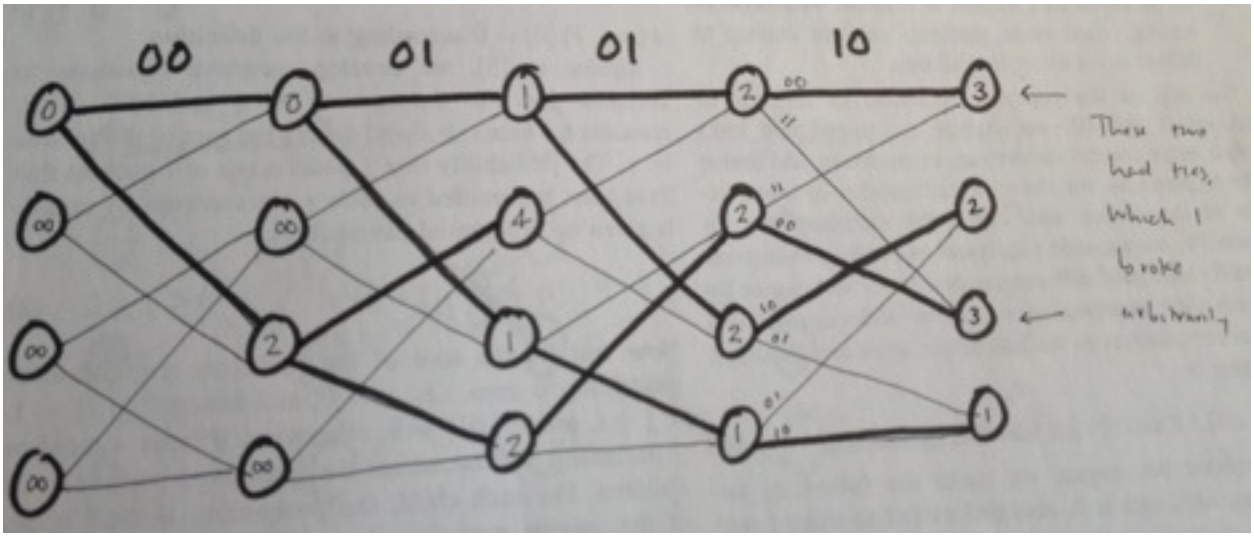
  Aside: In implementation, you could do this with a variety of data structures. In class, I'm just keeping track on the board by coloring over the appropriate arc.

---

Example: The end

---

(Image shows the next iteration)



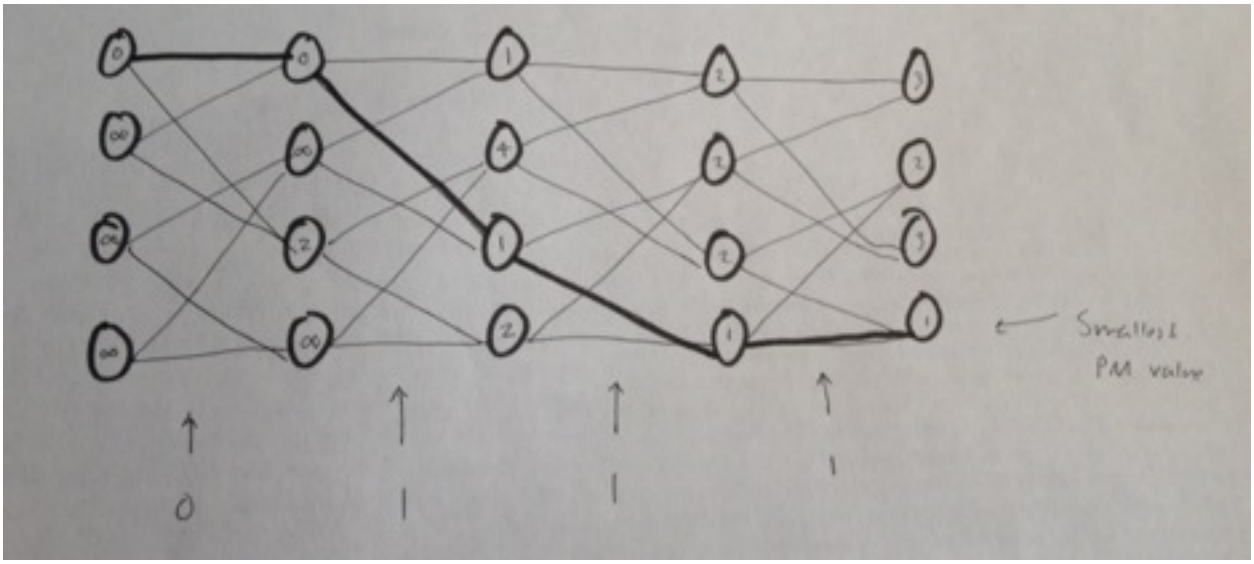

Having completed four iterations, let's look at the states. Which one was the transmitter most likely in at this point? State [11]: it has the smallest path metric.

What does the path metric represent? It represents the smallest number of bit errors obtainable on any path to that state at that time. So the path metric of 1 for state [11] at this end time indicates that there is a path with a 1 bit error between the codeword corresponding to that path and the received codeword. The other states indicate that the codewords corresponding to their respective path are a Hamming distance of either 2 or 3 away from the received codeword.

So the most likely "ending state" is [11]. How can we recover the message? We just need to trace back through the trellis, and figure out what received message bit corresponds to each transition:

```
  [11] -> [11] -> [10] -> [00] -> [00]
       ^       ^       ^       ^
       1       1       1       0
```

So it looks like our received message was 0111.




In this case we were able to correct the bit error.  Without calculating the parity bits for that message, can you tell where the bit error came about in our received message?  It happened between step 1 and 2.  This is where the path metric for our chosen path increased from 0 to 1. (Note: this process worked because in this case we were able to decode the message correctly.)

We can check this: If the message had been transmitted correctly, that message would've resulted in the parity bits 00 11 01 10.  Our parity bits (00 01 01 10) do indeed reflect a 1-bit error in this message, on the second set of parity bits (which corresponds to "between step 1 and 2" in our algorithm).

==========
Viterbi Algorithm: Complexity
==========

What is the complexity of this algorithm?  At any given time there are 2^{K-1} paths that we're tracking, i.e., 2^{K-1} most-likely message that we're tracking.

An alternative way of thinking: at each step i, we're choosing between the 2 ways to get into each of the 2^{K-1} states.

Thus, we get L*2^{K-1} complexity, i.e., the time complexity of the algorithm grows exponentially with constraint length K, but only linearly with message length L (as opposed to exponentially in L, for simple enumeration).  This is fine; K is often small, whereas L is often large.

==========
Soft-decision Decoding
==========

Recall our binary-symmetric channel:

[transmitter] ----- BSC -----> [receiver]

We have actually been eliding one layer in this diagram:

[encode data][send as voltages] --->
[receive voltages][convert voltages to bits][decode data]

To transmit the bits, the transmitter takes the encoded data and sends it across the channel as voltages. For now, you can just assume that the transmitter sends 0 volts to represent a 0 bit, and 1 volt to represent a 1 bit (another popular choice is -1 for 0, +1 for 1). How we get these voltages across the channel is a topic for the second part of 6.02.

Let's talk about the "convert voltages to bits" part of the receiver's process. Without noise, the task on the receiver's end would be easy: for each voltage received, if it's 0 volts, interpret that as a 0 bit; if it's 1 volt, interpret that as a 1 bit.

But the voltages don't arrive as a sequence of 0 and 1 volts. Because the channel introduces noise, the received voltages can be corrupted. For instance, you might receive 0.02 instead of 0, or 0.96 instead of 1. The receiver thus needs to "digitize" the voltage samples into bits.

> Aside: I am still eliding a host of details here. For instance, the receiver actually receives multiple voltage samples per bit, not just one. But this is a challenge for another time.

For now, we'll use the following rule: if the received voltage is >= .5, interpret as a 1; else, a 0.

You should be wondering why I'm bringing this up at all. Consider the following sequence of voltages: .9999 .9997 .9999 .5001 .9998

By my rule, all of these will be interpreted as 1's. But that 4th voltage sample is awfully close to being interpreted as a 0 bit. In some sense, it's not a very "good" 1 bit.

When we immediately digitize voltage samples, we lose information about how "good" the bit is, and throwing away information is almost never a good idea when making decisions. If the Viterbi algorithm had access to these voltages, perhaps it could redefine its branch metric to indicate, e.g., that the distance between .999V .999V and 11 is very small, but the distance between .999V and .501V and 11 is greater, despite the fact that both of those voltage pairs would get digitized to 11.

Not surprisingly, this wasn't a hypothetical exercise. Everything I have taught you up to this point is really "hard-decision decoding":

we digitize bits first, and then decode.  We can also do "soft-decision decoding", where the Viterbi algorithm is sent the received voltages instead of the digitized bits.

To redefine our branch metric, we often use the square of the Euclidean distance between received voltages and expected voltages.

  Ex: for .999,.999 and 11: (.999-1)^2 + (.999-1)^2 = .000002
      for .999,.501 and 11: (.999-1)^2 + (.501-1)^2 = .249002

  Aside: there are other metrics we could use.  The reason we choose this one is because it works best for the type of noise we see on most channels.  You'll learn more about this in the coming weeks.

The rest of the Viterbi algorithm is exactly the same.

## Free Distance (again)

From Monday, recall the free distance: d_free = the smallest-weight among nonzero codewords.

In our example, d_free = 5, corresponding to path [00] -> [10] -> [01] ->
[00].  This means we expect to be able to correct floor(5-1/2) = 2 bit errors as long as they are "far enough apart".

Looking at the diagram, you can get a better sense of how far "far enough apart" is for this code.  It takes our decoder six output bits to "settle" back to the [00] state.  If we were receiving the all zero codeword, we'd be back in the correct state at that point, and ready to correct more errors.

## Puncturing

The rates of convolutional codes are 1/r, where r is the number of parity bits per message bit.  The code we've been working with has rate 1/2.

What happens if we want to have a rate higher than 1/2?  Or a rate between 1/r and 1/(r+1)?

To achieve these types of rates, we can use a technique called puncturing.  In puncturing, the transmitter only sends a subset of the parity bits, not all of them.  This subset is agreed upon ahead of time by the transmitter and receiver.

For instance, take our example with two parity equations. These equations result in two streams of bits, which we interleave when we transmit:

    p0[0]p1[0] p0[1]p1[1] p0[2]p1[2] p0[3]p1[3] p0[4]p1[4] ...

Let's puncture the first stream by skipping every third bit in the stream:

    p0[0]p1[0] p0[1]p1[1] - p1[2] p0[3]p1[3] p0[4]p1[4] - p1[5]

(I've placed a "-" where no parity bit from p0 is sent). This process could be described by the puncturing vector 110.

We can puncture the second stream at the same time. Let's use the vector 101. We get:

    p0[0]p1[0] p0[1] - - p1[2] p0[3]p1[3] p0[4] - - p1[5]

   (As a stream, that's just p0[0]p1[0] p0[1]p1[2] p0[3]p1[3]
   p0[4]p1[5])

Three questions to answer:
1. What is the rate now?
2. How does the decoder deal with a punctured stream?
3. How are the error correction properties affected?

Before, our rate was 1/2, or equivalently, six parity bits for every 3 message bits (2 parity bits for every bit). Now we send four parity bits for every 3 message bits; our rate is 3/4.

On the decoder's end, since it knows which parity bits are missing, it just skips them; they don't factor into the calculation of branch metrics. Other than that, the decoding procedure is exactly the same.

Regarding the error correction properties: intuitively, we will likely be able to correct fewer errors with the punctured code.

One very nice thing about puncturing is that it allows us to use a single convolutional encoder/decoder regardless of the rate we want. So if we have already built a convolutional encoder/decoder, but we want a rate 3/4 code, we don't have to build a different device; we can use the same implementation and puncture the stream.

==========
The end of coding
==========

This lecture ends the first part of 6.02! To recap, we discussed:

1. Encoding data efficiently via Hamming codes and LZW compression. Hamming codes assume we know the symbol probabilities beforehand, and that they are independent; LZW makes no such assumptions.

2. Adding error correcting abilities to our codes via linear block codes (e.g., Hamming codes) and convolutional codes.

3. Decoding these two types of codes efficiently, via syndrome decoding and Viterbi decoding, respectively.

When we choose how to encode data, we care about multiple things:
- How many bits we're using (we'd like to use fewer)
- How the rate of the code compares to the channel capacity (are we getting as much information through the channel as possible?)
- How efficient the encoding and decoding processes are
- The constraints of the system itself (remember with the space probe examples, those transmitters could only handle certain block sizes)

What you will learn about in the second part of this course is how we actually get those bits across a channel, and how we deal with the types of errors that can manifest there.