[Approximate Challenge] Convolutional Coding

Convolutional Codes

"The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point."

— Claude Shannon

Imagine designing a protocol to communicate with a robot 12 billion miles away from you. No matter what the size of your communication array, you must accept that the message will likely arrive garbled, confused, noisy, and weak. You could try to send the same message over-and-over again, hoping that at least one of your messages will make it through unscathed; but, how many times is enough? How will this affect the cost of sending messages, or the responsiveness to your messages?

Back on earth, imagine communicating with a computer with only a tiny antenna which moves erratically throughout cities across the globe. No matter how you design your broadcast antennas, your messages will arrive garbled, confused, noisy, and weak... but cell signals still have to work somehow (most of the time, anyway).

These are both problems of *robust coding*. This challenge is to implement encoding and decoding algorithms for a particularly nice kind of robust coding: a *convolutional code*. These codes were a major component of the communications algorithm for Voyagers 1 and 2, Cassini, Mars Explorer, and Pathfinder.

Convolutional codes are a form of stream coding invented by Peter Elias in 1955. Convolutional codes consist of an encoder and a decoder and are advantageous because the encoder is incredibly simple and the decoder is parallelizable.

Convolutional codes are classified by two numbers, (*N*, *K*). *N* is the *output bit count* and *K* is the *constraint length*. More informally, N counts the number of output transmit bits encoded for each input source bit, meaning that a convolutional code transmits source data at a rate of 1/N source bits per transmitted bits; K describes the amount of internal memory the encoder has, dramatically impacting the complexity of decoding.

For example, Voyager 1 and 2 used a (2, 7) convolutional code, while Mars Exploration Rover and Pathfinder used a (6, 15) code.

To fully classify an (*N*, *K*) convolutional code, we also need *N* bit vectors *g0* through *g(N-1)*, each of length *K*. These are the *coding polynomials*.

Briefly, if you're familiar with DSP, a convolutional code encoder works by producing one output stream for each coding polynomial by convolving the polynomial with the input stream.

Briefly, if you're familiar with inference on Markov models, a convolutional code decoder works by using the Viterbi algorithm to infer the most likely latent state of the encoder given an observed output stream and then reads the input stream off of it.

Ultimately, the goal of this challenge is to decode a message sent from Voyager and recode it for transmission to Pathfinder. More specifically, you'll be provided with the coding polynomials for an incoming message that you must decode, and then a new set of coding polynomials to use to re-encode the message to deliver elsewhere.

It may be the case that the messages you receive are corrupted with random bit-flipping noise---but the convolutional code decoder will account for this, ensuring perfect transmission (with a high degree of probability).

What follows describes how to build a convolutional code encoder and decoder and then the format and limits of the challenge.

Convolutional Encoding

Let's first consider a very simple (2, 1) convolutional code with coding polynomials.

```
g0 = [1]
g1 = [1]
```

This will turn out to be the "repeat 2" code. Encoding works like this:

```
(+) ---> t0
 ^
 |
 m0 <--- s
 |
 v
(+) ---> t1
```

Imagine the above shift register circuit as mapping the flow of bits through the encoder. Bits flow in through `s` one-by-one, and are stored in a memory cell `m0`. Then, we flow further to emit the first two transmitted bits `t0` and `t1`, which are both simply copies of the data stored in `m0`. Thus, if you put a source bit `0` in at `s` then both `t0` and `t1` emit `0` and visa versa.

The structure of this shift register is encoded in the coding polynomials. In particular, `g0[0] = 1` indicates that `t0` is a sum including data from memory cell `m0` and likewise for `g1[0] = 1` and `t1`.

All memory cells are initialized with 0 and all computation is performed in the binary ring Z2, e.g. addition is XOR, multiplication is AND.

A more complex example introduces the idea of delays. Consider the following polynomials

```
g0 = [1,0]
g1 = [0,1]
```

which correspond to the following shift register diagram

```
(+) ---> (+) ---> t0
          ^
          |
 m1  <<<  m0 <--- s
 |
 v
(+) ---> (+) ---> t1
```

or, drawn more simply,

```
        (+) ---> t0
         ^
```

```
       |
  m1  <<<  m0 <--- s
  |
  v
 (+) ------------> t1
```

In this diagram, the `<<<` indicates a "delay". When a signal bit is fed in through `s` it updates the value of `m0` and the old value of `m0` "flows" on to update the value of `m1`. Then, the values of `m0` and `m1` are queried to produce the output bits `t0` and `t1`.

This code produces an output message which goes "two steps forward one step back". As an example, the input signal `[1,1,0,1,1]` produces the output signal `[[1,0],[1,1],[0,1],[1,0],[1,1]]` where the first value of each pair is the same as the input and the second value is the value of the memory register `m0` in the previous state.

Finally, we can see how the coding polynomials indicate the structure of the encoder more clearly now that there are multiple memory cells. In general, an (*N*, *K*) convolutional code has *N* output lines and *K* memory cells.

As a final example, consider the (2, 7) convolutional code used in the Voyager messages. It has coding polynomials `g0 = [1,1,1,1,0,0,1]` and `g1 = [1,0,1,1,0,1,1]` which results in the following

```
(+)------------(+)--(+)--(+)--(+) ---> t0
 ^              ^    ^    ^    ^
 |              |    |    |    |
 m6 < m5 < m4 < m3 < m2 < m1 < m0 <--- s
 |    |         |    |         |
 v    v         v    v         v
(+)--(+)-------(+)--(+)-------(+) ---> t1
```

Additional Notes on Convolutional Encoding

The examples above provide the general idea, but these notes are important for understanding conventions and use of encoders.

- Convolutional encoding follows a shift register model.
- You can generate each "frame" of an encoding using a single matrix multiply.
- Typically we write the output bitstream as the concatenation of each output frame, so an output `[[1,0],[1,1],[0,1],[1,0],[1,1]]` becomes the bitstream `1011011011`.
- We assume all of the registers are zeroed to start.
- After the last source bit is sent, we send *K* more zeros to pad the message. This clears all of the remaining information from memory and sends it along the channel.

Examples

- For `g0 = [1]` encoding is the identity.
- For `g0 = [1]`, `g1 = [1]`, `g2 = [1]` encoding is "repeat thrice".
- For the (2,7) Voyager code above, sending the ASCII stream "hi", `0110100001101001` results in the output.

  `0011010111011001111010011101101001100000011100`

  a stream of `(16*N + K*N)` bits including the padding zeros.

Convolutional Decoding

The key to decoding a convolutional code efficiently under potential errors (bit-flips) introduced by the communication channel is to think of the shift register encoding circuit as a *finite state machine* (FSM).

In particular, an (*N*, *K*) code has `2^K` states and emits *N* symbols (bits) on transition.

As an observer of only the output bits the decoder must somehow model its *best guess* as to the internal state of the encoder. This is known as the *maximum likelihood estimate* of the encoder state. Once we have the MLE of the state trajectory of the encoder over the entire transmitted message we can read off `m0` (and chop off the tail padding) to recover our best guess at the input message.

Since an (*N*, *K*) encoder sends *N* bits for every one input bit it's likely that even if some number of those transmitted bits are flipped it won't prevent the MLE from still being accurate to the source message. This is where the *robustness* of the convolutional code comes from.

The next question to tackle is how to create an MLE of the internal state of a FSM given only its emitted output. For this, we'll consider a small but non-trivial example (2,2) code with `g0 = [0,1]` and `g1 = [1,1]`. Since *K*=2 there are 4 internal states of the encoder: `s=00`, `01`, `10`, and `11`. The encoder begins in state `00` and we can model the system's responses to an input bit of `0` or `1`.

If the first input bit is `0` then the new state is `00` (still) and the encoder emits the bits `s*g0 = 0` and `s*g1 = 0` where `(*)` is the vector inner or "dot" product. If the first input were `1` instead then the new state is `10` and the encoder emits `s*g0 = 0` and `s*g1 = 1`. What's important to note is that there are only two possible emissions from this first bit (`00` and `01`), but when we receive the first two bits there is a chance one has been flipped so we may actually observe any of the four possibilities `00`, `01`, `10`, or `11`.

So, to be clear, imagine that the first source bit was `1` and thus the encoder emitted the two bits `01`. Now, let's imagine that we actually observe the bits `11`. If we take the *Hamming Distance*, the number of un-matching bits, between the observed pair of bits and each of the two possiblilities we'll see that the bit emission `00` is at Hamming Distance 2 from what we observed while the bit emission `01` is at Hamming Distance 1. Thus, from this data alone, our MLE of the source signal is `1`, which happens to be accurate. This demonstrates the decoder recovering from a single error.

```
Frame 1 Summary    Verdict
----------------    -------
Source      = 1
Transmitted = 01
Received    = 11
MLE         = 1      MATCH!
```

The other thing that our MLE suggests is that the new state of the encoder after the first signal bit is `10`. We could use this as the assumed internal state for decoding the next bit and repeat the process.

While iterating this process this captures the spirit of an MLE, it is unfortunately insufficient. In particular, if we had been *wrong* about the first bit then we would continue on a false assumption about the internal state. As a result, we'll almost certainly be wrong about every subsequent bit.

Instead, what we'll want to do is consider *all possible paths* through the space of internal states and keep only the most likely (smallest net Hamming Distance) path. This will be the MLE.

Constructing state space on a Trellis

To construct the actual decoder algorithm we need to take the intuition gained from examining the encoder as an FSM and extend it to take account of transitions over time better. For this we will introduce a kind of graph called a Trellis.

A Trellis is a visual representation of the information and state of the convolutional encoder displayed against the actual received message. It's laid out in a grid of `2^K` rows each with `L/N+1` columns where `L` is the length of the received message. Each column will be called an *epoch* and each row represents a particular selection of encoder state. The first epoch is the time before the encoder has received any inputs (and thus is in the all zero state), then between each epoch an input bit is received an `N` output bits are emitted leaving the encoder in a new state.

For the (2,2) code from the previous section, this looks like the following for `L = 14`

```
    State
 Epoch -----  0  1  2  3  4  5  6  7

      11          () () () ()
      01          () () () () ()
      10       () () () () ()
      00 () () () () () () () ()
         ------ -------------- ------
         filling "steady state"  flushing
```

Here the `()` indicates a *possible state* over time. In the region marked "filling" we note that (assuming, as usual, that the encoder begins in the all zero state) after receiving the first signal bit it is not posible for the encoder to reach states like `01` or `11`. On the other side there's the "flushing" zone which represents the trailing zero-padding in the signal used to flush the memory back to the all zero state.

Since this Trellis has 8 epochs it represents a message of `N*(8-1)` transmitted bits or `7` signal bits. Additionally, since `K=2` the last two signal bits must have been `0`s to flush the encoder state so the true signal must have been 5 bits long.

Now that we have the Trellis grid, we should consider transitions. A transition between epoch `t` and epoch `t+1` occurs when the `t+1`th signal bit is consumed by the encoder and `N` outbit bits released leaving the encoder in a (potentially) new internal state. For instance, consider the transition from epoch `0` to epoch `1`. We begin in coordinate (00,0) and if the `1`st signal bit is a `0` then we'll transition to coordinate (00, 1) and emit the output bits `00`. If it were a `1` then we'll transition to coordinate (10, 1) and emit the output bits `01`.

Estimating the MLE with Viterbi's min-sum algorithm

Now that we have a Trellis for our encoder we can use it to compute the MLE with ease. If we consider a path from epoch 0 to epoch `L/N` we can easily compute the message which *ought* to have been transmitted and furthermore we can compute the net Hamming ditance between that supposed message and the one we actually received. We'll determine the MLE by finding the state-space path with output message of the smallest Hamming distance from the message we received.

We know that we must arrive at the final epoch of the Trellis in state `00`, so we ask "What is the minimum *distortion* path to arrive at (00,*L/N*) given the transmission we received?". Viterbi's recursive min-sum algorithm is an elegant way to solve this.

To run this example we'll need a received input. Let's imagine we receive the following (possibly corrupted) 14 bits

```
01101110011100
```

What was the code that generated this transmission?

We'll work from the tail end producing the value `MS(00,7)` which is a pair of values `(path, distance)` where `path` is the minimum-length path through state space taken by encoder and `distance` is the net Hamming distance between what ought to have been transmitted and what we actually received. Since we could only have transitioned from either (01,6) or (00,6) we consider recursively those two emissions, `MS(01,6)` and `MS(00,6)`. Since both transitions `(01,6) -> (00,7)` and `(00,6) -> (00,7)` result in the emission of the output bits `00` and those are indeed what we observed then this transition does not increase the net Hamming distance of the path. Therefore, the minimal distortion path arriving at `MS(00,7)` is the shorter (the *min* part) of the two paths `MS(01,6)` and `MS(00,6)` appending (the *sum* part) the transition to state `00` and the increased distortion (in this case 0).

This algorithm continues recursively until we arrive at coordinate `(00,0)` where `MS(00,0) = ([],0)` by definition.

When this algorithm completes we simply take the `path` component from `MS(00,7)` and read off the `0`th memory cell from the internal state of the encoder: this is the most likely input signal estimate!

If you complete the Trellis from the previous example you will find that the source message was `11001`.

Notes

- If you're paying close attention you may have noticed that it's not necessary to actually transmit output from the final, `K`th flushing bit since it always drives the system to the all-zero state which means the final bits are constant: `00`. A real system would avoid sending these.

Resources

- Notes on Convolutional Coding
- Chapter 48 of David McKay's *Information Theory, Inference, and Learning Algorithms*

Input Format

An input specification contains 3 pieces:

1. The configuration of the receiving decoder
2. The configuration of the transmitting encoder
3. An incoming encoded bitstream

The two configurations are both represented in the same way. First, there is a line determining the N and K parameters of the code, and then there are N lines each with a length-K bitstring which indicate the N coding polynomials.

Thus, a problem which asks you to receive a Voyager-encoded message and then transmit it as a thrice-duped message looks like

```
2 7
1111001
1011011
3 1
1
1
1
```

The incoming message is simply an arbitrary length encoded bitstream. For reading convenience there may be whitespace included and this should be ignored.

Output Format

Many of the test cases include the addition of random transmission errors. It is possible that these errors cannot be corrected by the decoder—too much information has been lost.
Grading for this problem is thus not all-or-nothing but, instead, answers are penalized according to the

hamming distance between the error-free correct result and the actual output. For this reason, even a perfectly performing submission may not receive a perfect score.
Test cases 0-9 are extremely low-error transmissions, however. A functioning decoder should successfully decode them perfectly.

Sample Input

```
2 7
1111001
1011011
3 1
1
1
1
001110000100000010000010110110011110100111011010010101101111011111110011
010000011111010111011010010101101111011111110011010000011111010100001101
001100011011001011111110011111101011010110000001111101101011000101000010
111011110101100001000001111101010011101110111000010100100010110110110010
111111101001000110010001111101111100101111010110000111101000000111110101
000011101000111001110011110000111100001111000011001011110110000000110100
011101110111000101000001100001111111001101000001111101010011010101001010
110000010111100111110001111101101011001011111110100111001101111100011010
010101011001000110010010010001010100100101110000011111110110101001010101
10011100110111111111010101001001010010000000010000100010001011100001110
101101011000000111110101001101010100100101111101001100100011010101001010
111110101000011010110001010000100011010101001010110000010100001011011010
010101011001111101100011010111001101110001000111111100111001101111101010
110000
```

Sample Output

(The following is displayed with apparent whitespace wrapping. This should not be included in the actual output.)

```
000111000111000111000000000111111000111000000000000111111000111000000111000111111111000000111111000000
111000000000000000000111111000111000000111000111111111000000111111000000111000000000000000000111111000
000111000000000111111111000111000111000111111000000111000111000000111000000000000000000111111111000111
000000000111111000111111111111000000111000000000000000000111111000000000111111000111111000111111000000
000111111111000111000111000111111000111111000111000111111111000000111111000111111111111000000111000000
111000000000000000000111111000000111000111000111111000000111111000000111111000000111111000000111111000
111111111111000111111111000000111000000111111111000111000000000111111111000000111111000000111000000000
000000000111111000000000000111000111111111000111000000000000111000000000000000000111111111000111000111
000111111000111111111000000111111000111000000111000111111000111111000111000111111000000000000111000111
111000000111111111000111111000111000000111000111111000111111111000000111111000000000000111000111111000
000000111000000111111000111111000000000111111000000111000111000000111000000000000000000111111000000000
000111000111111000000111000000000111111000000000000111000111111111000000000000000111111111000111000000
000111111000000000000111000111111111000111000000000111111000111000000111000111111000111111111111000111
111000111111111000000111111111000000111111000000111000111111111000000
```

Explanation

The sample input specifies first the Voyager (2,7) code with coding polynomials `g0 = [1,1,1,1,0,0,1]` and `g1 = [1,0,1,1,0,1,1]` which corresponds to the following shift register diagram

```
(+)------------(+)--(+)--(+)--(+) ---> t0
 ^              ^    ^    ^    ^
 |              |    |    |    |
 m6 < m5 < m4 < m3 < m2 < m1 < m0 <--- s
 |    |         |    |         |
 v    v         v    v         v
(+)--(+)-------(+)--(+)-------(+) ---> t1
```

It then specifies the (3,1) thrice-repeating code with coding polynomials `g0 = [1]`, `g1 = [1]`, and `g2 = [1]` which corresponds to the following shift register diagram

```
m0 <--- s
|
*-----> t1
|
*-----> t2
|
*-----> t3
```

Finally, it lists a 942 bit encoded message on 14 lines although the whitespace must be deleted before interpretation as a single bitstream.

The output is that input decoded probabilistically according to the Voyager model and then re-encoded using the repeat-thrice code. It is printed as 1395 bits.

Tested by Stanislav Pak