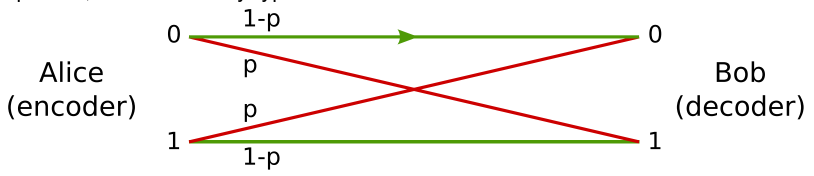
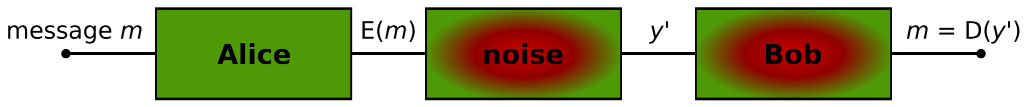
Viterbi

Alice and Bob are working together to bake some treats while they are home for winter break. Alice is in charge of sending Bob the recipe, and Bob is responsible for following directions. Bob is *extremely* literal- if an error in the recipe says to use 2000 eggs instead of 2, he will use 2000 eggs without a second thought. Alice is aware of Bob's lack of common sense, but she is also busy. She already has the recipe open on her phone, so she wants to send it via email (using wifi) to Bob. Unfortunately, Alice's evil next-door-neighboor Eve has her microwave running continuously at maximum power. Microwaves emit a lot of radiation around wifi's 2.4GHz channels, causing interference. This lab will explore different techniques for ensuring that Alice's message will make it to Bob uncorrupted.

Preliminaries

We assume that Alice's message is \(N\) bits long with each bit iid Bernoulli(0.5). We model the channel as a binary symmetric channel (http://en.wikipedia.org/wiki/Binary_symmetric_channel), as shown below. Each bit sent through the channel is flipped (independently) with probability p. We'll assume p=0.05, which is a fairly typical value for wireless communications.





Alt

We assume that Alice and Bob use some sort of message integrity check in their message (like a CRC (http://en.wikipedia.org/wiki/Cyclic_redundancy_check)) - to keep things simple, let's assume Bob can detect any error with probability 1.

We also assume that Bob sends Alice an ACK or NACK and assume that this message always succeeds. If Bob sends an ACK, Alice knows Bob got the message. If Bob sends a NACK, Alice tries to send again.

Repetition Code

We need a way to correct errors introduced by our noisy channel. One of the simplest error correcting codes is the repetition code (http://en.wikipedia.org/wiki/Repetition_code).

Repetition coding works by having Alice send each bit of her message \(r\) times. For each bit, Bob uses majority logic - i.e. for \(r=3\) if he gets two 1s and one 0, he will decide 1. The code below computes the probability that the message is corrupted for various values of \(r\).

In [ ]:
```
%pylab
%matplotlib inline
from scipy.stats import binom
r=[2*i+1 for i in range(15)]
def single_bit_error(r, p=0.05):
    return 1-binom.cdf(r//2, r, p)
def message_error(r, p=0.05):
    return 1-(1-single_bit_error(r,p))**256
message_error(12)
semilogy(r, [message_error(i) for i in r])
```

This is good! We can get to low probabilities of error if we increase the number of repetitions. However, increasing the number of repetitions means we are sending a lot more data, lowering Alice's effective datarate. Alice's phone also has limited battery life, and sending lots of copies of the same data consumes a lot of energy. Is it possible to correct a lot of errors without having to send so many extra bits?

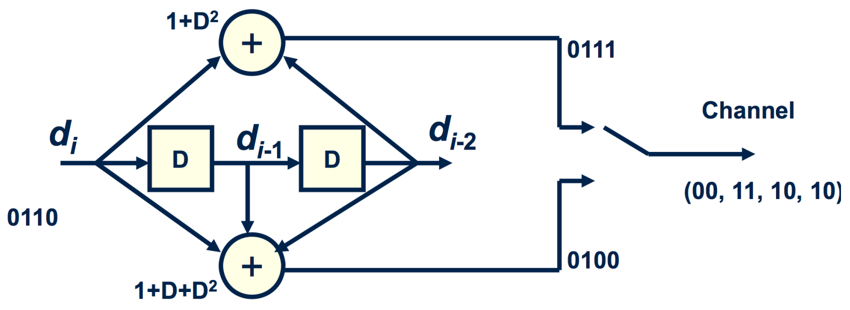
Shannon says that we should be able to do much better. Using random coding with infinite block length, we can achieve capacity, which for the BSC is \(1-H(p) \approx 0.71\) bits per channel use for \(p=0.05\). This is way better than the \(\frac{1}{r}\) channel uses per bit given by repetition coding.

Using random codes sounds great for Alice, but what about Bob? How is he supposed to decode what Alice is sending him? It is not clear how much computation it will take for him to do a good job decoding. He could try syndrome decoding or some other exhaustive search, but this takes exponential time (or space) in the length of the message, which could be really long!

Is there something better?

Convolutional Coding

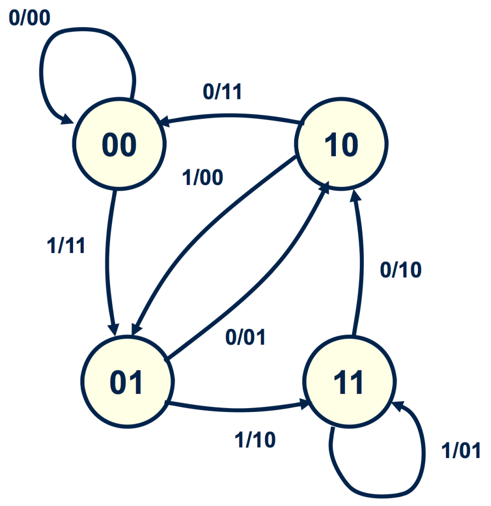
Yes! The 802.11 standards for wifi use convolutional codes and LDPC codes to correct errors. Convolutional codes can be efficiently decoded using the Viterbi algorithm, so they will be the focus of this lab.



The above picture is a block diagram for a simple convolutional encoder. The input message is treated as a stream of bits. The input is shifted through a series of delays - at time t, the input is in[t], the first delay element (the "D" on the left) contains in[t-1], and the last delay elements (the "D" on the right) contains in[t-2]. In this example, each input bit produces two output bits - the first output computed by the top "adder" and the second output computed by the bottom "adder". These additions are modulo-2, so each output is a 0 or 1 bit. In this example, the first (top) output at time t is computed as (in[t]+in[t-2]) mod 2. The second (bottom) output at time t is computed as (in[t]+in[t-1]+in[t-2]) mod 2. The two outputs are interleaved into one bitstream so the output is (top[0], bottom[0], top[1], bottom[1], top[2], bottom[2], ...).

The first thing to note is that this is not actually all the different from repetition coding. Like repetition coding, we are adding redundancy by generating multiple output bits per input bit. However, unlike repetition coding, convolutional codes have memory. Each output bit is a function of multiple input bits. The idea is that if there is an error, you should be able to use the surrounding bit estimates to help you figure out what was actually sent.

The figure below shows the state transition diagram corresponding to the example encoder above. Each transition is labelled input/(top, bottom). Each state represents the contents of the delays, with the most significant bit being the last (right) delay. Be sure to convince yourself that the encoder above is equivalent to the state transition diagram below.



If we assume that the input bits are iid Bernoulli(0.5), this is a Markov chain with every state equally likely. We can run the Viterbi algorithm on our output bits (even after going through a noisy channel) to recover a good estimate of the input bits.

An implementation for a Viterbi decoder can be constructed as follows: 1. A state \(s\) has a *path metric* \(p_s\) that gives the number of observed bit errors associated with being in \(s\) at a given time. 2. A state \(s\) and input bit \(b\) have *branch metric* \(b_{s,b}\) that compares the observed channel output to the expected output given that you were in state \(s\) and had input bit \(b\). The branch metric is the number of different bits between the observed and expected output (Hamming weight). 3. If input \(b\) has transitions from state \(s\) to \(s'\), we can compute an updated path metric as \(p_s + b_{s,b}\). 4. Each state \(s'\) will have two incoming transitions, we select the minimum \(p_s+b_{s,b}\) and call that our new path metric \(p_s'\). This is called Add-Compare-Select. 5. Traceback uses the decisions at each add-compare-select to reconstruct the input bit sequence. Starting at the ending state with the smallest path metric, traceback finds the predecessor based on the decision made by the add-compare-select unit. For example, if at the end, state zero has the lowest path metric, and the last add-compare-select for state zero chose the path_metric coming from state 2, we know that the last input bit was zero and the previous state was 2. This continues backwards until it reaches the beginning.

This link (http://home.netcom.com/~chip.f/viterbi/algrthms2.html) may be a helpful resource for understanding implementing the Viterbi algorithm for convolutional codes (note that what we are designing is called a hard decoder- soft decoders are a refinement that we won't worry about).

We assume that we start and end in state 0. We end in state 0 by appending enough 0s to the end of our input to force us into 0. Our decoder relies on this by initializing all path metrics to a big number, except for 0 which we initialize to 0.

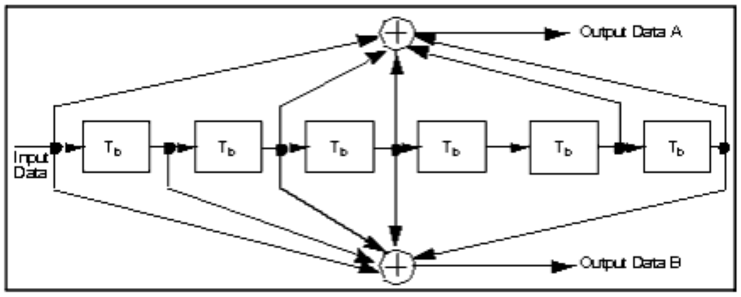
Q1

In the space below, implement a viterbi decoder for our 4-state example code.

A1

In [ ]:
```python
import numpy as np
def apply_generator(state, generator):
    return reduce(lambda x,y: x^y, map(lambda x: x[0]*x[1], zip(state,generator)), 0)

example_generators=[[1,0,1], [1,1,1]]
def encode(bits_in, generators=example_generators):
    l = max(map(len, generators))
    bits_in = list(bits_in) + [0]*(l-1) # end in state 0
    state = [0]*l
    output = []
    for b in bits_in:
        state[1:] = state[:-1]
        state[0] = b
        for g in generators:
            output.append(apply_generator(state, g))
    return output

def bsc(bits_in, p=0.05):
    out = []
    for b in bits_in:
        if np.random.uniform() > p:
            out.append(b)
        else:
            out.append(1-b)
    return out

def int2state(i, w):
    return [i&(1<<k)>0 for k in range(w)]

def viterbi(bits_in, generators=example_generators):
    pass
```

In [ ]:
```
# Test your code here
a=encode( (1,0,1,1,1,1,1,1))
b=bsc(a,p=0.05)
c=viterbi(b)
print a
print b
print c
```



The above picture shows the convolutional encoder actually used by the wifi standards. This encoder still puts out two output bits per input bit, but there are a lot more states. This means it might be able to do a better job correcting errors, but it increases the amount of computation our decoder needs to perform.

Q2

After you have your code working for the 4-state example code, run the following code to make sure it works on the more complicated wifi codes.

A2

In [ ]:
```
# Test your code here
wifi_generators = [ [1,0,1,1,0,1,1], [1,1,1,1,0,0,1] ]
a = encode( (1,0,1,0,1,0,1,0), generators=wifi_generators)
b=[i for i in bsc(a,p=0.05)]
c=viterbi(b, generators=wifi_generators)
print a
print b
print c
```

Let's see how these two codes compare. We are going to plot the bit error rate = \(\frac{\textrm{Number of incorrectly decoded bits}}{\textrm{Total number of bits}}\) for some different channel parameters.

Q3

For \(0.01\le p \le 0.1\) on the x-axis, plot the bit error rate on a log scale on the y-axis. Run 10 trials for 512-bit long inputs at each channel parameter. Plot Both the WiFi and example codes.

A3

In [ ]:

Bit error rate is a good metric for evaluating how well a code performs in noise. In addition to BER, the other two main features used to evaluate decoders for error correction are throughput and energy efficieny. The previous section probably took a long time to run, so our throughput is probably pretty low. Let's measure throughput more accurately in the next section.

Q4

Use the following code block to measure the throughput of your Viterbi decoder.

A4

In [ ]:
```
import timeit

n_trials = 10
received = []
for i in range(n_trials):
    bits_in=np.random.randint(0,1, 512)
    received.append(bsc(encode(bits_in, generators=wifi_generators), p=0.05))

start_time = timeit.default_timer()
for i in range(n_trials):
    decoded = viterbi(received[i], generators=wifi_generators)

elapsed = timeit.default_timer() - start_time

print str((n_trials * 512) /float(elapsed)) + " bits per second"
```

Q5

Approximate the number of arithmetic operations needed to decode one bit using the algorithm above. If you want to watch a youtube video at 10Mbps, we many operations per second does your processor need to do to keep up?

A5 Your Answer Here

The answers to Q4 and Q5 suggest our python implementation is very inefficient (unless your implementation is *way* better than the solutions). Still, the answer to Q5 is a lot of operations per second for a low power wifi chip to run on a serial processor. Fortunately, the computations for each state can be performed in parallel, so specialized hardware can decode with very high throughput without needing multi-gigahertz clocks. This parallelism, in addition to the simplicity of the arithmetic operations (no floating-point needed), makes convolutional codes and Viterbi decoders very hardware friendly.

Q6

Look at this decoder (http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=535288&url=http%3A%2F%2Fieeexplore.ieee.org%2Fiel3%2F3909%2F11339%2F00535288.pdf%3Farnumber%3D535288) designed at Berkeley in the mid-90s. Figure 7 shows decode rate and power. Use the values from figure 7 to find the chip's energy per bit. Look up the typical power for the processor you have been using to write this lab, and use your answer from Q4 to find Energy / bit = Power / Throughput. Compare the throughput and energy per bit for the chip from the 90s vs. your computer. Which is more efficient? Recall that your computer has an advantage of almost two decades of Moore's law making everything faster and more efficient.

A6

In [ ]:
```
#thanks http://stackoverflow.com/questions/10237926/convert-string-to-list-of-bits-and-viceversa
def tobits(s):
    result = []
    for c in s:
        bits = bin(ord(c))[2:]
        bits = '00000000'[len(bits):] + bits
        result.extend([int(b) for b in bits])
    return result

def frombits(bits):
    chars = []
    for b in range(len(bits) / 8):
        byte = bits[b*8:(b+1)*8]
        chars.append(chr(int(''.join([str(bit) for bit in byte]), 2)))
    return ''.join(chars)
```

In [ ]:

```
fin = open('secret_bits.txt', 'r')
secret_bits = [int(i) for i in fin.read()[1:-1].split(',')]
fin.close()
```

Q7

We have included Alice's message for Bob that she encoded with the wifi convolutional codes. It went through a BSC(0.04) channel. Can you recover the message?

A7

In [ ]:

```
print viterbi(secret_bits, generators=wifi_generators)
```

Example encoder and state transition diagram courtesy Bora Nikolic's Spr2013 EE290C Lectures.