18.413: Error-Correcting Codes Lab March 4, 2004

Lecture 9

Lecturer: Daniel A. Spielman

9.1 Related Reading

- Fan, Chapter 2, Sections 3 and 4.
- Programming Tips #10: Working with GF (2) matrices.

9.2 LDPC Codes

We will examine the simplest and most natural low-density parity-check (LDPC) codes. These are specified by a check matrix in which each column has 3 ones and each row has 6. There should be twice as many columns as rows. Thus, the resulting code will have rate around 1/2. (It could be more if the rows turn out not to be a basis)

In particular, we will need to generate such a check matrix at random.

The most natural (to me) way to do this is to view the check matrix as specifying a graph. We will have one *bit node* for each column, and one *check node* for each row. We will form a bipartite graph between the bits nodes and check nodes, with an edge connecting a bit node to a check node if the corresponding entry in the matrix is 1.

Thus, we need to form a random bipartite graph between a set of n bit nodes and a set of $n/2$ check nodes, in which each bit node has degree 3 and each check node has degree 6. To choose a random graph with these parameters, consider the *sockets* on each side, where a socket is a place where an edge is attached. So, each bit node has 3 sockets and each check node has 6. You can form a random graph by matching up the bit node sockets and the check node sockets, using a random permutation.

The problem you will encounter is that you might form two edges between some bit node and some check node. To fix this, you could either just try again (and stop when you get a propper graph), or alter your permutation. I prefer to alter the permutation by performing a random edge exchange: take one of the duplicated edges, choose another edge at random, and swap their endpoints on one side. Repeat this until you've eliminated all the defects from the graph.

9.3 Decoding

Decoding will be by belief propogation on the normal graph. The normal graph looks a lot like the graph described above, except that

- For each bit node, it has an extra edge with just one endpoint
- Each bit node is called an equallity constraint (which corresponds to a repetition code)
- Each check node is called a parity constraint
- There is a variable on each edge.

At the beginning of the algorithm, each of the interal edges (those with two endpoints) is initialized to representing equal probabilities of 0 and 1, and the variables on the external edges (those with degree one attached to the bit nodes) correspond to the received signal from the channel. Thus, they have some *prior* probability of being 0 or 1.

The algorithm will alternate between two types of phases: equality phases and check phases. During an equality phase, each equality constraint receives an *intrinsic* probability for the variable on each of its edges. It then sends out an *extrinsic* probability to the variables on each of its edges. This extrinsic probability is the probability that the corresponding bit is 1, given the other *intrinsic* probabilities. Note that since this is an extrinsic probability, it does not depend up the intrinsic probability for that variable.

The same thing happens during a parity phase, except that the probability is computed using the fact that the associated variables have to have parity 0.

As the phases alternate, the extrinsic probabilities output by one phase become the intrinsic probabilities input to the other.

While the treatment of intrinsic probabilities is identical to that of prior probabilities, we call them intrisic instead of prior because they might not be the actual priors.

By the rule above, during each equality phase, a message is sent to the one-endpoint edge attached to each bit node. This message is almost what the output of the decoder should be. The defect is that this message is an extrinsic probability, and the decoder should output a posterior probability. To fix this, just combine the extrinsic probability for that edge with the prior probability (which came from the channel). After any number of iterations, one can do this to get the guess currently running through the system. As the number of iterations grows, these guesses should converge.

9.4 Why?

I just gave you the algorithm, but I failed to describe why it has the form it does. Here's the reason: if we had a tree instead of a graph, and we wanted to get the estimate for each node, this is the algorthm we would use. In this case, the graph is not a tree. But, we are pretending that it is one.

9.5 SNR, dB

The standard way to report the standard deviation of the Gaussian channel is through the signal-to-noise ratio (SNR). Symbolically, this is written E_b/N_0, and if we are using ± 1 signalling, it is defined to be

$$\frac{E_b}{N_0} = \frac{1}{2R\sigma^2},$$

where R is the rate of the code you are using.

To complicate things further, SNR is usually recorded in decibels. That is $10\log_{10}(E_b/N_0)$ dB. So, if I ask you for $1.5dB$, that means you should set E_b/N_0 so that $10\log_{10}(E_b/N_0) = 1.5$. That is,

$$\frac{E_b}{N_0} = 10^{1.5/10} = 1.4125.$$

Then, you need to reverse this formula again to solve for σ, which in this case is $\sigma = 0.8414$. So, when E_b/N_0 goes up, σ goes down.