

Good Codes and Minimum Distance

Recall that for a code to be guaranteed to correct up to t errors, its minimum distance must be at least $2t + 1$.

What's the minimum distance for the random codes used to prove the noisy coding theorem?

A random n -bit code is very likely to have minimum distance $d \leq n/2$ — if we pick two codewords randomly, about half their bits will differ. So these codes are likely *not guaranteed* to correct patterns of $n/4$ or more errors.

A BSC with error probability Q will produce about nQ errors. So for $Q > 1/4$, we expect to get more errors than the code is guaranteed to correct. Yet we know these codes are good!

Conclusion: A code may be able to correct *almost all* patterns of t errors even if it can't correct *all* such patterns.

What a Good Linear Code Looks Like

Minimum distance isn't the whole story, but nevertheless, it's not good for a linear code to have very low-weight code words (and hence very small minimum distance).

A consequence: The generator matrix for a good code should not be sparse — each row should have many 1s, so that encoding a message with only one 1 produces a codeword that has many 1s.

The decoder's perspective: To be confident of decoding correctly, getting even *one* bit wrong should produce a large change in the codeword, which will be noticeable (unless we're very unlucky).

Low Density Parity Check Codes

We should avoid sparse generator matrices. But can we use a sparse parity-check matrix?

Doing so isn't *quite* optimal, but such "Low Density Parity Check" (LDPC) codes can be very good.

The big advantage of LDPC codes: There is a *computationally feasible* way of decoding them that is good, though not optimal.

We can construct LDPC codes randomly, in various ways. One way: to make an $[n, k]$ code, randomly generate columns of H with exactly three 1s in them.

For best results, equalize the number of 1s in each row (as much as possible) by randomly picking the position of the three 1s in the next column from among rows that don't already have $3n/(n-k)$ 1s in them.

Example: A $[50, 25]$ LDPC Code

Here's the parity-check matrix for a small LDPC code (three 1s in each column, six in each row).

```

001100001010000000000000100000000000000001000
000000000000000000001010000010000100100000010000
0010101000000000001000100000100000000000000000000
000001010000000010000000000000000000000110010000000
00000000100000000010000000000110000000001000010
0000100000100000000010000000000100000000001010
0000000000101000100000000000001001000010000000
0100010000100000010000000000000100000000000001
0000000000000011010000000010000001000010000000
010000000001000000000011001000000000001000000000
00000000010101000000000000000000000011000010000
0000000000000000001000000010000100000000000001101
00100000000010000001100100000000100000000000000
100000100000000100010000001010000000000000000000
0001001000100000000000010000001010000000000000
00000010000010000001000000000100000000011000000
10000000100001000000000001000000000000000000101
0000000000000000010001000100001000001000000100000
001010001000000010000000000001000010000000000
10000000010000000000000001000000100100000000100
000000001000000000000110000001000000011000000000
0000100000000000100110000010000000000000100000000
000000001000011000000000100000000100000000100000
010000000000000000000100100000000000001100100000
00001000000000000000000000000110000000000110010

```

A Generator Matrix for the Example

A systematic generator matrix obtained from the parity-check matrix (with columns re-ordered):

```

10000000000000000000000000110000011111011101001010
010000000000000000000000001001010101001101110111000
0010000000000000000000000011001100110000100111000
00010000000000000000000000101101011111110010010
000010000000000000000000001010011000110001000100001
00000100000000000000000000111000001111010001011011
00000010000000000000000000111101011111010001011011
000000010000000000000000001011011011011100010011011
00000000100000000000000000100101000110011001000100
000000000100000000000000001001011001111010000110000
00000000001000000000000000100001010001100110100011
000000000001000000000000001000010000000000001101
000000000000100000000000000011000000000000110010
00000000000001000000000000101001010000100110100111
00000000000000010000000000001001100110101110000011
00000000000000001000000000111101100100011101001001
000000000000000001000000010000001001101010101010000
00000000000000000010000000100000010011010101010000
000000000000000000010000000100000011001111010100010000
00000000000000000000100000011001111010100010000
000000000000000000000100000010001001100110011001100
0000000000000000000000100001110010000010101000101000
000000000000000000000001000000011001111010100010000
00000000000000000000000010001111101000010110110000000
00000000000000000000000010011001000001110101111100
0000000000000000000000000100111010000010110110000000
0000000000000000000000000010011001000001110101111100
0000000000000000000000000011001110100000101110101000

```

Decoding LDPC Codes

To encode a message with an LDPC, we just multiply it by the generator matrix. But how do we decode?

The optimal method (assuming a BSC, and equally-probable messages) is to pick the codeword nearest to what was received. But this is computationally infeasible.

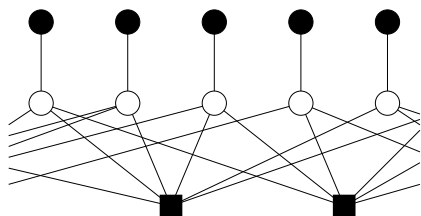
The reason LDPC codes are interesting is that the sparseness of their parity-check matrices allows for an *approximate* (good, but not optimal) decoding method that works by *propagating probabilities* through a graph.

Graphical Representation of a Code

We can represent a code by a graph:

- Empty circles represent bits of a codeword.
- Black circles represent received data bits.
- Black squares represent parity checks.

Here's a fragment of such a graph:



Notice that each codeword bit connects to three parity checks — corresponding to the three 1s in each column of H . Each parity check connects to six codeword bits.

Our task: Fill in the empty circles.

Decoding by Propagating Probabilities

We can't be absolutely sure of the codeword bits, but we can keep track of the *odds* in favour of 1 over 0 (the ratio of the probability of 1 over the probability of 0).

Each black node will send each codeword bit it connects to a message giving its idea of what the odds for 1 over 0 should be for that bit.

All the messages a codeword receives are multiplied to give the current idea of what the odds are for that bit — used to guess the codeword once these odds have stabilized.

But first, we iterate: Each codeword bit sends each parity check it connects to a message with its current odds, which the parity check node uses to update its messages to other codeword bits. Messages propagate until the odds have stabilized.

Details of the Messages

Received data bit to codeword bit: Odds sent are $(1-Q)/Q$ if the received data is 1, $Q/(1-Q)$ if the received data is 0.

Parity check to codeword bit: Message is the probability of the parity check being satisfied if that bit is 1, divided by the probability if that bit is 0. These probabilities are calculated based on that parity check's idea of the odds for the *other* bits in the parity check being 1 versus 0.

Codeword bit to a parity check: Message is the odds of the bit being 1 versus 0, based on the received data, and on the messages from the *other* parity checks the codeword bit is involved in.

Avoiding Double-Counting Information

Messages sent between codeword bits and parity checks exclude information obtained from the node the message is being sent to. This avoids undesirable "double-counting" of information when a message comes back from that node.

But: This works perfectly only if the graph is a tree. If there are cycles in the graph, information can return to its source indirectly.

This is why probability propagation is only an *approximate* decoding method. It works well up to a point, but doesn't have as low an error rate as nearest-neighbor (maximum likelihood) decoding would achieve.

Demonstration of LDPC Codes

I tried rate 1/2 LDPC codes with three bits in each column of H , with varying codeword lengths, tested using a BSC with varying error probability, Q (and hence capacity, C).

Here are the block error rates for three such codes, estimated from 1000 simulated messages:

Q	C	[100, 50]	[1000, 500]	[10000, 5000]
0.02	0.86	0.000	0.000	0.000
0.03	0.81	0.012	0.000	0.000
0.04	0.76	0.059	0.000	0.000
0.05	0.71	0.108	0.000	0.000
0.06	0.67	0.213	0.005	0.000
0.07	0.63	0.327	0.104	0.000
0.08	0.60	0.482	0.404	0.125

Tests were done with software available from my web page, <http://www.cs.utoronto.ca/~radford/>

History of LDPC and Related Codes

- Gallager, LDPC codes — 1961.
True merits not realized? Computers too slow? Largely ignored and forgotten.
- Berrou, *et al*, TURBO codes — 1993.
Surprisingly good codes, practically decodable, but not really understood.
- MacKay and Neal — 1995.
Reinvent LDPC codes, slightly improved. Show they're almost as good as TURBO codes. Decoding algorithm related to other probabilistic inference methods.
- Many (Richardson, Frey, etc.) — ongoing.
Further improvements in LDPC codes, relationship to TURBO codes, theory of why it all works.