Finding Good LDPC Codes

Daniel A. Spielman*

Dept. of Mathematics
Massachusetts Institute of Technology
Cambridge, MA 02139
spielman@math.mit.edu

Abstract

We develop heuristics for finding good distributions of irregular low density parity check (LDPC) codes. Using these heuristics, we construct rate 1/2 LDPC codes that have lower bit error rates than Turbo Codes at very low signal-to-noise ratios.

1 Introduction

In 1993, the debut of turbo codes [1] along with experiments demonstrating their remarkably low bit-error-rates at very low signal-to-noise ratios excited the coding theory community. A few years later, low density parity check codes [2] reemerged through experiments demonstrating [7, 8] that they too allow low bit-error-rates at low signal-to-noise ratios, although not as low at turbo codes. Since then, LDPC codes have been catching up. Improvements over Gallager's initial construction were made by using carefully chosen irregular graphs [6] and by using fields other than GF(2) [9]. Recently, Davey and MacKay [10] have constructed irregular rate 1/4 LDPC codes over GF(8) that beat Turbo Codes at very low SNRs. In this work, we construct rate 1/2 irregular LDPC codes that beat Turbo Codes at very low SNRs. Our codes are designed to work over $GF(2)$, so we expect to see better performance after we incorporate larger fields into our techniques.

Our codes were constructed by heuristic algorithms designed to find good degree sequences for irregular LDPC codes. These algorithms are a generalization of those used in [5]. As our present implementations of these algorithms are quite slow, we have not yet had enough time to understand their full potential. We expect such understanding to occur only after we have optimized the implementations of the heuristics. The codes presented in this paper were the result of applying one heuristic once, and another twice. This took several days.

All results in this paper are presented for the Gaussian channel.

1.1 Irregular Codes

A LDPC code is specified by a bipartite graph between a set of n *symbol nodes* and a set of $(1-r)n$ *check nodes*, where n is the block length of the code and r is the designed

*Supported in part by NSF CAREER Award CCR-9701304, an Alfred P. Sloan Foundation Fellowship, and a Sloan/Cabot Award from the M.I.T. School of Science

rate[1] Each symbol node is associated with a bit to be transmitted, and the check nodes impose linear constraints upon those bits.

When Gallager[2] introduced LDPC codes, he proposed using bipartite graphs in which the degrees of nodes on each side of the graph were uniform (*e.g.,* all the symbol nodes could could have degree 3 and all the check nodes could have degree 6). We will call such LDPC codes "regular".

For this work, we will build LDPC codes from the types of irregular graphs suggested in [4, 5]. These graphs are specified by a symbol node degree sequence $(\lambda_1, \lambda_2, \lambda_3, \ldots)$ and a check node degree sequence $(\rho_1, \rho_2, \rho_3, \ldots)$, where λ_i (ρ_i) specifies the proportion of symbol nodes (check nodes) of degree i, and

$$\sum_i \lambda_i = 1, \qquad \sum_i \rho_i = 1.$$

Given two degree sequences, we form a LDPC code by choosing a random bipartite graph in which the proportions of nodes of each degree is given by the appropriate term in one of the degree sequences.

In [4], we proved that, by carefully choosing the degrees of nodes on each side of the graph, it was possible to produce LDPC codes that quickly approached the capacity of the erasure channel. In [6], we presented experiments which showed that some irregular LDPC codes from [4] performed better under belief propagation decoding than any regular LDPC codes. In [5], we analyzed the performance of a discretized version of belief propagation, and showed that irregular LDPC codes performed better than regular LDPC codes when decoded using this algorithm. The main tool of [5] was a linear programing based algorithm that, given a degree sequence for the check nodes, would find an optimal degree sequence for the symbol nodes, and *vice versa.* In this work, we adapt this approach to full belief propagation. However, to make this adaptation, we must sacrifice guarantees of optimality.

1.2 Belief Propagation

In this section, we sketch how the belief propagation algorithm works. In this description, we assume that all codewords are equally likely and we only associate one bit with each symbol node. We refer readers desiring a more complete description to [7, 12].

The belief propagation algorithm (a.k.a. the sum-product algorithm) begins by considering the received symbols and computing, for each symbol node, the probability that the associated bit is 0 given the received symbol for that bit. Each edge in the graph is then assigned the value of the symbol node to which it is attached. After this initialization, the algorithm proceeds by alternating between *check passes* and *message passes.*

We understand a check pass by imagining a random 0/1 variable associated with each edge, with the value of the edge being the probability its variable is zero. During a check pass, each edge is assigned the value of the probability that the sum of the variables associated with the other edges connected to its check has parity zero, assuming that the variables are independent. In particular, the value an edge is assigned during a check pass is not a function of the value of that edge before the check pass. To make this explicit, consider a check node with values $p_1, \ldots, p_k$ on its edges. After the check pass, these edges will carry the values

[1] It is possible for the rate of the code to be higher than $(1-r)n$ if there are redundancies among the constraints imposed by the check nodes.

$$\oplus_{k-1}(p_2, \ldots p_k), \oplus_{k-1}(p_1, p_3, \ldots p_k), \ldots, \oplus_{k-1}(p_1, \ldots, p_{k-1}),$$

respectively, where

$$\oplus_{k-1}(p_1, \ldots, p_{k-1}) = \frac{1 + \prod_i (2p_i - 1)}{2}.$$

To understand a message pass, we imagine that each edge carries a random variable obtained by sending the bit associated with its symbol node through some channel, with the value of the edge being the probability that the bit was 0 zero given that the variable was received. During a message pass, each edge is assigned the probability that the bit associated with its symbol node is 0, assuming that the received symbol for that bit and the values carried by the other edges of that symbol node represent independent observations of the bit. For a symbol node, if p_0 is the probability that the bit associated with the node is zero given the received symbol for that bit, and if the values on the edges connected to that symbol node are $p_1, \ldots p_k$, then the values on those edges after the message pass will be

$$\otimes_k(p_0, p_2, \ldots, p_k), \otimes_k(p_0, p_1, p_3, \ldots, p_k), \ldots \otimes_k (p_0, p_1, \ldots, p_{k-1}),$$

where

$$\otimes_k = \frac{\prod_i p_i}{\prod_i p_i + \prod_i (1 - p_i)}.$$

It will later be useful to note that

$$\otimes_k(p_0, p_1, \ldots p_k) = \otimes_2(p_0, \oplus_{k-1}(p_1, \ldots p_{k-1})).$$

After some number of rounds, the algorithm is stopped after a check pass. The output of the algorithm is obtained by assigning each symbol node the best estimate of its bit's value that can be obtained from the symbol received for that bit and the values of the edges connected to that symbol node, assuming that each of these values comes from an independent measurement. In the language of the previous paragraph, this quantity is $\otimes_{k+1}(p_0, p_1, \ldots p_k)$.

Unless the graph underlying the LDPC code is a tree, the above independence assumptions can be false. When the graph is a tree, this algorithm is known to converge to the best a posteriori estimate for each bit[11, 12]. In experiments with randomly generated LDPC codes, it has been observed that this algorithm converges to a very good estimate for each bit, even though the underlying graph is not a tree. This observation has led to the conjecture that the performance of the belief propagation algorithm on LDPC codes derived from random graphs is almost identical to the performance of the belief propagation algorithm on an infinite tree, and that it becomes more similar as the block size of the LDPC code grows. One justification for this conjecture is that the expected girth of a random graph with fixed degree sequences grows logarithmically in the number of nodes in the graph. Thus, the first few rounds of belief propagation decoding do accurately reflect the performance of belief propagation decoding in an infinite tree.

While it is known that the belief propagation algorithm does converge on infinite trees, we are unaware of any simple formula for what it converges to. That is, given the structure of the infinite tree, given an initial SNR, and given a number of rounds, we know of no simple way of determining the bit error rate after that number of rounds. In

fact, we are unable to do this for **any** non-trivial[2] pair of degree sequences even as n goes to infinity!

However, given a pair of degree sequences, an initial SNR and a number of rounds, we can sample the distribution of values on edges after that number of rounds. For convenience, we assume that the all-0 codeword was transmitted. Let M_0 denote the distribution of values on edges after the initialization of the belief propagation algorithm. Let C_1 denote the distribution after the first check pass, M_1 denote the distribution after the first message pass, and so on. We can sample M_0 by simulating the Gaussian channel. We can sample C_i by performing check passes on samples from M_{i-1}, and we can sample M_i by performing message passes on samples from C_{i-1}, and so on.

To make this precise, it helps to first consider the infinite tree corresponding to a regular graph. Assume that every symbol node has degree l and that every check node has degree r. Then, we sample from C_i by drawing $r-1$ samples from M_{i-1}, say $p_1, \ldots p_{r-1}$, and computing

$$\otimes_{r-1}(p_1, \ldots p_{r-1}).$$

Similarly, we sample from M_i by drawing $l-1$ samples from C_{l-1}, say $p_1, \ldots p_{l-1}$ and one sample from M_0, say p_0, and computing

$$\oplus_l(p_0, p_1, \ldots p_{l-1}).$$

We will overload our notation by expressing these relations between distributions as

$$\begin{aligned} C_i &= \bigoplus_{r-1}(\overbrace{M_{i-1}, \ldots, M_{i-1}}^{r-1}), and \\ M_i &= \bigotimes_{l}(M_0, \overbrace{C_i, \ldots C_i}^{l-1}). \end{aligned}$$

If we apply this approach to sampling precisely, then we are actually simulating the process on a very large tree. As this requires immense amounts of time, we instead approximate these distributions by generating a large set of samples from each. Whenever we need a sample from a distribution, we just pick one at random from the pre-sampled set. Naturally, we form our set of samples corresponding to C_i before those corresponding to M_i, and so on. For convenience, we will assume that our set of samples for each space approximates that which we would get if we chose independent samples from that space.

As the correct definition of the infinite tree corresponding to an irregular graph specified by a pair of degree sequences is not transparent to the author, we will settle for an explanation of how to sample from the corresponding distributions C_i and M_i. First, note that the probability that a randomly chosen edge is connected to a symbol node of degree i is

$$\frac{(\lambda_i \cdot i)}{\sum_j (\lambda_j \cdot j)}.$$

It is important to note that this probability is very different from the probability that a randomly chosen symbol node has degree i. For convenience, set $\lambda'_i = (i\lambda_i)/(\sum_j j\lambda_j)$ and $\rho'_i = (i\rho_i)/(\sum_j j\rho_j)$. To sample from M_i we, with probability λ'_j, compute $\otimes_j(p_0, p_1, \ldots, p_{j-1})$,

[2]We would consider the graphs in which symbol nodes have degree 3 and check nodes have degree 6 to be non-trivial.

where p_0 is drawn from M_0 and $p_1, \ldots p_{j-1}$ are drawn from C_j. We write this relation as:

$$M_i = \sum_j \lambda'_j \bigotimes_j (M_0, \overbrace{C_j, \ldots, C_j}^{j-1}), \text{ and} \quad (1)$$

$$C_i = \sum_j \rho'_j \bigoplus_{j-1} (\overbrace{M_{j-1}, \ldots, M_{j-1}}^{j-1}), \quad (2)$$

where by a linear combination of probability spaces with coefficient sum one, we mean that each is sampled with probability according to its coefficient.

2 The LP Technique

The goal of the linear programming technique[5] is, given one degree sequence and an initial noise level, to find a complementary degree sequence for which the probability of bit error goes to zero as the number of decoding rounds increases. In[5] this task was simplified by the fact that the distributions M_i and C_i produced by the discretized belief propagation algorithm could be completely specified by one parameter. Moreover, the parameter specifying the distribution M_i (C_i) could be obtained from a simple formula in the parameter specifying the distribution C_i (M_{i-1}).

In our case, we cannot simply characterize the distributions M_i and C_i encountered in the belief propagation algorithm. Instead, we work with sets of samples believed to approximate these distributions as described in the previous section.

Given a check node degree sequence and an initial signal-to-noise ratio, we want to set up a linear program that will find a symbol node degree sequence such that the distribution M_i is better than the distribution M_{i-1} for all i, if such a distribution exists. To measure the quality of a distribution, we use the objective function:

$$F(M_i) = -\int_p log(p) d\mu_i(p), \quad (3)$$

where μ_i is the measure associated with distribution M_i. The smaller $F(M_i)$ is, the better we consider M_i to be. We work with (3) because it is easy to approximate from our set of samples and because it is clearly respects formula (1). That is,

$$-\int_p log(p) d\mu_i(p) = -\sum_j \lambda'_j \int_p log(p) d\mu_{i,j}(p),$$

where $\mu_{i,j}$ is the measure corresponding to the distribution

$$M_{i,j} = \bigotimes_j (M_0, \overbrace{C_i, \ldots, C_i}^{j-1}).$$

It is almost possible to construct a linear program that, given M_0 and a check node degree sequence, will solve for λ_is so that $F(M_i) < F(M_{i-1})$ for all i. Unfortunately, the dependence of M_i upon M_{i-1} through C_i prevents us from solving these inequalities simultaneously.

3 Our Heuristics

We tried two approaches to finding good degree sequences. These were:

(a) **ignore the dependencies:** In this approach, we fix symbol and check degree sequences, generate the corresponding distributions $M_0, \ldots M_{k-1}$ and $C_1, \ldots C_k$ for some k, and then try to find a symbol (check) degree sequence that would do better than the original one with input distributions $C_1, \ldots, C_k$ ($M_0, \ldots M_{k-1}$). That is, given $(\lambda'_i)_i$ and $(\rho'_i)_i$, we try to find $(\alpha'_i)_i$ so that

$$\sum_i \lambda'_i F(\bigotimes_i (M_0, \overbrace{C_j, \ldots, C_j}^{i-1})) > \sum_i \alpha'_i F(\bigotimes_{i-1} (M_0, \overbrace{C_j, \ldots, C_j}^{i-1})),$$

for all j, while keeping the rate of the resulting code the same.

(b) **Pretend that all distributions are like the input distribution:** For any initial signal-to-noise ratio, we get an initial distribution M_0. Construct many such initial distributions $M_{0,l}$ corresponding to increasing signal-to-noise ratios as l grows. For some check degree sequence $(\rho'_i)_i$, use linear programming to find a degree sequence $(\lambda'_i)_i$ such that

$$\sum_i \lambda'_i F(\bigotimes_i (M_0, \overbrace{C_l, \ldots C_l}^{i-1})) < F(\bigotimes_2 (M_0, M_{0,l})),$$

where

$$C_l = \sum_i \rho'_i \bigoplus_{i-1} (\overbrace{M_{0,l}, \ldots, M_{0,l}}^{i-1}).$$

That is, we just try to find symbol degree sequences such that if the distributions after message passes looked like the initial distributions, each successive round would have better distributions.

4 Results

The experiments reported here were done on LDPC codes built from degree sequences obtained by combining approaches (a) and (b). We first used approach (b) to find a good symbol degree sequence given the check degree sequence $\rho_9 = 1$. We then used approach (a) to refine this symbol degree sequence, keeping the same check degree sequence. We then refined the symbol degree sequence once more.

The symbol node degree sequence we settled on was ($\lambda_2 = .4804, \lambda_3 = .2797, \lambda_6 = .0138, \lambda_7 = .1830, \lambda_{31} = .0431$).

in Figure 1 and Figure 2, we show the data obtained in simulations of rate 1/2 codes of lengths 15824 and 32242 respectively. In Figure 4, we compare the performance of these two codes with that of Turbo codes reported in [3]. The points on the lowest curve at .6 and .7 db should be viewed with suspicion until we have the results of more simulations. The same holds for the highest point at 8 db. We expect that when we perform more simulations with more decoding rounds in each trial, the middle columns of Figures 1 and 2 will better resemble the last columns of these Figures.

SRN Db	rounds	trials	BER	< 15700	$< 13K$	$< 11K$	$< 10K$
.4	250	500	.1627	298	243	146	55
.45	250	1500	.0918	556	402	227	83
.5	150	3000	.0505	694	429	233	89
.6	150	2000	.0098	111	55	29	11
.7	120	6000	.00104	52	12	7	3
.8	120	6000	1.29e-04	8	0	0	0

Figure 1: This code has block length 15842. For each signal-to-noise level, we list the number of rounds of belief propagation that were used to produce these results, the number of trials at this noise level, the bit error rate over all these trials, and the number of times the decoding algorithm got less that 15700, 13000, 11000, and 10000 of the bits correct.

SRN Db	rounds	trials	BER	$< 32K$	$< 27K$	$< 24K$	$< 20K$
.4	250	500	.0831	183	134	81	8
.45	250	1500	.0371	265	175	103	6
.5	150	1200	.00849	68	28	16	3
.6	150	2000	3.38e-04	3	1	1	0
.7	120	7400	1.04e-04	6	0	0	0
.7	150	4000	1.94e-04	3	1	1	1

Figure 2: This code has block length 32242. For each signal-to-noise level, we list the number of rounds of belief propagation that were used to produce these results, the number of trials at this noise level, the bit error rate over all these trials, and the number of times the decoding algorithm got less that 32000, 27000, 24000, and 20000 of the bits correct.

5 Failed Attempts

We also tried to use approach (a) to obtain better check degree sequences. While the linear program did return more interesting check degree sequences, these did not appear to perform as well in our simulations. In fact, we observed problematic behavior: some of the codes we produced would make quick progress initially, but then slow down to a crawl and occasionally get stuck with very few errors remaining.

However, we should note that we only made a few attempts at finding good check degree sequences, and it might be that if we tried a few more times we would have better luck.

6 Conclusions and Future Work

At the moment, we are unsure whether this approach can be used to find substantially better LDPC codes over $GF(2)$. So far, the work has progressed slowly

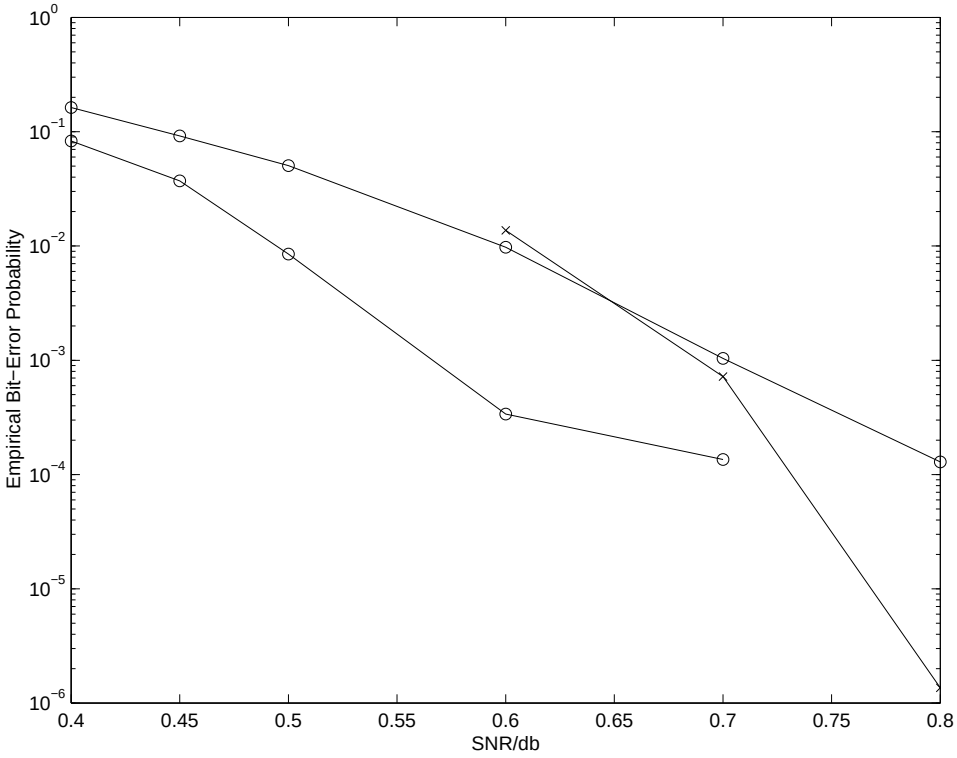

Figure 3: The leftmost two curves were generated from the data in Figures 1 and 2. The other curve is from the 1/2 rate Turbo code of [3] The points on the lowest curve at .6 and .7 db should be viewed with suspicion until we have the results of more simulations. The same holds for the highest point at 8 db.

because we implemented our algorithms in Matlab. This was quite slow: just obtaining data for approach (a) or (b) with reasonable numerical accuracy could take a whole day. We are presently translating the algorithms to C, and are observing a dramatic speedup. This should make it possible to explore these approaches more fully.

The obvious next step in this research is to apply these techniques to the construction of codes over fields other than $GF(2)$. We will do this as soon as we have completed our faster implementation. Given how much better regular LDPC codes over $GF(8)$ are than regular codes over $GF(2)$, and given how much better these irregular LDPC codes over $GF(2)$ are than the regular codes over $GF(2)$, we presume that terrific results will result from combining the two appraoches.

References

[1] C. Berrou, A Glavieux, and P. Thitimajshima, "Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes", *Proceedings of IEEE International Communications Conference, 1993.*

[2] R. G. Gallager, **Low-Density Parity-Check Codes**, MIT Press, 1963.

[3] JPL. Turbo code performance. Available from **http://www331.jpl.nasa.gov/public/TurboPerf.html**, September 1998.

[4] M. Luby, M. Mitzenmacher, M. A. Shokrollahi, D. A. Spielman, and V. Stemann, "Practical Loss-Resilient Codes", *Proc.* 29^{th} *ACM Symposium on Theory of Computing*, 1997, pp. 150–159.

[5] M. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. A. Spielman, "Analysis of Low Density Codes and Improved Designs Using Irregular Graphs", *Proceedings of the* 30^{th} *ACM Symposium on Theory of Computing*, 1998, pp. 249–258.

[6] M. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. A. Spielman, "Improved Low Density Parity Check Codes Using Irregular Graphs and Belief Propagation", submitted to the *1998 International Symposium on Information Theory.*

[7] D. J. C. MacKay and R. M. Neal, "Good Error Correcting Codes Based on Very Sparse Matrices", to appear in *IEEE Transactions on Information Theory.* Available from **http://wol.ra.phy.cam.ac.uk/mackay**.

[8] D. J. C. MacKay and R. M. Neal, "Near Shannon Limit Performance of Low Density Parity Check Codes", to appear in *Electronic Letters.*

[9] M. C. Davey and D. J. C. MacKay, "Low Density Parity Check Codes over $GF(q)$", early manuscript, 1997.

[10] M. C. Davey and D. J. C. MacKay, "Low Density Parity Check Codes over $GF(q)$", *Information Theory Workshop 1998, Killarney, Ireland.*

[11] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, Morgan Kaufmann Publishers, 1988.

[12] N. Wiberg, "Codes and decoding on general graphs" Ph.D. dissertation, Dept. Elec. Eng, U. Linköping, Sweeden, April 1996.