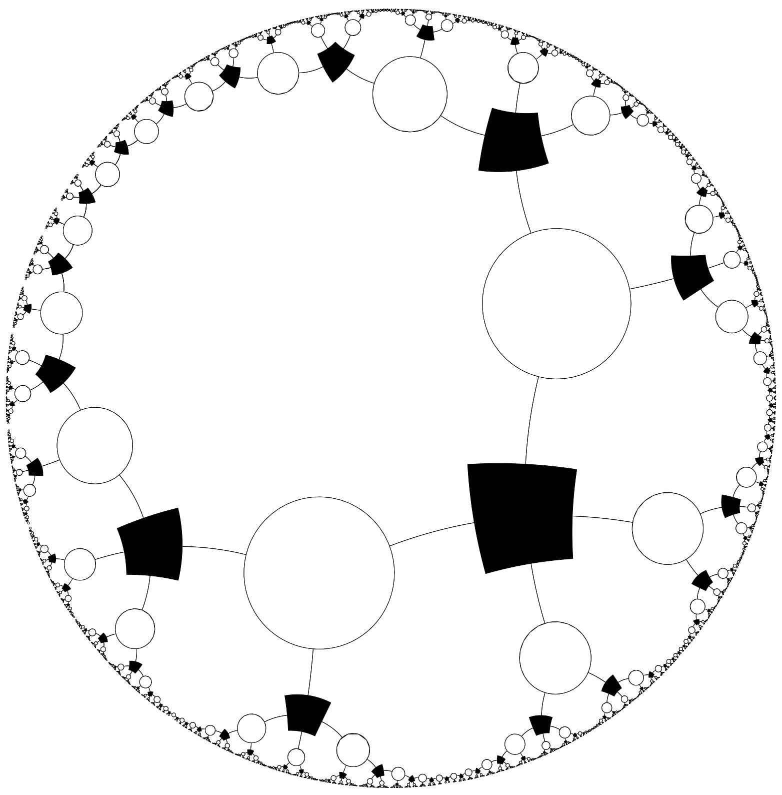
Part VI

Sparse Graph Codes



About Part VI

The central problem of communication theory is to construct an encoding and a decoding system that make it possible to communicate reliably over a noisy channel. During the 1990s, remarkable progress was made towards the Shannon limit, using codes that are defined in terms of sparse random graphs, and which are decoded by a simple probability-based message-passing algorithm.

In a *sparse-graph code*, the nodes in the graph represent the transmitted bits and the constraints they satisfy. For a linear code with a codeword length N and rate $R = K/N$, the number of constraints is of order $M = N - K$. Any linear code can be described by a graph, but what makes a sparse-graph code special is that each constraint involves only a small number of variables in the graph: so the number of edges in the graph scales roughly linearly with N, rather than quadratically.

In the following four chapters we will look at four families of sparse-graph codes: three families that are excellent for error-correction: *low-density parity-check codes*, *turbo codes*, and *repeat–accumulate codes*; and the family of *digital fountain codes*, which are outstanding for erasure-correction.

All these codes can be decoded by a local message-passing algorithm on the graph, the sum–product algorithm, and, while this algorithm is not a perfect maximum likelihood decoder, the empirical results are record-breaking.

47

Low-Density Parity-Check Codes

A low-density parity-check code (or Gallager code) is a block code that has a parity-check matrix, **H**, every row and column of which is 'sparse'.

A *regular* Gallager code is a low-density parity-check code in which every column of **H** has the same weight j and every row has the same weight k; regular Gallager codes are constructed at random subject to these constraints. A low-density parity-check code with $j = 3$ and $k = 4$ is illustrated in figure 47.1.

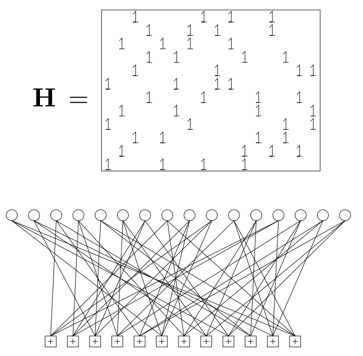


Figure 47.1. A low-density parity-check matrix and the corresponding graph of a rate-1/4 low-density parity-check code with blocklength $N = 16$, and $M = 12$ constraints. Each white circle represents a transmitted bit. Each bit participates in $j = 3$ constraints, represented by $\boxed{+}$ squares. Each constraint forces the sum of the $k = 4$ bits to which it is connected to be even.

▶ 47.1 Theoretical properties

Low-density parity-check codes lend themselves to theoretical study. The following results are proved in Gallager (1963) and MacKay (1999b).

Low-density parity-check codes, in spite of their simple construction, are good codes, *given an optimal decoder* (good codes in the sense of section 11.4). Furthermore, they have good distance (in the sense of section 13.2). These two results hold for any column weight $j \geq 3$. Furthermore, there are sequences of low-density parity-check codes in which j increases gradually with N, in such a way that the ratio j/N still goes to zero, that are *very good*, and that have very good distance.

However, we don't have an optimal decoder, and decoding low-density parity-check codes is an NP-complete problem. So what can we do in practice?

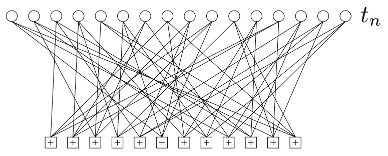
▶ 47.2 Practical decoding

Given a channel output **r**, we wish to find the codeword **t** whose likelihood $P(\mathbf{r} \mid \mathbf{t})$ is biggest. All the effective decoding strategies for low-density parity-check codes are message-passing algorithms. The best algorithm known is the sum–product algorithm, also known as iterative probabilistic decoding or belief propagation.

We'll assume that the channel is a memoryless channel (though more complex channels can easily be handled by running the sum–product algorithm on a more complex graph that represents the expected correlations among the errors (Worthen and Stark, 1998)). For any memoryless channel, there are two approaches to the decoding problem, both of which lead to the generic problem 'find the **x** that maximizes

$$P^*(\mathbf{x}) = P(\mathbf{x})\, \mathbb{1}[\mathbf{H}\mathbf{x} = \mathbf{z}]', \tag{47.1}$$

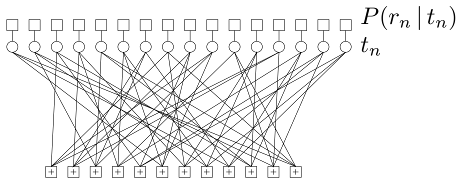
where $P(\mathbf{x})$ is a separable distribution on a binary vector **x**, and **z** is another binary vector. Each of these two approaches represents the decoding problem in terms of a factor graph (Chapter 26).



(a) The prior distribution over codewords

$$P(\mathbf{t}) \propto \mathbb{1}[\mathbf{Ht} = \mathbf{0}].$$

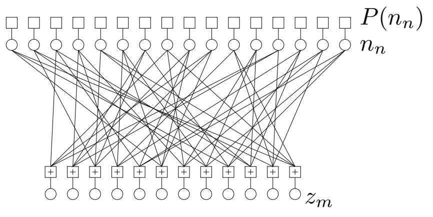
The variable nodes are the transmitted bits $\{t_n\}$.
Each $\boxplus$ node represents the factor $\mathbb{1}[\sum_{n\in\mathcal{N}(m)} t_n = 0 \bmod 2]$.



(b) The posterior distribution over codewords,

$$P(\mathbf{t} \,|\, \mathbf{r}) \propto P(\mathbf{t})P(\mathbf{r} \,|\, \mathbf{t}).$$

Each upper function node represents a likelihood factor $P(r_n \,|\, t_n)$.



(c) The joint probability of the noise $\mathbf{n}$ and syndrome $\mathbf{z}$,

$$P(\mathbf{n}, \mathbf{z}) = P(\mathbf{n})\, \mathbb{1}[\mathbf{z} = \mathbf{Hn}].$$

The top variable nodes are now the noise bits $\{n_n\}$.
The added variable nodes at the base are the syndrome values $\{z_m\}$.
Each definition $z_m = \sum_n H_{mn} n_n \bmod 2$ is enforced by a $\boxplus$ factor.

Figure 47.2. Factor graphs associated with a low-density parity-check code.

The codeword decoding viewpoint

First, we note that the prior distribution over codewords,

$$P(\mathbf{t}) \;\propto\; \mathbb{1}[\mathbf{Ht} = \mathbf{0} \bmod 2], \tag{47.2}$$

can be represented by a factor graph (figure 47.2a), with the factorization being

$$P(\mathbf{t}) \;\propto\; \prod_m \mathbb{1}[\sum_{n\in\mathcal{N}(m)} t_n = 0 \bmod 2]. \tag{47.3}$$

(We'll omit the 'mod 2's from now on.) The posterior distribution over codewords is given by multiplying this prior by the likelihood, which introduces another N factors, one for each received bit.

$$\begin{aligned} P(\mathbf{t} \,|\, \mathbf{r}) &\;\propto\; P(\mathbf{t})P(\mathbf{r} \,|\, \mathbf{t}) \\ &\;\propto\; \prod_m \mathbb{1}[\sum_{n\in\mathcal{N}(m)} t_n = 0] \prod_n P(r_n \,|\, t_n) \end{aligned} \tag{47.4}$$

The factor graph corresponding to this function is shown in figure 47.2b. It is the same as the graph for the prior, except for the addition of likelihood 'dongles' to the transmitted bits.

In this viewpoint, the received signal r_n can live in any alphabet; all that matters are the values of $P(r_n \,|\, t_n)$.

The syndrome decoding viewpoint

Alternatively, we can view the channel output in terms of a binary received vector $\mathbf{r}$ and a noise vector $\mathbf{n}$, with a probability distribution $P(\mathbf{n})$ that can be derived from the channel properties and whatever additional information is available at the channel outputs.

For example, with a binary symmetric channel, we define the noise by $\mathbf{r} = \mathbf{t} + \mathbf{n}$, the syndrome $\mathbf{z} = \mathbf{Hr}$, and noise model $P(n_n = 1) = f$. For other channels such as the Gaussian channel with output $\mathbf{y}$, we may define a received binary vector $\mathbf{r}$ however we wish and obtain an effective binary noise model $P(\mathbf{n})$ from $\mathbf{y}$ (exercises 9.18 (p.155) and 25.1 (p.325)).

The joint probability of the noise $\mathbf{n}$ and syndrome $\mathbf{z} = \mathbf{Hn}$ can be factored as

$$\begin{aligned} P(\mathbf{n}, \mathbf{z}) &= P(\mathbf{n})\, \mathbb{1}[\mathbf{z} = \mathbf{Hn}] \\ &= \prod_n P(n_n) \prod_m \mathbb{1}[z_m = \sum_{n \in \mathcal{N}(m)} n_n]. \end{aligned} \tag{47.5}$$

The factor graph of this function is shown in figure 47.2c. The variables $\mathbf{n}$ and $\mathbf{z}$ can also be drawn in a 'belief network' (also known as a 'Bayesian network', 'causal network', or 'influence diagram') similar to figure 47.2a, but with arrows on the edges from the upper circular nodes (which represent the variables $\mathbf{n}$) to the lower square nodes (which now represent the variables $\mathbf{z}$). We can say that every bit x_n is the parent of j checks z_m, and each check z_m is the child of k bits.

Both decoding viewpoints involve essentially the same graph. Either version of the decoding problem can be expressed as the generic decoding problem 'find the $\mathbf{x}$ that maximizes

$$P^*(\mathbf{x}) = P(\mathbf{x})\, \mathbb{1}[\mathbf{Hx} = \mathbf{z}]\text{'}; \tag{47.6}$$

in the codeword decoding viewpoint, $\mathbf{x}$ is the codeword $\mathbf{t}$, and $\mathbf{z}$ is 0; in the syndrome decoding viewpoint, $\mathbf{x}$ is the noise $\mathbf{n}$, and $\mathbf{z}$ is the syndrome.

It doesn't matter which viewpoint we take when we apply the sum–product algorithm. The two decoding algorithms are isomorphic and will give equivalent outcomes (unless numerical errors intervene).

> I tend to use the syndrome decoding viewpoint because it has one advantage: one does not need to implement an *encoder* for a code in order to be able to simulate a decoding problem realistically.

We'll now talk in terms of the generic decoding problem.

▶ 47.3 Decoding with the sum–product algorithm

We aim, given the observed checks, to compute the marginal posterior probabilities $P(x_n = 1 \mid \mathbf{z}, \mathbf{H})$ for each n. It is hard to compute these exactly because the graph contains many cycles. However, it is interesting to implement the decoding algorithm that would be appropriate if there were no cycles, on the assumption that the errors introduced might be relatively small. This approach of ignoring cycles has been used in the artificial intelligence literature but is now frowned upon because it produces inaccurate probabilities. However, if we are decoding a good error-correcting code, we don't care about accurate marginal probabilities – we just want the correct codeword. Also, the posterior probability, in the case of a good code communicating at an achievable rate, is expected typically to be hugely concentrated on the most probable decoding; so we are dealing with a distinctive probability distribution to which experience gained in other fields may not apply.

The sum–product algorithm was presented in Chapter 26. We now write out explicitly how it works for solving the decoding problem

$$\mathbf{Hx} = \mathbf{z} \pmod 2.$$

For brevity, we reabsorb the dongles hanging off the x and z nodes in figure 47.2c and modify the sum–product algorithm accordingly. The graph in which $\mathbf{x}$ and $\mathbf{z}$ live is then the original graph (figure 47.2a) whose edges are defined by the 1s in $\mathbf{H}$. The graph contains nodes of two types, which we'll call checks and bits. The graph connecting the checks and bits is a bipartite graph: bits connect only to checks, and *vice versa.* On each iteration, a probability ratio is propagated along each edge in the graph, and each bit node x_n updates its probability that it should be in state 1.

We denote the set of bits n that participate in check m by $\mathcal{N}(m) \equiv \{n : H_{mn} = 1\}$. Similarly we define the set of checks in which bit n participates, $\mathcal{M}(n) \equiv \{m : H_{mn} = 1\}$. We denote a set $\mathcal{N}(m)$ with bit n excluded by $\mathcal{N}(m)\backslash n$. The algorithm has two alternating parts, in which quantities q_{mn} and r_{mn} associated with each edge in the graph are iteratively updated. The quantity q^x_{mn} is meant to be the probability that bit n of $\mathbf{x}$ has the value x, given the information obtained via checks other than check m. The quantity r^x_{mn} is meant to be the probability of check m being satisfied if bit n of $\mathbf{x}$ is considered fixed at x and the other bits have a separable distribution given by the probabilities $\{q_{mn'} : n' \in \mathcal{N}(m)\backslash n\}$. The algorithm would produce the exact posterior probabilities of all the bits after a fixed number of iterations if the bipartite graph defined by the matrix $\mathbf{H}$ contained no cycles.

Initialization. Let $p^0_n = P(x_n = 0)$ (the prior probability that bit x_n is 0), and let $p^1_n = P(x_n = 1) = 1 - p^0_n$. If we are taking the syndrome decoding viewpoint and the channel is a binary symmetric channel then p^1_n will equal f. If the noise level varies in a known way (for example if the channel is a binary-input Gaussian channel with a real output) then p^1_n is initialized to the appropriate normalized likelihood. For every (n, m) such that $H_{mn} = 1$ the variables q^0_{mn} and q^1_{mn} are initialized to the values p^0_n and p^1_n respectively.

Horizontal step. In the *horizontal* step of the algorithm (horizontal from the point of view of the matrix $\mathbf{H}$), we run through the checks m and compute for each $n \in \mathcal{N}(m)$ two probabilities: first, r^0_{mn}, the probability of the observed value of z_m arising when $x_n = 0$, given that the other bits $\{x_{n'} : n' \neq n\}$ have a separable distribution given by the probabilities $\{q^0_{mn'}, q^1_{mn'}\}$, defined by:

$$r^0_{mn} = \sum_{\{x_{n'}:\, n' \in \mathcal{N}(m)\backslash n\}} P\left(z_m \,|\, x_n = 0,\, \{x_{n'} :\, n' \in \mathcal{N}(m)\backslash n\}\right) \prod_{n' \in \mathcal{N}(m)\backslash n} q^{x_{n'}}_{mn'} \tag{47.7}$$

and second, r^1_{mn}, the probability of the observed value of z_m arising when $x_n = 1$, defined by:

$$r^1_{mn} = \sum_{\{x_{n'}:\, n' \in \mathcal{N}(m)\backslash n\}} P\left(z_m \,|\, x_n = 1,\, \{x_{n'} :\, n' \in \mathcal{N}(m)\backslash n\}\right) \prod_{n' \in \mathcal{N}(m)\backslash n} q^{x_{n'}}_{mn'} . \tag{47.8}$$

The conditional probabilities in these summations are either zero or one, depending on whether the observed z_m matches the hypothesized values for x_n and the $\{x_{n'}\}$.

These probabilities can be computed in various obvious ways based on equation (47.7) and (47.8). The computations may be done most efficiently (if $|\mathcal{N}(m)|$ is large) by regarding $z_m + x_n$ as the final state of a Markov chain with states 0 and 1, this chain being started in state 0, and undergoing transitions corresponding to additions of the various $x_{n'}$, with transition probabilities given by the corresponding $q^0_{mn'}$ and $q^1_{mn'}$. The probabilities for z_m having its observed value given either $x_n = 0$ or $x_n = 1$ can then be found efficiently by use of the forward–backward algorithm (section 25.3).

A particularly convenient implementation of this method uses forward and backward passes in which products of the differences $\delta q_{mn} \equiv q^0_{mn} - q^1_{mn}$ are computed. We obtain $\delta r_{mn} \equiv r^0_{mn} - r^1_{mn}$ from the identity:

$$\delta r_{mn} = (-1)^{z_m} \prod_{n' \in \mathcal{N}(m)\backslash n} \delta q_{mn'}. \tag{47.9}$$

This identity is derived by iterating the following observation: if $\zeta = x_\mu + x_\nu \bmod 2$, and x_μ and x_ν have probabilities q^0_μ, q^0_ν and q^1_μ, q^1_ν of being 0 and 1, then $P(\zeta = 1) = q^1_\mu q^0_\nu + q^0_\mu q^1_\nu$ and $P(\zeta = 0) = q^0_\mu q^0_\nu + q^1_\mu q^1_\nu$. Thus $P(\zeta = 0) - P(\zeta = 1) = (q^0_\mu - q^1_\mu)(q^0_\nu - q^1_\nu)$.

We recover r^0_{mn} and r^1_{mn} using

$$r^0_{mn} = 1/2(1 + \delta r_{mn}), \quad r^1_{mn} = 1/2(1 - \delta r_{mn}). \tag{47.10}$$

The transformations into differences δq and back from δr to $\{r\}$ may be viewed as a Fourier transform and an inverse Fourier transformation.

Vertical step. The *vertical* step takes the computed values of r^0_{mn} and r^1_{mn} and updates the values of the probabilities q^0_{mn} and q^1_{mn}. For each n we compute:

$$q^0_{mn} = \alpha_{mn}\, p^0_n \prod_{m' \in \mathcal{M}(n)\backslash m} r^0_{m'n} \tag{47.11}$$

$$q^1_{mn} = \alpha_{mn}\, p^1_n \prod_{m' \in \mathcal{M}(n)\backslash m} r^1_{m'n} \tag{47.12}$$

where α_{mn} is chosen such that $q^0_{mn} + q^1_{mn} = 1$. These products can be efficiently computed in a downward pass and an upward pass.

We can also compute the 'pseudoposterior probabilities' q^0_n and q^1_n at this iteration, given by:

$$q^0_n = \alpha_n\, p^0_n \prod_{m \in \mathcal{M}(n)} r^0_{mn}, \tag{47.13}$$

$$q^1_n = \alpha_n\, p^1_n \prod_{m \in \mathcal{M}(n)} r^1_{mn}. \tag{47.14}$$

These quantities are used to create a tentative decoding $\hat{\mathbf{x}}$, the consistency of which is used to decide whether the decoding algorithm can halt. (Halt if $\mathbf{H}\hat{\mathbf{x}} = \mathbf{z}$.)

At this point, the algorithm repeats from the horizontal step.

The stop-when-it's-done decoding method. The recommended decoding procedure is to set $\hat{x}_n$ to 1 if $q^1_n > 0.5$ and see if the checks $\mathbf{H}\hat{\mathbf{x}} = \mathbf{z} \bmod 2$ are all satisfied, halting when they are, and declaring a failure if some maximum number of iterations (e.g. 200 or 1000) occurs without successful decoding. In the event of a failure, we may still report $\hat{\mathbf{x}}$, but we flag the whole block as a failure.

We note in passing the difference between this decoding procedure and the widespread practice in the turbo code community, where the decoding algorithm is run for a *fixed* number of iterations (irrespective of whether the decoder finds a consistent state at some earlier time). This practice is wasteful of computer time, and it blurs the distinction between undetected and detected errors. In our procedure, 'undetected' errors occur if the decoder finds an $\hat{\mathbf{x}}$

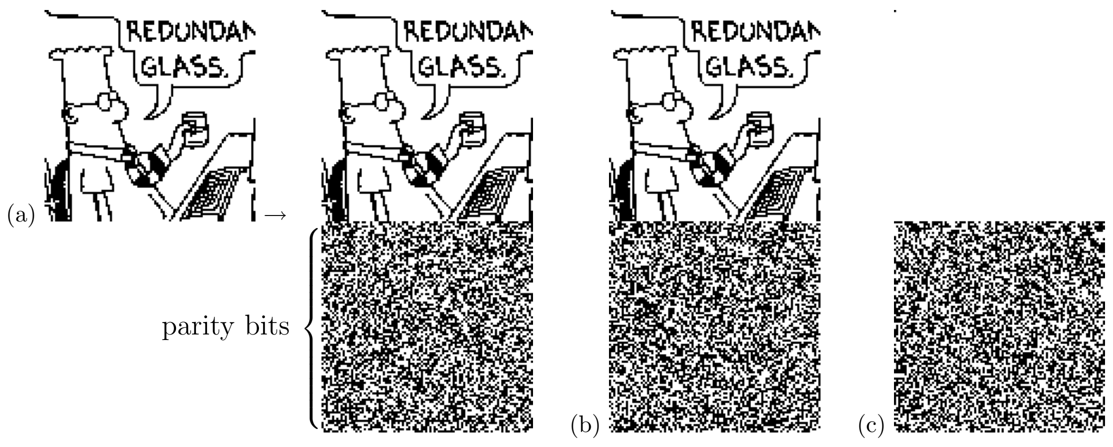


Figure 47.3. Demonstration of encoding with a rate-1/2 Gallager code. The encoder is derived from a very sparse $10\,000 \times 20\,000$ parity-check matrix with three 1s per column (figure 47.4). (a) The code creates transmitted vectors consisting of 10 000 source bits and 10 000 parity-check bits. (b) Here, the source sequence has been altered by changing the first bit. Notice that many of the parity-check bits are changed. Each parity bit depends on about half of the source bits. (c) The transmission for the case $\mathbf{s} = (1, 0, 0, \ldots, 0)$. This vector is the difference (modulo 2) between transmissions (a) and (b). [Dilbert image Copyright©1997 United Feature Syndicate, Inc., used with permission.]

satisfying $\mathbf{H}\hat{\mathbf{x}} = \mathbf{z} \bmod 2$ that is not equal to the true $\mathbf{x}$. 'Detected' errors occur if the algorithm runs for the maximum number of iterations without finding a valid decoding. Undetected errors are of scientific interest because they reveal distance properties of a code. And in engineering practice, it would seem preferable for the blocks that are known to contain detected errors to be so labelled if practically possible.

Cost. In a brute-force approach, the time to create the generator matrix scales as N^3, where N is the block size. The encoding time scales as N^2, but encoding involves only binary arithmetic, so for the block lengths studied here it takes considerably less time than the simulation of the Gaussian channel. Decoding involves approximately $6Nj$ floating-point multiplies per iteration, so the total number of operations per decoded bit (assuming 20 iterations) is about $120t/R$, independent of blocklength. For the codes presented in the next section, this is about 800 operations.

The encoding complexity can be reduced by clever encoding tricks invented by Richardson and Urbanke (2001b) or by specially constructing the parity-check matrix (MacKay *et al.*, 1999).

The decoding complexity can be reduced, with only a small loss in performance, by passing low-precision messages in place of real numbers (Richardson and Urbanke, 2001a).

▶ 47.4 Pictorial demonstration of Gallager codes

Figures 47.3–47.7 illustrate visually the conditions under which low-density parity-check codes can give reliable communication over binary symmetric channels and Gaussian channels. These demonstrations may be viewed as animations on the world wide web.[1]

[1]`http://www.inference.phy.cam.ac.uk/mackay/codes/gifs/`

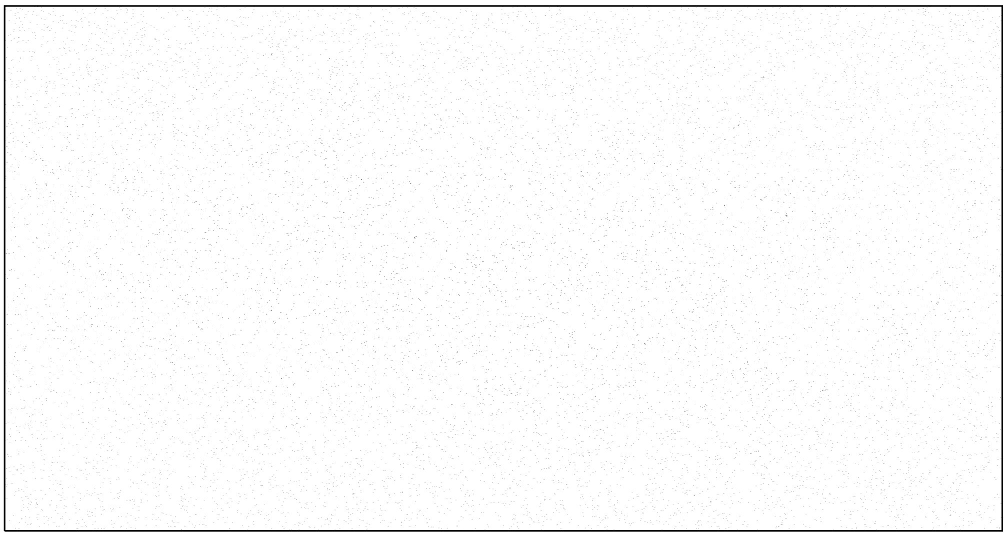
$\mathbf{H} =$ 

Figure 47.4. A low-density parity-check matrix with $N = 20\,000$ columns of weight $j = 3$ and $M = 10\,000$ rows of weight $k = 6$.

Encoding

Figure 47.3 illustrates the encoding operation for the case of a Gallager code whose parity-check matrix is a $10\,000 \times 20\,000$ matrix with three 1s per column (figure 47.4). The high density of the *generator* matrix is illustrated in figure 47.3b and c by showing the change in the transmitted vector when one of the 10 000 source bits is altered. Of course, the source images shown here are highly redundant, and such images should really be compressed before encoding. Redundant images are chosen in these demonstrations to make it easier to see the correction process during the iterative decoding. The decoding algorithm does *not* take advantage of the redundancy of the source vector, and it would work in exactly the same way irrespective of the choice of source vector.

Iterative decoding

The transmission is sent over a channel with noise level $f = 7.5\%$ and the received vector is shown in the upper left of figure 47.5. The subsequent pictures in figure 47.5 show the iterative probabilistic decoding process. The sequence of figures shows the best guess, bit by bit, given by the iterative decoder, after 0, 1, 2, 3, 10, 11, 12, and 13 iterations. The decoder halts after the 13th iteration when the best guess violates no parity checks. This final decoding is error free.

In the case of an unusually noisy transmission, the decoding algorithm fails to find a valid decoding. For this code and a channel with $f = 7.5\%$, such failures happen about once in every 100 000 transmissions. Figure 47.6 shows this error rate compared with the block error rates of classical error-correcting codes.

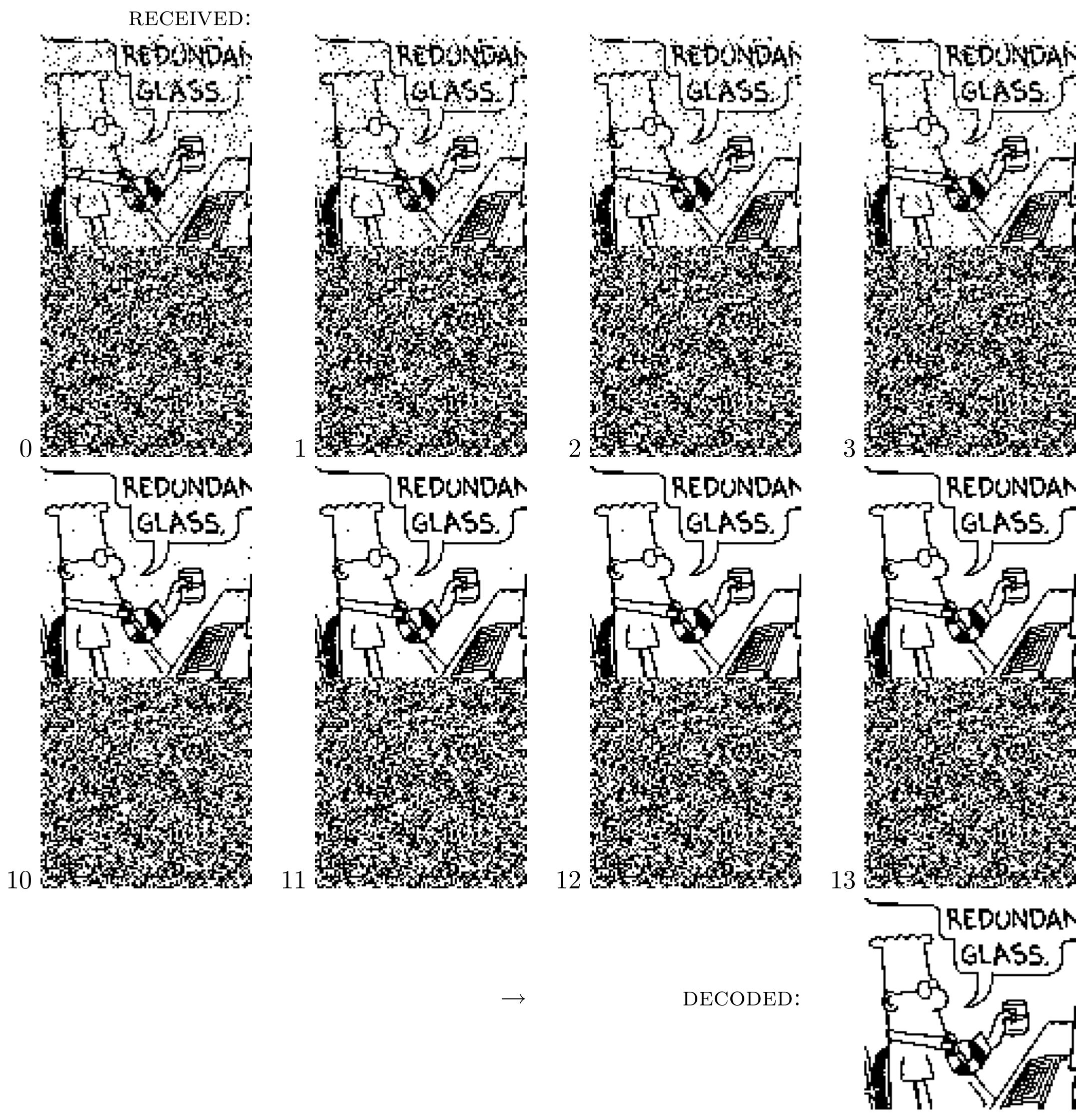


Figure 47.5. Iterative probabilistic decoding of a low-density parity-check code for a transmission received over a channel with noise level $f = 7.5\%$. The sequence of figures shows the best guess, bit by bit, given by the iterative decoder, after 0, 1, 2, 3, 10, 11, 12, and 13 iterations. The decoder halts after the 13th iteration when the best guess violates no parity checks. This final decoding is error free.

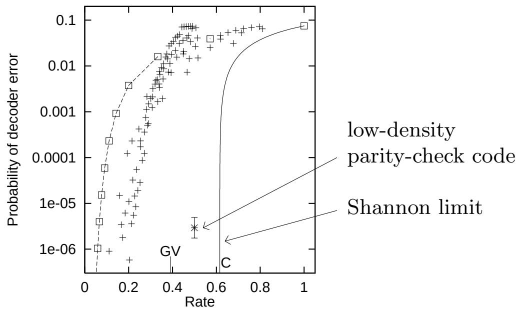


Figure 47.6. Error probability of the low-density parity-check code (with error bars) for binary symmetric channel with $f = 7.5\%$, compared with algebraic codes. Squares: repetition codes and Hamming (7, 4) code; other points: Reed–Muller and BCH codes.

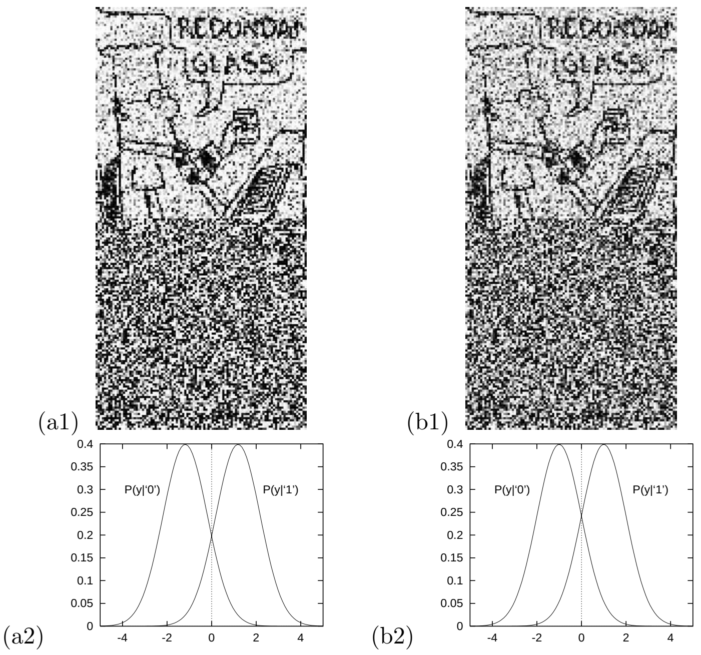


Figure 47.7. Demonstration of a Gallager code for a Gaussian channel. (a1) The received vector after transmission over a Gaussian channel with $x/\sigma = 1.185$ ($E_b/N_0 = 1.47\,\text{dB}$). The greyscale represents the value of the normalized likelihood. This transmission can be perfectly decoded by the sum–product decoder. The empirical probability of decoding failure is about 10^{-5}. (a2) The probability distribution of the output y of the channel with $x/\sigma = 1.185$ for each of the two possible inputs. (b1) The received transmission over a Gaussian channel with $x/\sigma = 1.0$, which corresponds to the Shannon limit. (b2) The probability distribution of the output y of the channel with $x/\sigma = 1.0$ for each of the two possible inputs.

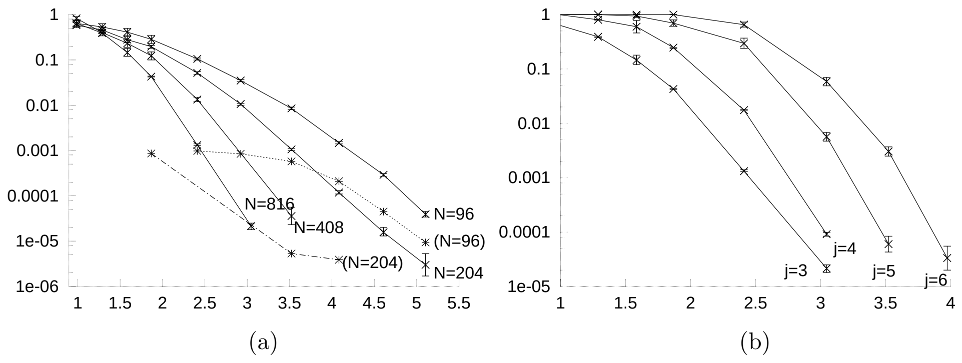


Figure 47.8. Performance of rate-1/2 Gallager codes on the Gaussian channel. Vertical axis: block error probability. Horizontal axis: signal-to-noise ratio E_b/N_0. (a) Dependence on blocklength N for $(j,k) = (3,6)$ codes. From left to right: $N = 816$, $N = 408$, $N = 204$, $N = 96$. The dashed lines show the frequency of undetected errors, which is measurable only when the blocklength is as small as $N = 96$ or $N = 204$. (b) Dependence on column weight j for codes of blocklength $N = 816$.

Gaussian channel

In figure 47.7 the left picture shows the received vector after transmission over a Gaussian channel with $x/\sigma = 1.185$. The greyscale represents the value of the normalized likelihood, $\frac{P(y\,|\,t=1)}{P(y\,|\,t=1)+P(y\,|\,t=0)}$. This signal-to-noise ratio $x/\sigma = 1.185$ is a noise level at which this rate-1/2 Gallager code communicates reliably (the probability of error is $\simeq 10^{-5}$). To show how close we are to the Shannon limit, the right panel shows the received vector when the signal-to-noise ratio is reduced to $x/\sigma = 1.0$, which corresponds to the Shannon limit for codes of rate 1/2.

Variation of performance with code parameters

Figure 47.8 shows how the parameters N and j affect the performance of low-density parity-check codes. As Shannon would predict, increasing the blocklength leads to improved performance. The dependence on j follows a different pattern. Given an *optimal* decoder, the best performance would be obtained for the codes closest to random codes, that is, the codes with largest j. However, the sum–product decoder makes poor progress in dense graphs, so the best performance is obtained for a small value of j. Among the values

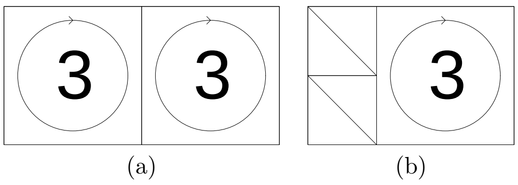


Figure 47.9. Schematic illustration of constructions (a) of a completely regular Gallager code with $j = 3$, $k = 6$ and $R = 1/2$; (b) of a nearly-regular Gallager code with rate 1/3. Notation: an integer represents a number of permutation matrices superposed on the surrounding square. A diagonal line represents an identity matrix.

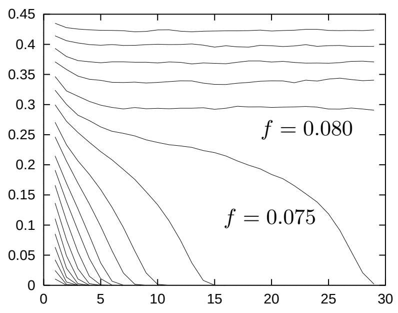


Figure 47.10. Monte Carlo simulation of density evolution, following the decoding process for $j = 4, k = 8$. Each curve shows the average entropy of a bit as a function of number of iterations, as estimated by a Monte Carlo algorithm using 10 000 samples per iteration. The noise level of the binary symmetric channel f increases by steps of 0.005 from bottom graph ($f = 0.010$) to top graph ($f = 0.100$). There is evidently a threshold at about $f = 0.075$, above which the algorithm cannot determine $\mathbf{x}$. From MacKay (1999b).

of j shown in the figure, $j = 3$ is the best, for a blocklength of 816, down to a block error probability of 10^{-5}.

This observation motivates construction of Gallager codes with some columns of weight 2. A construction with $M/2$ columns of weight 2 is shown in figure 47.9b. Too many columns of weight 2, and the code becomes a much poorer code.

As we'll discuss later, we can do even better by making the code even more irregular.

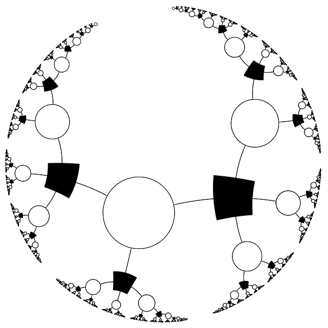


Figure 47.11. Local topology of the graph of a Gallager code with column weight $j = 3$ and row weight $k = 4$. White nodes represent bits, x_l; black nodes represent checks, z_m; each edge corresponds to a 1 in $\mathbf{H}$.

▶ 47.5 Density evolution

One way to study the decoding algorithm is to imagine it running on an infinite tree-like graph with the same local topology as the Gallager code's graph. The larger the matrix $\mathbf{H}$, the closer its decoding properties should approach those of the infinite graph.

Imagine an infinite belief network with no loops, in which every bit x_n connects to j checks and every check z_m connects to k bits (figure 47.11). We consider the iterative flow of information in this network, and examine the average entropy of one bit as a function of number of iterations. At each iteration, a bit has accumulated information from its local network out to a radius equal to the number of iterations. Successful decoding will occur only if the average entropy of a bit decreases to zero as the number of iterations increases.

The iterations of an infinite belief network can be simulated by Monte Carlo methods – a technique first used by Gallager (1963). Imagine a network of radius I (the total number of iterations) centred on one bit. Our aim is to compute the conditional entropy of the central bit x given the state $\mathbf{z}$ of all checks out to radius I. To evaluate the probability that the central bit is 1 given a *particular* syndrome $\mathbf{z}$ involves an I-step propagation from the outside of the network into the centre. At the ith iteration, probabilities r at radius $I - i + 1$ are transformed into qs and then into rs at radius $I - i$ in a way that depends on the states x of the unknown bits at radius $I - i$. In the Monte Carlo method, rather than simulating this network exactly, which would take a time that grows exponentially with I, we create for each iteration a representative sample (of size 100, say) of the values of $\{r, x\}$. In the case of a regular network with parameters j, k, each new pair $\{r, x\}$ in the list at the ith iteration is created by drawing the new x from its distribution and drawing at random with replacement $(j-1)(k-1)$ pairs $\{r, x\}$ from the list at the $(i-1)$th iteration; these are assembled into a tree fragment (figure 47.12) and the sum–product algorithm is run from top to bottom to find the new r value associated with the new node.

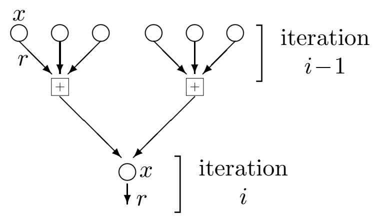


Figure 47.12. A tree-fragment constructed during Monte Carlo simulation of density evolution. This fragment is appropriate for a regular $j=3$, $k=4$ Gallager code.

As an example, the results of runs with $j=4$, $k=8$ and noise densities f between 0.01 and 0.10, using 10 000 samples at each iteration, are shown in figure 47.10. Runs with low enough noise level show a collapse to zero entropy after a small number of iterations, and those with high noise level decrease to a non-zero entropy corresponding to a failure to decode.

The boundary between these two behaviours is called the *threshold* of the decoding algorithm for the binary symmetric channel. Figure 47.10 shows by Monte Carlo simulation that the threshold for regular $(j, k) = (4, 8)$ codes is about 0.075. Richardson and Urbanke (2001a) have derived thresholds for regular codes by a tour de force of direct analytic methods. Some of these thresholds are shown in table 47.13.

(j,k)	$f_{\max}$
(3,6)	0.084
(4,8)	0.076
(5,10)	0.068

Table 47.13. Thresholds $f_{\max}$ for regular low-density parity-check codes, assuming sum–product decoding algorithm, from Richardson and Urbanke (2001a). The Shannon limit for rate-1/2 codes is $f_{\max} = 0.11$.

Approximate density evolution

For practical purposes, the computational cost of density evolution can be reduced by making Gaussian approximations to the probability distributions over the messages in density evolution, and updating only the parameters of these approximations. For further information about these techniques, which produce diagrams known as *EXIT charts*, see (ten Brink, 1999; Chung *et al.*, 2001; ten Brink *et al.*, 2002).

47.6 Improving Gallager codes

Since the rediscovery of Gallager codes, two methods have been found for enhancing their performance.

$GF(4)$	$\leftrightarrow$	binary
0	$\leftrightarrow$	00
1	$\leftrightarrow$	01
A	$\leftrightarrow$	10
B	$\leftrightarrow$	11

Table 47.14. Translation between $GF(4)$ and binary for message symbols.

Clump bits and checks together

First, we can make Gallager codes in which the variable nodes are grouped together into metavariables consisting of say 3 binary variables, and the check nodes are similarly grouped together into metachecks. As before, a sparse graph can be constructed connecting metavariables to metachecks, with a lot of freedom about the details of how the variables and checks within are wired up. One way to set the wiring is to work in a finite field $GF(q)$ such as $GF(4)$ or $GF(8)$, define low-density parity-check matrices using elements of $GF(q)$, and translate our binary messages into $GF(q)$ using a mapping such as the one for $GF(4)$ given in table 47.14. Now, when messages are passed during decoding, those messages are probabilities and likelihoods over *conjunctions* of binary variables. For example if each clump contains three binary variables then the likelihoods will describe the likelihoods of the eight alternative states of those bits.

$GF(4)$	$\rightarrow$	binary
0	$\rightarrow$	$\begin{smallmatrix}00\\00\end{smallmatrix}$
1	$\rightarrow$	$\begin{smallmatrix}10\\01\end{smallmatrix}$
A	$\rightarrow$	$\begin{smallmatrix}11\\10\end{smallmatrix}$
B	$\rightarrow$	$\begin{smallmatrix}01\\11\end{smallmatrix}$

Table 47.15. Translation between $GF(4)$ and binary for matrix entries. An $M \times N$ parity-check matrix over $GF(4)$ can be turned into a $2M \times 2N$ binary parity-check matrix in this way.

With carefully optimized constructions, the resulting codes over $GF(4)$,

$$\begin{aligned}
F^0 &= [f^0 + f^1] + [f^A + f^B] \\
F^1 &= [f^0 - f^1] + [f^A - f^B] \\
F^A &= [f^0 + f^1] - [f^A + f^B] \\
F^B &= [f^0 - f^1] - [f^A - f^B]
\end{aligned}$$

Algorithm 47.16. The Fourier transform over $GF(4)$.
The Fourier transform F of a function f over $GF(2)$ is given by $F^0 = f^0 + f^1$, $F^1 = f^0 - f^1$. Transforms over $GF(2^k)$ can be viewed as a sequence of binary transforms in each of k dimensions. The inverse transform is identical to the Fourier transform, except that we also divide by 2^k.

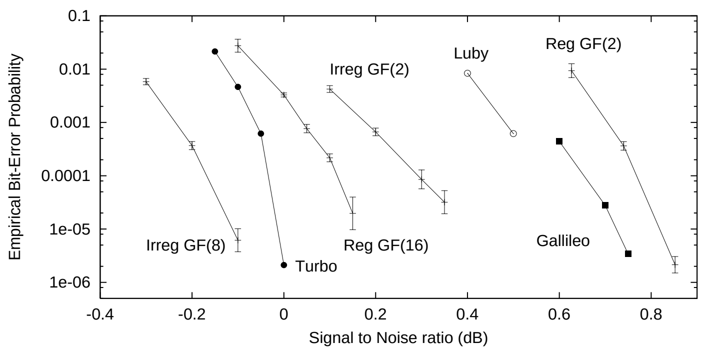


Figure 47.17. Comparison of regular binary Gallager codes with irregular codes, codes over $GF(q)$, and other outstanding codes of rate $1/4$. From left (best performance) to right: Irregular low-density parity-check code over $GF(8)$, blocklength 48 000 bits (Davey, 1999); JPL turbo code (JPL, 1996) blocklength 65 536; Regular low-density parity-check over $GF(16)$, blocklength 24 448 bits (Davey and MacKay, 1998); Irregular binary low-density parity-check code, blocklength 16 000 bits (Davey, 1999); Luby *et al.* (1998) irregular binary low-density parity-check code, blocklength 64 000 bits; JPL code for Galileo (in 1992, this was the best known code of rate 1/4); Regular binary low-density parity-check code: blocklength 40 000 bits (MacKay, 1999b). The Shannon limit is at about -0.79 dB. As of 2003, even better sparse-graph codes have been constructed.

$GF(8)$, and $GF(16)$ perform nearly one decibel better than comparable binary Gallager codes.

The computational cost for decoding in $GF(q)$ scales as $q \log q$, if the appropriate Fourier transform is used in the check nodes: the update rule for the check-to-variable message,

$$r^a_{mn} = \sum_{\mathbf{x}:x_n=a} \mathbb{1}\left[\sum_{n' \in \mathcal{N}(m)} H_{mn'} x_{n'} = z_m\right] \prod_{j \in \mathcal{N}(m)\backslash n} q^{x_j}_{mj}, \tag{47.15}$$

is a convolution of the quantities q^a_{mj}, so the summation can be replaced by a product of the Fourier transforms of q^a_{mj} for $j \in \mathcal{N}(m)\backslash n$, followed by an inverse Fourier transform. The Fourier transform for $GF(4)$ is shown in algorithm 47.16.

Make the graph irregular

The second way of improving Gallager codes, introduced by Luby *et al.* (2001b), is to make their graphs *irregular*. Instead of giving all variable nodes the same degree j, we can have some variable nodes with degree 2, some 3, some 4, and a few with degree 20. Check nodes can also be given unequal degrees – this helps improve performance on erasure channels, but it turns out that for the Gaussian channel, the best graphs have regular check degrees.

Figure 47.17 illustrates the benefits offered by these two methods for improving Gallager codes, focussing on codes of rate $1/4$. Making the binary code irregular gives a win of about 0.4 dB; switching from $GF(2)$ to $GF(16)$ gives

DIFFERENCE SET CYCLIC CODES						
N	7	21	73	273	1057	4161
M	4	**10**	**28**	**82**	**244**	**730**
K	3	11	45	191	813	3431
d	4	**6**	**10**	**18**	34	66
k	3	**5**	**9**	17	33	65

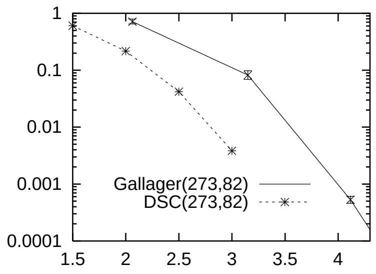


Figure 47.18. An algebraically constructed low-density parity-check code satisfying many redundant constraints outperforms an equivalent random Gallager code. The table shows the N, M, K, distance d, and row weight k of some difference-set cyclic codes, highlighting the codes that have large d/N, small k, and large N/M. In the comparison the Gallager code had $(j, k) = (4, 13)$, and rate identical to the $N = 273$ difference-set cyclic code. Vertical axis: block error probability. Horizontal axis: signal-to-noise ratio E_b/N_0 (dB).

about 0.6 dB; and Matthew Davey's code that combines both these features – it's irregular over $GF(8)$ – gives a win of about 0.9 dB over the regular binary Gallager code.

Methods for optimizing the *profile* of a Gallager code (that is, its number of rows and columns of each degree), have been developed by Richardson *et al.* (2001) and have led to low-density parity-check codes whose performance, when decoded by the sum–product algorithm, is within a hair's breadth of the Shannon limit.

Algebraic constructions of Gallager codes

The performance of regular Gallager codes can be enhanced in a third manner: by designing the code to have *redundant sparse constraints*. There is a *difference-set cyclic code*, for example, that has $N = 273$ and $K = 191$, but the code satisfies not $M = 82$ but N, i.e., *273* low-weight constraints (figure 47.18). It is impossible to make random Gallager codes that have anywhere near this much redundancy among their checks. The difference-set cyclic code performs about 0.7 dB better than an equivalent random Gallager code.

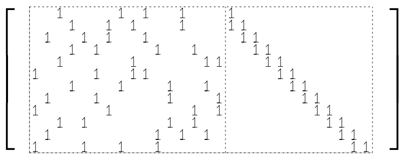
An open problem is to discover codes sharing the remarkable properties of the difference-set cyclic codes but with different blocklengths and rates. I call this task *the Tanner challenge*.

▶ 47.7 Fast encoding of low-density parity-check codes

We now discuss methods for fast encoding of low-density parity-check codes – faster than the standard method, in which a generator matrix $\mathbf{G}$ is found by Gaussian elimination (at a cost of order M^3) and then each block is encoded by multiplying it by $\mathbf{G}$ (at a cost of order M^2).

Staircase codes

Certain low-density parity-check matrices with M columns of weight 2 or less can be encoded easily in linear time. For example, if the matrix has a *staircase* structure as illustrated by the right-hand side of

$\mathbf{H} =$

, (47.16)

and if the data **s** are loaded into the first K bits, then the M parity bits **p** can be computed from left to right in linear time.

$$\begin{aligned} p_1 &= & & \textstyle\sum_{n=1}^{K} H_{1n}s_n \\ p_2 &= p_1 &+& \textstyle\sum_{n=1}^{K} H_{2n}s_n \\ p_3 &= p_2 &+& \textstyle\sum_{n=1}^{K} H_{3n}s_n \\ &\vdots \\ p_M &= p_{M-1} &+& \textstyle\sum_{n=1}^{K} H_{Mn}s_n . \end{aligned} \tag{47.17}$$

If we call two parts of the **H** matrix $[\mathbf{H}_s|\mathbf{H}_p]$, we can describe the encoding operation in two steps: first compute an intermediate parity vector $\mathbf{v} = \mathbf{H}_s\mathbf{s}$; then pass **v** through an accumulator to create **p**.

The cost of this encoding method is linear if the sparsity of **H** is exploited when computing the sums in (47.17).

Fast encoding of general low-density parity-check codes

Richardson and Urbanke (2001b) demonstrated an elegant method by which the encoding cost of any low-density parity-check code can be reduced from the straightforward method's M^2 to a cost of $N + g^2$, where g, the *gap*, is hopefully a small constant, and in the worst cases scales as a small fraction of N.

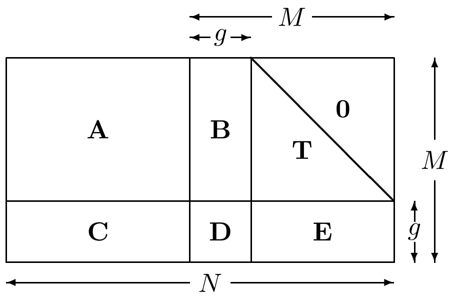


Figure 47.19. The parity-check matrix in approximate lower-triangular form.

In the first step, the parity-check matrix is rearranged, by row-interchange and column-interchange, into the *approximate lower-triangular form* shown in figure 47.19. The original matrix **H** was very sparse, so the six matrices **A**, **B**, **T**, **C**, **D**, and **E** are also very sparse. The matrix **T** is lower triangular and has 1s everywhere on the diagonal.

$$\mathbf{H} = \begin{bmatrix} \mathbf{A} & \mathbf{B} & \mathbf{T} \\ \mathbf{C} & \mathbf{D} & \mathbf{E} \end{bmatrix}. \tag{47.18}$$

The source vector **s** of length $K = N - M$ is encoded into a transmission $\mathbf{t} = [\mathbf{s}, \mathbf{p}_1, \mathbf{p}_2]$ as follows.

1. Compute the upper syndrome of the source vector,

$$\mathbf{z}_A = \mathbf{A}\mathbf{s}. \tag{47.19}$$

   This can be done in linear time.

2. Find a setting of the second parity bits, $\mathbf{p}_2^A$, such that the upper syndrome is zero.

$$\mathbf{p}_2^A = -\mathbf{T}^{-1}\mathbf{z}_A. \tag{47.20}$$

   This vector can be found in linear time by back-substitution, i.e., computing the first bit of $\mathbf{p}_2^A$, then the second, then the third, and so forth.

3. Compute the lower syndrome of the vector $[\mathbf{s}, \mathbf{0}, \mathbf{p}_2^A]$:

$$\mathbf{z}_B = \mathbf{C}\mathbf{s} - \mathbf{E}\mathbf{p}_2^A. \tag{47.21}$$

   This can be done in linear time.

4. Now we get to the clever bit. Define the matrix

$$\mathbf{F} \equiv -\mathbf{E}\mathbf{T}^{-1}\mathbf{B} + \mathbf{D}, \tag{47.22}$$

   and find its inverse, $\mathbf{F}^{-1}$. This computation needs to be done once only, and its cost is of order g^3. This inverse $\mathbf{F}^{-1}$ is a dense $g \times g$ matrix. [If $\mathbf{F}$ is not invertible then either $\mathbf{H}$ is not of full rank, or else further column permutations of $\mathbf{H}$ can produce an $\mathbf{F}$ that is invertible.]

   Set the first parity bits, $\mathbf{p}_1$, to

$$\mathbf{p}_1 = -\mathbf{F}^{-1}\mathbf{z}_B. \tag{47.23}$$

   This operation has a cost of order g^2.

   Claim: At this point, we have found the correct setting of the first parity bits, $\mathbf{p}_1$.

5. Discard the tentative parity bits $\mathbf{p}_2^A$ and find the new upper syndrome,

$$\mathbf{z}_C = \mathbf{z}_A + \mathbf{B}\mathbf{p}_1. \tag{47.24}$$

   This can be done in linear time.

6. Find a setting of the second parity bits, $\mathbf{p}_2$, such that the upper syndrome is zero,

$$\mathbf{p}_2 = -\mathbf{T}^{-1}\mathbf{z}_C \tag{47.25}$$

   This vector can be found in linear time by back-substitution.

▶ 47.8 Further reading

Low-density parity-check codes codes were first studied in 1962 by Gallager, then were generally forgotten by the coding theory community. Tanner (1981) generalized Gallager's work by introducing more general constraint nodes; the codes that are now called turbo product codes should in fact be called Tanner product codes, since Tanner proposed them, and his colleagues (Karplus and Krit, 1991) implemented them in hardware. Publications on Gallager codes contributing to their 1990s rebirth include (Wiberg *et al.*, 1995; MacKay and Neal, 1995; MacKay and Neal, 1996; Wiberg, 1996; MacKay, 1999b; Spielman, 1996; Sipser and Spielman, 1996). Low-precision decoding algorithms and fast encoding algorithms for Gallager codes are discussed in (Richardson and Urbanke, 2001a; Richardson and Urbanke, 2001b). MacKay and Davey (2000) showed that low-density parity-check codes can outperform Reed–Solomon codes, even on the Reed–Solomon codes' home turf: high rate and short block-lengths. Other important papers include (Luby *et al.*, 2001a; Luby *et al.*, 2001b; Luby *et al.*, 1997; Davey and MacKay, 1998; Richardson *et al.*, 2001; Chung *et al.*, 2001). Useful tools for the design of irregular low-density parity-check codes include (Chung *et al.*, 1999; Urbanke, 2001).

See (Wiberg, 1996; Frey, 1998; McEliece *et al.*, 1998) for further discussion of the sum–product algorithm.

For a view of low-density parity-check code decoding in terms of group theory and coding theory, see (Forney, 2001; Offer and Soljanin, 2000; Offer and Soljanin, 2001); and for background reading on this topic see (Hartmann and Rudolph, 1976; Terras, 1999). There is a growing literature on the practical design of low-density parity-check codes (Mao and Banihashemi, 2000; Mao and Banihashemi, 2001; ten Brink *et al.*, 2002); they are now being adopted for applications from hard drives to satellite communications.

For low-density parity-check codes applicable to quantum error-correction, see MacKay *et al.* (2004).

▶ 47.9 Exercises

Exercise 47.1.[2] The 'hyperbolic tangent' version of the decoding algorithm. In section 47.3, the sum–product decoding algorithm for low-density parity-check codes was presented first in terms of quantities $q_{mn}^{0/1}$ and $r_{mn}^{0/1}$, then in terms of quantities δq and δr. There is a third description, in which the $\{q\}$ are replaced by log probability-ratios,

$$l_{mn} \equiv \ln \frac{q_{mn}^0}{q_{mn}^1}. \tag{47.26}$$

Show that

$$\delta q_{mn} \equiv q_{mn}^0 - q_{mn}^1 = \tanh(l_{mn}/2). \tag{47.27}$$

Derive the update rules for $\{r\}$ and $\{l\}$.

Exercise 47.2.[2, p.572] I am sometimes asked 'why not decode *other* linear codes, for example algebraic codes, by transforming their parity-check matrices so that they are low-density, and applying the sum–product algorithm?' [Recall that any linear combination of rows of $\mathbf{H}$, $\mathbf{H}' = \mathbf{P}\mathbf{H}$, is a valid parity-check matrix for a code, as long as the matrix $\mathbf{P}$ is invertible; so there are many parity check matrices for any one code.]

Explain why a random linear code does not have a low-density parity-check matrix. [Here, low-density means 'having row-weight at most k', where k is some small constant $\ll N$.]

Exercise 47.3.[3] Show that if a low-density parity-check code has more than M columns of weight 2 – say αM columns, where $\alpha > 1$ – then the code will have words with weight of order $\log M$.

Exercise 47.4.[5] In section 13.5 we found the expected value of the weight enumerator function $A(w)$, averaging over the ensemble of all random linear codes. This calculation can also be carried out for the ensemble of low-density parity-check codes (Gallager, 1963; MacKay, 1999b; Litsyn and Shevelev, 2002). It is plausible, however, that the mean value of $A(w)$ is not always a good indicator of the *typical* value of $A(w)$ in the ensemble. For example, if, at a particular value of w, 99% of codes have $A(w) = 0$, and 1% have $A(w) = 100\,000$, then while we might say the typical value of $A(w)$ is zero, the mean is found to be 1000. Find the *typical* weight enumerator function of low-density parity-check codes.

▶ 47.10 Solutions

Solution to exercise 47.2 (p.572). Consider codes of rate R and blocklength N, having $K = RN$ source bits and $M = (1-R)N$ parity-check bits. Let all the codes have their bits ordered so that the first K bits are independent, so that we could if we wish put the code in systematic form,

$$\mathbf{G} = [\mathbf{1}_K|\mathbf{P}^{\mathsf{T}}]; \;\; \mathbf{H} = [\mathbf{P}|\mathbf{1}_M]. \tag{47.28}$$

The number of *distinct* linear codes is the number of matrices $\mathbf{P}$, which is $\mathcal{N}_1 = 2^{MK} = 2^{N^2R(1-R)}$. Can these all be expressed as distinct low-density parity-check codes?

$\log \mathcal{N}_1 \simeq N^2 R(1-R)$

The number of low-density parity-check matrices with row-weight k is

$$\binom{N}{k}^M \tag{47.29}$$

and the number of distinct codes that they define is at most

$$\mathcal{N}_2 = \binom{N}{k}^M \Big/ M!, \tag{47.30}$$

which is much smaller than $\mathcal{N}_1$, so, by the pigeon-hole principle, it is not possible for every random linear code to map on to a low-density $\mathbf{H}$.

$\log \mathcal{N}_2 < Nk \log N$