Recall that an error correcting code is a procedure for introducing redundancy to a message we want to send to someone so that even if parts of the encoded message are corrupted, the message can be uniquely recovered. In the last lecture you saw various ways to design error correcting codes, but let's see how those would play out if you were to use them in practice.

Suppose you designed some code and you use it to send a message to your friend. During transmission, the message gets corrupted in some unknown manner. How should your friend go about fixing the errors?

Let's do an example. Recall the Reed-Solomon code from this morning's lecture. Say Alice has a five symbol message written in English, for example `speak`, that she wants to send to Bob. Alice will encode this message like this. Think of each of the letters in the message as coming from an alphabet of size 29 (English has 26 letters, but we can also include symbols like the white space, a comma, a dot), so we can identify each letter with a number in the set $\{0, \ldots, 28\}$. If we identify the letters by their position in the alphabet, for instance, the message `speak` would be represented as $(18, 15, 4, 0, 10)$. To encode this message in order to protect from errors, we need to introduce some redundancies – say we take our 5-symbol message and convert it into an 9-symbol codeword. In the Reed-Solomon code, this codeword will consist of the 9 symbols:

$$\begin{aligned}
18 + 15 \cdot 0 + 4 \cdot 0^2 + 0 \cdot 0^3 + 10 \cdot 0^4 &= 18 \pmod{29} \\
18 + 15 \cdot 1 + 4 \cdot 1^2 + 0 \cdot 1^3 + 10 \cdot 1^4 &= 18 \pmod{29} \\
18 + 15 \cdot 2 + 4 \cdot 2^2 + 0 \cdot 2^3 + 10 \cdot 2^4 &= 21 \pmod{29} \\
&\vdots \\
18 + 15 \cdot 8 + 4 \cdot 8^2 + 0 \cdot 8^3 + 10 \cdot 8^4 &= 0 \pmod{29}.
\end{aligned}$$

So Alice will encode the message $(18, 15, 4, 0, 10)$ by the 9 symbol codeword $(18, 18, 21, 10, 5, 5, 17, 27, 0)$. This is what Alice will send through the channel, and what Bob expects to see if there are no errors. How does he go about recovering the message?

If there are no errors, he can do the following thing. Bob introduces variables $(x_1, x_2, x_3, x_4, x_5)$ that denote the five symbols in the message. To recover the message, he has to solve this system of equations:

$$\begin{aligned}
x_1 + x_2 \cdot 0 + x_3 \cdot 0^2 + x_4 \cdot 0^3 + x_5 \cdot 0^4 &= 18 \pmod{29} \\
x_1 + x_2 \cdot 1 + x_3 \cdot 1^2 + x_4 \cdot 1^3 + x_5 \cdot 1^4 &= 18 \pmod{29} \\
x_1 + x_2 \cdot 2 + x_3 \cdot 2^2 + x_4 \cdot 2^3 + x_5 \cdot 2^4 &= 21 \pmod{29} \\
&\vdots \\
x_1 + x_2 \cdot 8 + x_3 \cdot 8^2 + x_4 \cdot 8^3 + x_5 \cdot 8^4 &= 0 \pmod{29}.
\end{aligned}$$

As it turns out, this system always has a solution, even if you had only 5 equations, which allows Bob to recover the message.

But what happens if there are errors? It depends on what the errors are like. If Bob knows, for example, that the third and the fifth symbol in the codeword have been corrupted, so he receives

something like

$$\text{received word} = (18, 18, ?, 10, ?, 5, 17, 27, 0)$$

he can still recover the message: He simply discards the third and fifth equation and still has 7 equations left, well enough to solve for the unknown 5 symbol message. But what happens if some symbols are corrupted, but Bob doesn't know which ones? For instance, he receives the following corrupted codeword:

$$\text{received word} = (18, 19, 21, 10, 5, 5, 17, 27, 0)$$

Here is what he can do in this case. He tries to solve the system of equations, but it turns out that there is no solution. He then knows that some of the symbols have been corrupted. It is reasonable for him to start with the conservative assumption that there has been one corruption. How does he know which one? One possibility is to try them all: For every sequence of 9 symbols $(a_1, a_2, \ldots, a_9)$ that differs from the received word in exactly one place, check if the system of equations with $(a_1, \ldots, a_9)$ on the right hand side has a solution. If it does, Bob has reason to believe that the solution indeed equals the encoded message.

But what happens if there are several possible solutions? We will now argue that this cannot happen.

1 Minimum distance and decoding radius

Suppose you have an encoding Enc that takes a message of length k and turns it into a codeword of length n. For this code to be able to correct errors, the least we could ask is that Enc is *injective* – namely, distinct messages always map to distinct codewords. However, this is not good enough: If M and M' are distinct messages but the codewords $Enc(M)$ and $Enc(M')$ differ in only one position, then it is easy to confuse the two because if one of them becomes corrupted at this position it can easily turn into the other.

In general, it sounds reasonable that to tolerate more corruptions, we want to make distinct codewords as far apart from one another as possible.

Definition 1. *We say an encoding Enc has* minimum distance d *if for every pair of messages $M \neq M'$, the codewords $Enc(M)$ and $Enc(M')$ differ in at most d positions.*

There is a price we have to pay for minimum distance: The larger we want to make the distance, the longer the codewords we have to use. However, we can then also tolerate more and more errors:

Claim 2. *Suppose an encoding Enc has minimum distance d. Then there exists a decoding Dec such that for every message M and for every corrupted word $corr$ that differs from $Enc(M)$ in fewer than $d/2$ po, $Dec(corr) = M$.*

Proof. The decoding Dec does the following: When given a corrupted word $corr$, it finds the codeword $c = Enc(M)$ that differs from it in the fewest positions (breaking ties arbitrarily) and outputs the message M.

Now suppose for contradiction that there is some message M and a word $corr$ that differs from $Enc(M)$ in fewer than $d/2$ positions such that $Dec(corr) = M' \neq M$. Then $corr$ and $Enc(M')$ must differ in fewer than $d/2$ places, so $Enc(M)$ and $Enc(M')$ differ in fewer than d places, contradicting the assumption that the minimum distance of Enc is d. □

It turns out that the Reed-Solomon code with message length n and codeword length m has minimum distance $n - k + 1$. When $k = 5$ and $n = 9$, the minimum distance is 5, which allows for correction of up to two errors.

So now at least we know that in principle, we can recover from up to $d/2$ errors in the corrupted codeword. But we have to pay a high price for this. To recover from one corruption, we have to try to solve a system of equations for all possible words at distance one from the received word. There are about $29n$ such systems of equations. More generally, to handle c corruptions, we have to repeat the procedure roughly $(29)^c\binom{n}{c}$ times, which becomes intractable even for small values of c, like $c = 3$. Is there a better way?

It turns out that there is, but it involves rather complicated calculations with polynomials and linear equations. Instead I will show you a much simpler scheme that allows us to recover errors in time *linear* in the number of corruptions c.

2 Parity check codes

To describe the codes in question, it will be helpful to shift perspective. When we want to design a code, it is more natural to first think of a scheme to encode the data, and then worry about the decoding later. But since we are now interested in having very quick decoding procedures, let's start by designing the decoding first and later worry about how to encode the data.

For inspiration, let's start with a very simple code over alphabet $\{0, 1\}$. We will encode a length k message via the following scheme:

$$(x_1, x_2, \ldots, x_k) \to (x_1, x_2, \ldots, x_k, x_1 + x_2 + \cdots + x_k)$$

where $+$ is addition modulo 2, or the XOR operation.

This is not a very good error correcting code, as its minimum distance is 2. However, the code can still be used to *detect* single errors. To check if an error occurred, simply take all the symbols in the corrupted word and add them up modulo 2. If the sum is zero, we know that a single error could not have occurred.

To summarize, there is a very easy way to check if a word c is an actual codeword in this code: c is a codeword if and only if $c_1 + c_2 + \cdots + c_m = 0$. In general, instead of describing the code by the encoding procedure, we can simply give a list of constraints that the bits of the codewords must satisfy.

How can we design codes that are resilient against more errors? Here is one such code. This code takes messages of length 3, encodes them into codewords of length 7, and can detect up to two errors:

$$(x_1, x_2, x_3) \to (x_1, x_2, x_3, x_1 + x_2, x_1 + x_3, x_2 + x_3, x_1 + x_2 + x_3)$$

You can verify that the minimum distance of this code is 4. Its codewords are those strings $(y_1, y_2, y_3, y_4, y_5, y_6, y_7)$ that satisfy the constraints

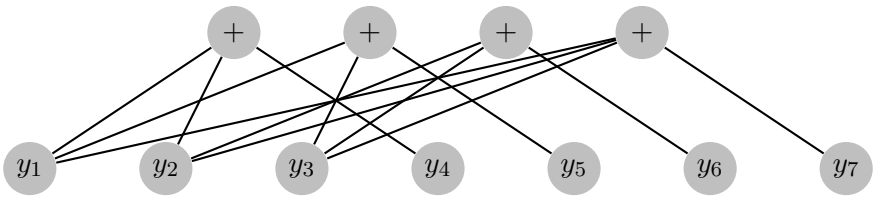
$$\begin{aligned} y_1 + y_2 + y_4 &= 0 \\ y_1 + y_3 + y_5 &= 0 \\ y_2 + y_3 + y_6 &= 0 \\ y_1 + y_2 + y_3 + y_7 &= 0. \end{aligned} \tag{1}$$

Notice something interesting: If the message length is k and the codeword length is n, then the number of constraints that describe the codewords is $n - k$. This is true as long as the code is *linear* – namely every bit of the codeword is a modulo 2 sum of a subset of its inputs – and all the bits of the codeword are linearly independent, and all parity check constraints are also linearly independent.

How do we go about getting codes that tolerate even more errors? You can imagine how the scheme we just described can be extended to longer messages, and it will tolerate a very large number of errors, more than half of the length of the codeword.[1] However, using this scheme is problematic for large values of k: It requires codewords of length $m = 2^k - 1$, which is really big even for say $k = 32$. What else can we do?

3 Random parity check codes

The trick is to pick a code where the parity check constraints are not chosen methodically, but at random. To explain how, it will help to think to represent the parity check constraints by a bipartite graph called the *constraint graph*. The bottom vertices of this graph represent the bits of the codeword. The top vertices represent the constraints, and edges indicate whether a codeword bit participates in the constraint. For example, we represent the constraints (1) by the this graph:



Now instead of settling for any specific constraint graph, we just want to choose one at random – but with the following requirements:

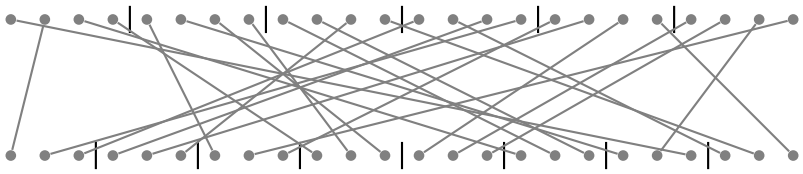
- Every constraint involves the same number d of codeword bits.
- Every bit of the codeword participates in the same number b of constraints.

Recall that if we want to encode k bits of information into an m bit codeword, the number of constraints is $n = m - k$, which is always smaller than m. To satisfy the above requirements, we must have $bm = dn$. How can we get a random constraint graph like this?

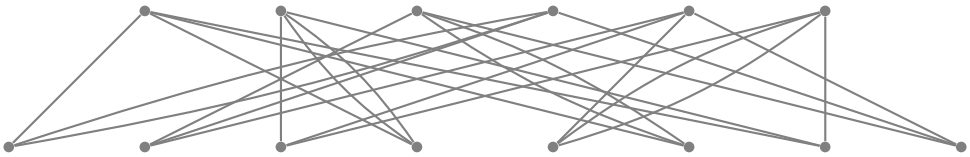
Here is one way to do it. We start with bm vertices at the bottom, dn vertices at the top, and choose a *random matching* from top to bottom. Then you divide the bottom vertices into n blocks with d vertices each, and the top vertices into m blocks with b vertices each. Finally, you contract all the vertices in the same block (that is, you turn them into one big vertex).

Here is an example with $n = 6$, $d = 4$, $m = 8$, $b = 3$:

[1] This code is called the *Hadamard code*.

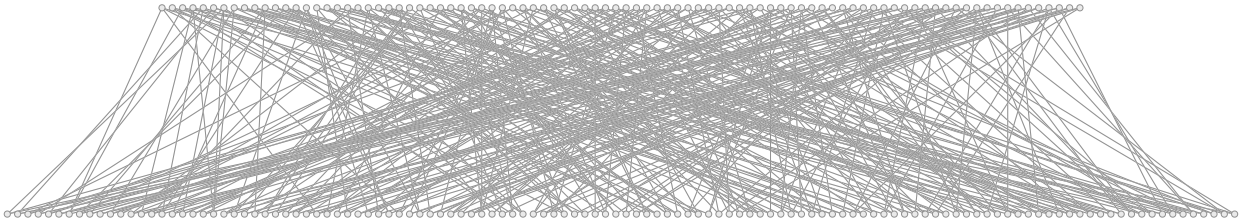


After contracting the vertices in each block, we obtain this graph:



This graph is regular. Notice how there are multiple edges between certain pairs of vertices. We will allow this.

Here is a much larger instance of a random graph (with $n = 90, d = 4, m = 120, b = 3$):



4 The distance of random parity check codes

Now that we have a candidate code – namely, a code with a random parity check matrix – we can go about designing a decoding algorithm for it. But before we do this, it will be helpful to understand the minimum distance of these codes.

Since the code in question is random, the minimum distance will be a random variable. For some parity check codes it could be low, for other ones it could be high. However, will argue that when the code is chosen at random, the minimum distance is unlikely to be too small.

How do we calculate the minimum distance of this random code? First let me show you a little trick. To argue that a random linear code has distance at least c, it is suffices to know that every non-zero codeword has *Hamming weight* at least c: That is, at least c positions of the codeword are 1. Here is why. Suppose the code has distance less than c. Then there exists two codewords, let's call them cw_1 and cw_2, so that the distance between them is less than c. Now we claim that if cw_1 and cw_2 are codewords, so is $cw_1 + cw_2$. This is easy to check: If all parity checks for cw_1 evaluate to zero, and all parity check for cw_2 evaluate to zero, then certainly they will all evaluate to zero for $cw_1 + cw_2$.

As a warm-up, let's begin by showing that the minimum distance is likely to be at least 2. By what we said, it is sufficient to argue that usually, every word of Hamming weight 1 is *not* a codeword. Let us show instead that the probability that *some* word of Hamming weight 1 *is* a codeword is

very small. How do we calculate this probability? There are n words of Hamming weight 1. Let us fix such a word w and suppose it is 1 in the ith position. Recall that the ith position participates in b parity checks. The only way for all of these b parity checks to evaluate to zero is if each of them covers the ith position an even number of times – in particular, more than once. This can happen only if there is a set of at most $b/2$ parity check vertices such that each one of them has at least two edges going to vertex i of the codeword.

We can turn this reasoning into a formula that upper bounds the probability that the distance is at most 1. Let's introduce some notation. Let W_1 be the set of all words of Hamming weight 1, w_i be the word that has 1 in position i and zero everywhere else, and let $P(w) = 0$ denote the event that the parity checks all evaluate to 0 for a word w. Finally, let S denote a subset of the parity check vertices and $e(S, \{i\})$ denote the number of edges between the vertices in S (at the top) and vertex i (at the bottom).

$$\begin{aligned}
\Pr[\text{distance} \le 1] &= \Pr[\exists w \in W : P(w)] \\
&\le \sum_{i=1}^{m} \Pr[P(w_i) = 0] \\
&\le \sum_{i=1}^{m} \Pr[\exists S : |S| \le b/2 \text{ and } e(S, \{i\}) \ge b] \\
&\le \sum_{i=1}^{m} \sum_{s=1}^{b/2} \sum_{S : |S|=s} \Pr[e(S, \{i\}) = b].
\end{aligned}$$

In all the lines of this calculation we have only used the union bound, which says that the probability of a union of events is at most the sum of the probabilities of the individual events in the union.

How do we calculate the probability that $e(S, \{i\}) = b$? To do this, we have to go back to our representation of the graph by matchings. Recall that vertex i was obtained by contracting b vertices at the bottom, while the vertices in S were obtained by contracting ds vertices at the top. The top and bottom vertices were connected by a random matching. We now ask what are the chances that in this matching, all b vertices at the bottom hit the set of ds vertices at the top. A simple calculation shows that this is

$$\Pr[e(S, \{i\}) = b] = \frac{ds}{dn} \cdot \frac{ds-1}{dn-1} \cdots \frac{ds-b+1}{dn-b+1} \le \left(\frac{s}{n}\right)^b \le \left(\frac{b}{2n}\right)^b.$$

Plugging this in, we obtain that[2]

$$\Pr[\text{distance} \le 1] \le \sum_{i=1}^{m} \sum_{s=1}^{b/2} \sum_{S : |S|=s} \left(\frac{b}{2n}\right)^b \le m\left(\frac{b}{2n}\right)^b \sum_{s=1}^{b/2} \binom{n}{s} \le \frac{m(n+b/2)^{b/2}}{(b/2)!}\left(\frac{b}{2n}\right)^b.$$

This probability is very small: For example, when $m = 120$, $n = 90$, $d = 8$, and $b = 6$, it is about 2%. Asymptotically, if d and b are constant and we let n become larger and larger, the probability that the distance is at most one is $O(1/n^{b/2-1})$.

So now we know that with pretty high probability, a random parity check code has minimum distance at least two. Is it larger than two? In fact a similar argument leads to the conclusion that with high probability, the minimum distance is quite large. We state this as a lemma, which you will prove as part of your homework.

[2]Here we assume b is even. If b is odd, the code has distance 2 or more with probability one, can you see why?

Lemma 3. *The probability that there exists a word w of Hamming weight h such that $P(w) = 0$ is at most*

$$\binom{m}{h} \cdot \frac{(n + bh/2)^{bh/2}}{(bh/2)!} \cdot \left(\frac{bh}{2n}\right)^{bh}.$$

When b, h, d are arbitrary constants and we let n go to infinity, this probability is at most $K/n^{bh/2-h}$, where K is some large constant. So by a union bound, the probability that there exists some $h, 1 \leq h \leq c$ such that $P(w) = 0$ for some w of Hamming weight h is at most

$$\sum_{h=1}^{c} \frac{K}{n^{bh/2-h}} \leq \frac{Kn^{b/2-1}}{1 - n^{b/2-1}} = O(n^{b/2-1}).$$

We have just proved the following theorem:

Theorem 4. *Fix b, c, d and let n go to infinity. The probability that a random parity check code has minimum distance at most c is at most $O(1/n^{b/2-1})$.*

In fact, with high probability the minimum distance is as large as αn for some constant $\alpha > 0$ that depends on b and d but not on n. But this is a bit more difficult to prove.

5 Decoding random parity check codes

We now go back to the question of how to decode random parity check codes.

Let us begin by investigating what happens when no error occurs in the transmission. If we receive a codeword cw, how are we supposed to decode it? This is a trick question as we never described how the *encoding* works in the first place.

Here is the trick. Given a constraint graph P of parity checks, we can reverse engineer the encoding procedure using a bit of linear algebra. Let's begin by working out an example – the parity check code given by the equations (1). In this case, we know how the encoding is supposed to work, but let's pretend that we don't and try to derive the encoding from the equations.

First, since there are $m = 7$ parity check bits $y_1, \ldots, y_7$ and $n = 4$ (linearly independent) constraints, we know that the message length is $k = m - n = 3$. Let's represent the message as (x_1, x_2, x_3). It turns out that without loss of generality (by some linear algebra), we may assume that the first 3 bits of the codeword equal the first 3 bits of the message, while the other 4 are some linear combinations of them. So we can write $y_1 = x_1, y_2 = x_2, y_3 = x_3$ and

$$\begin{aligned} y_4 &= a_{41}x_1 + a_{42}x_2 + a_{43}x_3 \\ y_5 &= a_{51}x_1 + a_{52}x_2 + a_{53}x_3 \\ y_6 &= a_{61}x_1 + a_{62}x_2 + a_{63}x_3 \\ y_7 &= a_{71}x_1 + a_{72}x_2 + a_{73}x_3 \end{aligned}$$

where a_{41} up to a_{73} are (for now) unknown coefficients (1 or 0). Let's see what kind of information we can get about these coefficients. From the equation $y_1 + y_2 + y_4 = 0$, we can conclude that

$$x_1 + x_2 + a_{41}x_1 + a_{42}x_2 + a_{43}x_3 = 0$$

Since this equation must hold for all messages (x_1, x_2, x_3), we must have $a_{41} = 1$ (when $x_1 = 0$ and $x_2 = x_3 = 0$), $a_{42} = 1$ (when $x_2 = 1$ and $x_1 = x_3 = 0$) and $a_{43} = 1$ (when $x_3 = 1$ and $x_1 = x_2 = 0$). By solving the equations related to the other constraints in a similar way, we find out that

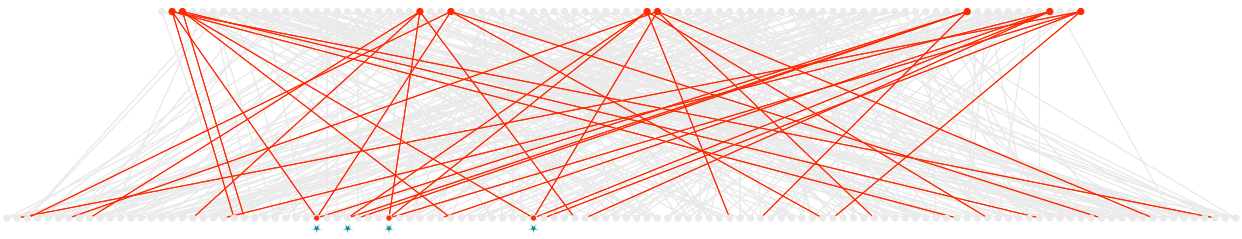
$$\begin{aligned} y_4 &= x_1 + x_2 \\ y_5 &= x_1 + x_3 \\ y_6 &= x_2 + x_3 \\ y_7 &= x_1 + x_2 + x_3. \end{aligned}$$

So we managed to reverse engineer the encoding algorithm from the parity check constraints. In fact this is always possible: For every parity constraint graph, we can set up a similar system of equations. The first k constraints $y_1, \ldots, y_k$ are equal to the first k bits $x_1, \ldots, x_k$ of the message, and to figure out the other ones, we set up the coefficients in the constraints as indeterminates and solve for them using the constraint equations.

One great thing about this scheme is that, at least in the case of no errors, the decoding is now extremely simple! To find the message corresponding to a codeword cw, we simply read off the first k bits of cw.

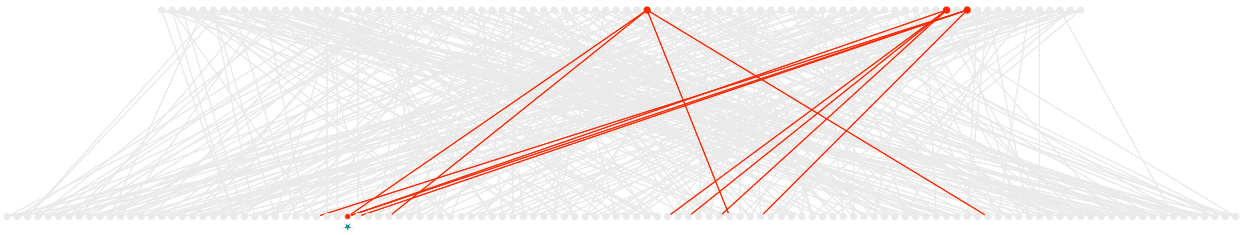
What happens when there are errors? It turns out that the decoding does not get much more complicated. To explain how it works, let's do an example. Here is a random parity check code, and suppose we have received a corrupted word, where the corruptions are marked in red (of course the decoder does not know where they are):

To check if the codeword is valid, the decoder evaluates the parity checks, but alas, some of them evaluate to 1! So the decoder knows there is a corruption, and decides to take a closer look at the bits of the received word that participate in these faulty constraints. In this drawing, the faulty constraints and their outgoing edges are marked in red:



This gives rise to many suspicious positions in the received word. But how do we know which ones are the corrupted ones? A good rule of thumb would be that the more faulty constraints a bottom vertex participates in, the more likely it is to be corrupt. In particular, if more than half the constraints that a bottom vertex participates in are dirty, then the vertex is suspicious. The

suspicious vertices are marked with a star. Notice that these include the corrupted bits of the codeword, but sometimes also some other ones! Nevertheless, let's *change* all the suspicious bits of the codeword (each 0 becomes a 1, and each 1 becomes a 0) and see what happens:



What happens is that there we got rid of some of the corruptions in the original word, but we may have introduced some new ones. However, notice that the we now have fewer violated constraints than before, and we can repeat the process until we get rid of all of them:

Let us now write a formal description of this decoding procedure:

Decoder for random parity check code P:
While $P(w) \neq 0$ (i.e., some parity checks are incorrect):
    Let F be the set of codeword positions such that
    more than $b/2$ of the parity checks evaluate to 1.
    Flip the bits of w in the positions indexed by F.
Output the first k bits of w.

How do we know that this process will ever terminate? In general, it won't. However, it turns out that for a random graph, not only will the process terminate, but it will do so very quickly: At each stage, the number of parity checks that evaluates to one decreases by a constant factor! Therefore, after $O(\log dc)$ steps, all the corruptions will be taken care of.

Lemma 5 (Sipser and Spielman)**.** *Fix any constants b and d and let $\alpha, \beta > 0$ be sufficiently small constants. For sufficiently large n, with probability at least 99% (over the choice of the random parity check code), the following is true. Suppose you are given any word w with is at distance at most αn from some codeword and this word causes p parity checks to evaluate to one. Then after flipping those bits of w that participate in more than $b/2$ parity checks that evaluate to one, the resulting codeword w' is also at distance at most αn from the same codeword, and at most $(1-\beta)p$ of the parity checks of w' evaluate to one.*

We won't prove this lemma, but you can try doing so at home. Its proof is more complicated than the proof of Lemma 3, but is based on very similar ideas.

Exercises

(a) Proof Lemma 3.

(b) Can you give a version of Theorem 4 that works even when $c = \alpha n$, where α is some small constant? The failure probability you obtain will not be $O(1/n^{b-2})$ but it can be made as small as, say, 1% when n is sufficiently large.

(c) Can you give an example of a (non-random) parity check code for which the decoding algorithm from Section 5 never terminates?

(d) Can you prove Lemma 5? (**Hints:** First try proving that the number of incorrect parity checks *strictly decreases* at every step, not necessarily by a factor of $1-\beta$. As an intermediate step, try to prove that the constraint graph has the following property with high probability: For every set of vertices T at the bottom of size at most αn, among the neighbors of T at the top, at least 3/4 have *exactly one* neighbor in S.)