ITCSC-INC Winter School 2015 26 January 2014
notes by Andrej Bogdanov

Today we will talk about randomness and some of the surprising roles it plays in the theory of computing and in coding theory.

Let's start with a contrived example to brush up on our probability.

1 Counting triangles and cliques

A *triangle* in a graph is a subset $\{x, y, z\}$ of three vertices such that $\{x, y\}$, $\{x, z\}$, and $\{y, z\}$ are all edges in the graph. For example, if the vertices represent users of a social network and edges represent friendships, then a triangle is a group of three mutual friends.

Given a graph with six vertices, does the graph contain a triangle?

This is not a precise mathematical question as I haven't told you what the graph is. One way to make sense of it is to look at a random graph on six vertices and ask for the *probability* that this graph contains a triangle.

For now, our model of a random graph on six vertices will be very simple: Each possible (simple) graph is equally likely to be chosen. In other words, each of the possible $\binom{6}{2} = 15$ edges appears in the graph independently with probability $1/2$. What is the probability that such a graph contains a triangle?

Since every graph is equally likely, the probability that a graph on 15 vertices contains a triangle equals the number of graphs with 6 vertices that contain a triangle divided by the total number of graphs with 6 vertices. Since each of the 15 edges can be present or absent, there are $2^{15} = 32,768$ possible graphs.

How do we calculate the number of such graphs that contain a triangle? I don't know of a good way to do so by hand, so I wrote a computer program that goes over all possible 2^{15} graphs and counts the ones that have a triangle. It tells me that there are 26,979 such graphs, so

$$\Pr_G[G \text{ contains a triangle}] = \frac{26,979}{32,768} \approx 0.823.$$

The probability that a random graph on 6 vertices contains a triangle is about 82%.

A k-clique in a graph is a subset of k vertices in which every pair forms an edge. You can think of it as a group of friends in a social network. My program tells me that the number of graphs on 6 vertices that contain a 4-clique is 5,142, so the probability that a 6 vertex graph contains a 4-clique is $5,142/32,768 \approx 0.157$.

Here is a quick way to obtain an *upper bound* on this probability. We will specify each vertex by a number in the set $\{1, 2, 3, 4, 5, 6\}$ and write T_S for the event that the set S is a clique. The probability that the graph contains a 4-clique is then

$$\Pr[T_S \text{ for some subset } S \text{ of four vertices}].$$

The *union bound* tells us that the probability that at least one event in a collection of event happens is at most the sum of the probabilities of all the events in the collection:

$$\Pr[T_S \text{ happens for at least one } S] \le \sum_S \Pr[T_S]$$

and so

$$\begin{aligned}\Pr_G[G \text{ contains a 4-clique}] &\le \sum_{\text{all subsets } S \text{ of size } 4} \Pr[T_S] \\ &= \sum_{\text{all subsets } S \text{ of size } 4} \Pr[(x,y) \text{ is an edge for all } x \ne y \in S] \\ &= \sum_{\text{all subsets } S \text{ of size } 4} \prod_{x \ne y \in S} \Pr[(x,y) \text{ is an edge}]\end{aligned}$$

because the events "(x, y) is an edge" are all independent, so the probability they all happen is the product of their probabilities, each of which equals 1/2. In every product there are $\binom{4}{2}$ items, so each term in the summation equals $2^{-\binom{4}{2}}$ and there are $\binom{6}{4}$ such terms. We get that

$$\Pr_G[G \text{ contains a 4-clique}] \le \binom{6}{4} \cdot 2^{-\binom{4}{2}} = 15 \cdot 2^{-6} = 15/32 \approx 0.469.$$

This is about three times worse than the value of 0.157 that I found on the computer but it took me a lot less work to get it.

What about the probability that a random graph on 7 vertices contains a 5-clique? The computer is too slow to produce an answer as it needs to consider all $2^{\binom{7}{2}} = 2{,}097{,}152$ possible graphs. Maybe this is because I am not a very clever programmer and you can do better; try your hand at it. But the method we just saw quickly tells us that for a random graph G on 7 vertices,

$$\Pr_G[G \text{ contains a 5-clique}] \le \frac{7}{5} 2^{-\binom{5}{2}} \approx 0.012$$

which is no more than 1%.

2 Randomness extraction

Many computer applications depend on the use of randomness. It is used in physical systems, communication protocols, and so on. Randomness is especially important in cryptography; without it there would be no secrets and no security.

Yet coming up with random numbers is not an easy task. Computers have access to a variety of processes that look random, but it is not always clear how to take advantage of them. For instance consider the following sequence of data

```
19 15 15 17 18 19 22 20 15 13
```

These are the mean temperatures measured in Hong Kong in the first ten days of December 2011. The numbers look somewhat random, but they also contain a large amount of non-random information. For example, they are all close to the annual mean winter temperature for the city and therefore correlated with one another.

How should we go about "extracting" the hidden randomness out of this data? Let's make things simple and suppose we want just one random bit, something that is 0 about half the time and 1 about half the time. One idea could be to compute the sum modulo two of the data

$$\mathtt{19} + \mathtt{15} + \mathtt{15} + \mathtt{17} + \mathtt{18} + \mathtt{19} + \mathtt{22} + \mathtt{20} + \mathtt{15} + \mathtt{13} \mod 2 = \mathtt{1}.$$

Now let's look at another data sequence

$$\mathtt{4\ 3\ 2\ 3\ 1\ 2\ 3\ 2\ 1\ 0}$$

which records the number of acquaintances of each person among a party of ten people. Surely this data also looks random, so if we want to get a random bit out of it we can again try to compute

$$\mathtt{4} + \mathtt{3} + \mathtt{2} + \mathtt{3} + \mathtt{1} + \mathtt{2} + \mathtt{3} + \mathtt{2} + \mathtt{1} + \mathtt{0} \mod 2 = \mathtt{0}.$$

Now repeat this experiment in different groups of people and you will discover that no matter which group you are looking at you will always get the answer `0`, which is not random at all! There is a simple explanation for this: Every pair of acquaintances was counted twice, so it is not surprise the sum is always an even number.

So the "randomness extraction" procedure that gave us a random bit out of a sequence of temperatures did not work for the sequence of acquaintance numbers. As computer scientists, we would like to have a universal procedure that we can apply to any data, provided the data contains some randomness. This is the problem of randomness extraction. Let's come up with a model for it and see what we can do.

Let us represent the random data X as a single number in the range $[n] = \{1, 2, \ldots, n\}$; think of n as being very large. There must be some uncertainty in the data, for otherwise there work be no randomness to extract from it. To keep our model simple, we will view this uncertainty as a subset $S \subseteq [n]$ of values that X could be taking. In the acquaintances example, S could comprise all the degree sequences that we may expect to observe at a random 10-person party. What we want to do now is extract some randomness out of this number; but we don't really know what the set S is, as this depends on the specifics of the data collection procedure. All we know is that since the sequence has some randomness in it, S should not be too small; to model this aspect we introduce a parameter K and require that S has size at least k.

We can specify the extraction procedure by a *subset* E of $[n]$: If the data falls inside E the procedure outputs 1, if not it outputs 0. Our model of a "random bit" will be very simple: We will merely ask that the procedure sometimes outputs 0 and it sometimes outputs 1; that is the set S needs to have some elements inside the set E and some elements outside the set. So we need to answer this question:

> Can you come up with a subset $E \subseteq [n]$ such that for every subset $S \subseteq [n]$ of size at least k, S has at least one element inside E and at least one element outside E?

A moment's thought shows that this is impossible unless $k > n/2$: We can always choose S to be the larger of the two sets E and its complement $\overline{E}$.

Two-source hitters This is bad news, so we need to make some additional assumptions about our data if we are to extract any randomness out of it. One possibility is to make restrictions on what S should look like beyond its size. But today we will look at something different.

In the scenario we looked at the extraction procedure had access to one data point from the set S. But suppose instead that it had access to two *independent* data points – say a sequence of temperatures as well as a sequence of acquaintance numbers. Can we extract some randomness from these two data points?

Let's model the two data sets as a pair of subsets $S, T \subseteq [n]$, each of size at least k. Now each pair of data items $x \in S, y \in T$ is a possible outcome. Our goal is to "extract" a random bit without knowing what S and T are.

Now the extraction procedure is specified not by a set, but by a *bipartite graph* with n left vertices and n right vertices. Given an outcome (x, y), $x \in S, y \in T$, if there is an edge between x and y we output a 1, and otherwise we output a 0.

The object we are looking at is called a two-source hitter in computer science, or bipartite Ramsey graph in mathematics.

Definition 1. *A* k-clique *in a bipartite graph is a pair of subsets S, T of the left and right vertices, respectively, of size k each such that (x, y) is an edge for every $x \in S$ and every $y \in T$.*

An k-independent set *in a bipartite graph is a pair of subsets S, T of the left and right vertices, respectively, of size k each such that (x, y) is not an edge for every $x \in S$ and every $y \in T$.*

A graph is a two-source hitter if there is at least one edge and at least one non-edge between all sufficiently large S and T; that is, the graph must avoid large cliques and large independent sets.

Definition 2. *An (n, k) two-source hitter is a bipartite graph G with n vertices on the left and n vertices on the right that does not contain a k-clique and does not contain a k-independent set.*

such that for every subset S of vertices on the left and subset T of vertices on the right, both of size k, the set $S \times T$ is not a clique and it is not an independent set.

Obtaining two-source hitters The existence of two-source hitters was shown by Erdös in 1946 in one of the earliest uses of the probabilistic method. Erdös showed that for a suitable choice of k (in terms of n), if the graph G is chosen at random then the probability that G is a two-source hitter is strictly greater than zero. So at least one G must be a two-source hitter.

We now upper bound the probability that G is *not* a two-source hitter.

$$\begin{aligned}\Pr[G \text{ is not a two-source hitter}] &= \Pr[G \text{ contains a } k\text{-clique or a } k\text{-ind. set}] \\ &\le \Pr[G \text{ contains a } k\text{-clique}] + \Pr[G \text{ contains a } k\text{-ind. set}]\end{aligned}$$

To bound these probabilities we do a similar calculation as above:

$$\begin{aligned}
\Pr[G \text{ contains a } k\text{-clique}] &= \Pr[\exists S, T\colon x, y \text{ is an edge for all } x \in S, y \in T] \\
&\le \sum_{S,T} \Pr[x, y \text{ is an edge for all } x \in S, y \in T] \\
&= \sum_{S,T} \prod_{x \in S, y \in T} \Pr[x, y \text{ is an edge}] \\
&= \binom{n}{k} \cdot \binom{n}{k} \cdot 2^{-k^2} \\
&\le n^{2k} \cdot 2^{-k^2} \\
&= 2^{2k \log_2 n - k^2}.
\end{aligned}$$

This probability is strictly smaller than one as long as $k^2 > 2k \log_2 n$, or $k > 2 \log_2 n$. Therefore (n, k) two-source hitters exist even when $k = 2 \log_2 n + 1$. How to find such hitters is a major unsolved problem in the theory of computing.

3 Error-correcting codes

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.

We will represent the possible messages as bit strings of a fixed length k. (You can think of this as the binary representation of the string you want to encode.) The number of possible messages that can be encoded is 2^k. To each such message x we associate a *codeword* $Enc(x)$, which we represent as a bit string of length $n \ge k$. The number n is called the *block length* of the code.

Here is an example for $k = 4$ and $n = 5$. We will encode a length k message via the following scheme:

$$Enc(x_1, x_2, x_3, x_4) = (x_1, x_2, x_3, x_4, x_1 + x_2 + x_3 + x_4)$$

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

This code has the following property. If x and x' are two different messages, the encodings $Enc(x)$ and $Enc(x')$ differ in at least two positions: If x and x' differ in at least two positions, this is clearly the case; otherwise, they differ in exactly one position, but the parity of the sums is different so the last entry of the codeword will also differ.

This is an example of an *error-detecting code* for one error: If one symbol of the codeword is flipped, the resulting string is no longer a codeword. (We will shortly discuss how to detect if a string is a codeword.) However, it is not error-correcting: The corrupted codeword does not uniquely determine the message that it came from. For example, $(1, 0, 0, 0, 0)$ may be a one symbol corruption of either $(0, 0, 0, 0, 0)$ or $(1, 0, 0, 0, 1)$, both of which are valid codewords.

Here is another example for $k = 4$ and $n = 7$, called the Hamming code, that also allows for error correction, as long as there is up to one error:

$$Enc(x_1, x_2, x_3, x_4) = (x_1, x_2, x_3, x_4, x_1 + x_2 + x_3, x_1 + x_2 + x_4, x_1 + x_3 + x_4).$$

In the exercises you will show that if x and x' are different messages, then the codewords $Enc(x)$ and $Enc(x')$ differ in at least three positions. If you corrupt a symbol of any given codeword, there is still a unique "closest codeword" to it: All others differ from the corrupted one in at least two positions. So at least in principle, it is possible to eliminate the corruptions and recover the message.

We now generalize this type of analysis to allow for more than one error.

4 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. 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 3. *We say an encoding Enc has* minimum distance *at least d if for every pair of messages $x \neq x'$, the codewords $Enc(x)$ and $Enc(x')$ differ in at least 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 4. *Suppose an encoding Enc has minimum distance d. Then there exists a decoding Dec such that for every message x and for every corrupted word y that differs from $Enc(x)$ in fewer than $d/2$ positions, $Dec(y) = x$.*

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

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

We will be interested in the asymptotic problem of designing codes for an arbitrarily large message length k that can tolerate a certain "quota" of errors.

The simplest example of a code of minimum distance d is the *repetition code*

$$Enc(x_1, x_2, \dots, x_k) = (x_1, \dots, x_1, x_2, \dots, x_2, \dots, x_k, \dots, x_k)$$

in which every symbol in the message is repeated d times. This code has block length $n = kd$. Such a code can handle an arbitrary *number* of errors, but this is not particularly interesting.

It is more realistic to expect that some *fraction* of bits in the codeword will be corrupted. For example, this happens when every bit is corrupted independently with some probability around δ. To design codes for this regime, we will think of the minimum distance d as being of the form δn for some constant δ

An example of a code of minimum distance more than $n/2$ is the *Hadamard code.* The Hadamard code of message length $k = 3$ is the following code with block length $n = 7$:

$$Enc(x_1, x_2, x_3) = (x_1, x_2, x_3, x_1 + x_2, x_1 + x_3, x_2 + x_3, x_1 + x_2 + x_3)$$

The minimum distance of this code is 4, which is greater than $n/2$. In general, the Hadamard code for message length k is given by

$$Enc(x_1, x_2, \ldots, x_k) = \Big(\sum_{i\in S} x_i\Big)_{S:\, S\subseteq[k], S\neq\varnothing}.$$

There is one codeword symbol for every nonempty subset of the message bits; the corresponding codeword entry is the modulo 2 sum of those message bits. Since there are $2^k - 1$ nonempty subsets of $[k]$, the message length is $n = 2^k - 1$. In the exercises you will show that the distance of this code is $d = 2^{k-1} - 1 > n/2$. Even for small values of k, say $k = 32$, the block length of this code is enormous.

We now have examples a code with large distance; so in principle, we can correct from a large numbers of errors. But how do we go about recovering from these errors? Let us begin with the simpler task of checking whether a codeword has been corrupted.

5 Parity check constraints

The types of codes we saw are called *linear codes*: Every bit of the codeword is a (modulo 2) linear function of the message bits. There is an alternative way to describe such codes. Instead of telling you how a codeword is calculated, I give you a collection of constraints that every codeword must satisfy.

To show how this is done, let us look at the error-detecting code

$$Enc(x_1, x_2, x_3, x_4) = (x_1, x_2, x_3, x_4, x_1 + x_2 + x_3 + x_4)$$

Given a potentially corrupted codeword, how do we check if an error has occurred? There is a simple procedure for this: 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: y is a codeword if and only if $y_1 + y_2 + y_3 + y_4 + y_5 = 0$. 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.

Similarly, the codewords of the Hadamard code for message length 3 are those 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 for all linear codes as long as all the bits of the codeword are linearly independent, and all parity check constraints are also linearly independent.

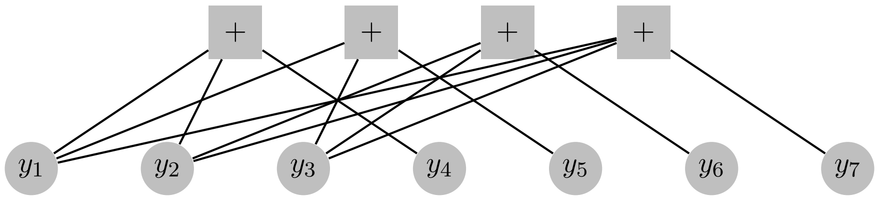
To check if y is a codeword of a given linear code, all we have to do is verify that y satisfies all $n-k$ parity check constraints of that code. What if the codeword is corrupted? If there is one corruption, we can guess where this corruption occurred, flip the corresponding bit in the codeword, and then perform the parity checks. After we go over all n possible guesses, we can be sure to recover the true codeword (and read off the message from its first k positions).

More generally, if we know that e errors have occurred, we can guess which one of the $\binom{n}{e}$ error patterns has occurred and do the parity checks. Even for small values of n and e, e.g. $n = 500$ and $e = 8$, this number is very large.

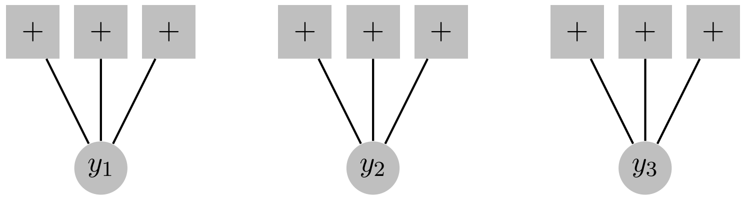
We will now see a way to obtain codes in which decoding can be done in time linear in the message length k, even when the number of corruptions grows linearly with the block length n, which is itself linear in k — a far cry from the Hadamard code!

6 Tanner graphs and decoding

The trick is to pick a code where the parity check constraints are not chosen methodically, but at random. To explain how and why, it will help to think to represent the parity check constraints by a bipartite graph called the *Tanner graph* of the code.[1] 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:



As an unrealistic but instructive example, consider a code whose Tanner graph consists of codeword bits with *disjoint* sets of neighbours:



This is not the actual Tanner graph of any code, since there should be fewer parity check bits than codeword bits; but we will use it to make a point anyway. If a symbol in the codeword of this

[1]In computer science, this type of graph is sometimes also called a constraint graph.

"code" is corrupted, all three parity checks it is represented in change their value from zero to one. To recover the codeword, we can then go over all the codeword symbols and flip their value if the parity checks they are represented in evaluate to one instead of zero.

In fact, for this decoding to be correct it is sufficient that the *corrupted* codeword bits have disjoint neighbourhoods, since all the other ones contribute nothing to the parity checks. Unfortunately even this condition is too much to ensure because if there is a parity check of degree at least two, it could be the case that two of its neighbours are corrupted and the resulting graph won't have this form.

The main insight we will use is that even if the neighbourhoods of the corrupted codeword bits are not disjoint, error correction along these lines should still be possible. For instance, suppose that codeword bits y_i and y_j each participate in three parity checks, only one of which is common to both. If y_i and y_j are both corrupted, the majority of parity checks each of them participates in will still evaluate to one! We can use this criterion as an indication that the corresponding codeword bit has been flipped.

Graphs that have this property of "almost disjoint neighbourhoods" for sets of vertices that are not too large are called *expanders*.

7 Expander codes

A bipartite graph with n left vertices and $3n/4$ right vertices that is b-regular on the left is called an expander if every set S of at most $n/(100b)$ vertices on the left has at least $(7/8)b|S|$ neighbours on the right. Theorem 8 below states that such graphs exist for infinitely many values of n.

(There is nothing special about these particular constants, but they work.) Consider a linear code whose Tanner graph is an expander. We apply the following decoding procedure to correct a possibly corrupted codeword y:

```
Procedure $Correct(y)$:
While $y$ is not a codeword (i.e., some parity checks are incorrect):
    Let $F$ be the set of codeword positions such that
    a majority of the parity checks evaluate to 1.
    Flip the bits of $y$ in the positions indexed by $F$.
Output $y$.
```

We will show that this correction algorithm is not only correct, but in the homework you will argue that it is also very fast. Let us show correctness first.

Theorem 5. *Suppose there exists a codeword c such that y differs from c in fewer than $n/(400b)$ positions. Then $Correct(y)$ outputs c.*

We will argue that every iteration of the while loop decreases the distance between y and c by at least a factor of two, so the algorithm must terminate in at most $\log_2 n$ iterations. Let S be the set of bits in which y and c differ before the iteration, and F be the set of bits that are flipped in the iteration. The set of bits on which y and c differ after the iteration is then the symmetric difference $S \oplus F$.

As a warmup, let us first show that

Lemma 6. $|F| \leq 3|S|$.

Proof. We will first assume that $|F| \leq n/(100b)$ and then argue that this assumption can be made without loss of generality.

For a vertex to make it into F, the majority of its neighbours must be parity checks that evaluate to 1. Therefore at most $(1/2)b|F|$ of the neighbours of F can evaluate to zero. By expansion, F has at least $(7/8)b|F|$ neighbours, so it must have at least $(3/8)b|F|$ neighbours that evaluate to 1. Each of those neighbours must be adjacent to at least one vertex in S (for otherwise the parity check would have been correct). Since S can have at most $b|S|$ neighbours, we get that $(3/8)b|F| \leq b|S|$, so $|F| \leq (8/3)|S| \leq 3|S|$ as desired.

If $|F| > n/(100b)$, the same argument shows that any subset of S' of size $n/(100b)$ in fact has size at most $3|S| < n/(100b)$, a contradiction. □

We can now show the main lemma of the analysis:

Lemma 7. $|F \oplus S| \leq |S|/2$.

Proof. We show that every vertex in $F \oplus S$ has a majority of neighbours that have at least one more neighbour in $F \cup S$.

A vertex is in $S - F$ if it is corrupted but not flipped. This means a majority of the parity checks it participates in must evaluate to zero, so they must involve at least one other corrupted vertex.

A vertex is in $F - S$ if it was flipped but not corrupted. Then a majority of the parity checks it participates in must involve at least one corrupted neighbour

As in the proof of the previous lemma, we can say that at least $(3/8)b|F \oplus S|$ neighbours of $|F \oplus S|$ have at least two neighbours in $F \cup S$. So the total number of vertices that can be neighbours of $F \cup S$ is at most $b|F \cup S| - (3/8)b|F \oplus S|$. On the other hand, by Lemma 6 $|F \cup S| \leq 4|S| \leq n/100$, so by expansion $|F \cup S| \geq (7/8)b|F \cup S|$. We conclude that

$$|F \oplus S| \leq \frac{1}{3}|F \cup S| \leq \frac{1}{3}|S| + \frac{1}{3}|F \oplus S|.$$

Rearranging terms we obtain the desired inequality. □

8 Random graphs and expansion

The codes in the previous section relied on the existence of expander graphs. We state this as a theorem.

Theorem 8. *If b is a sufficiently large constant, there exist expanders for every sufficiently large value of n (that is a multiple of 4.*

To prove this theorem, you need to consider a random *regular* bipartite graph and show that such a graph is an expander with high probability. We do this in a slightly more general setting where we allow any number of vertices (not just $n/2$) on the right.

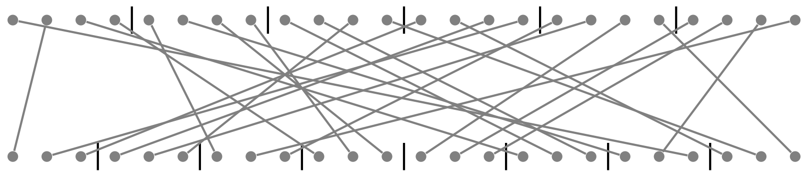
We now describe a process by which a random regular graph can be generated. We will consider the following type of graph:

- Every parity check vertex involves the same number d of codeword bits.
- Every bit of the codeword participates in the same number b of parity checks.

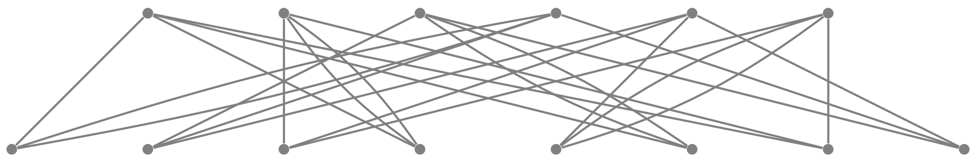
To satisfy the above requirements, we must have $bm = dn$. How can we get a random bipartite graph like this?

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 $n = 6$, $d = 4$, $m = 8$, $b = 3$:

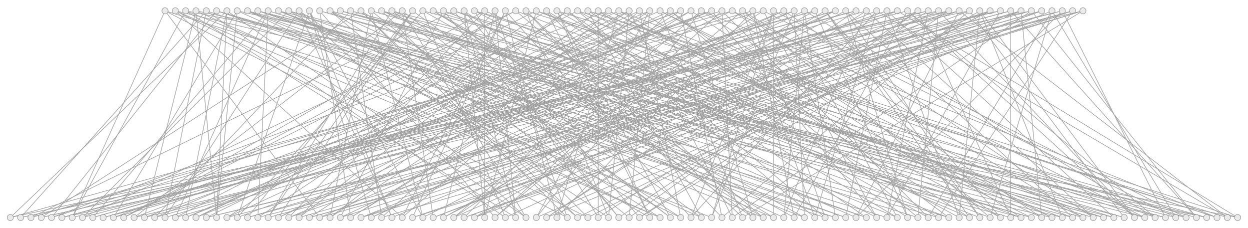


After contracting the vertices in each block, we obtain this graph:



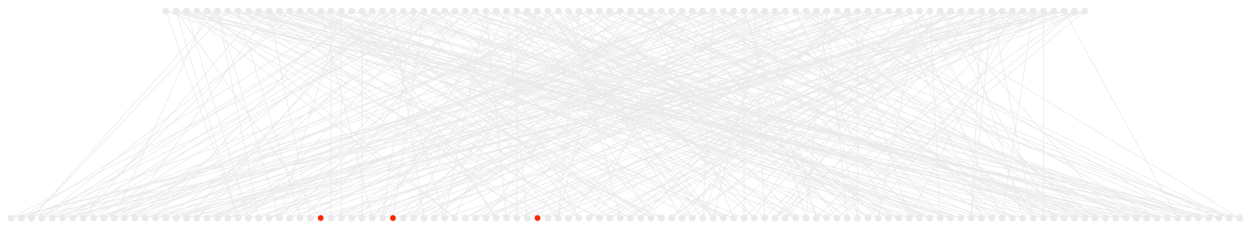
This graph is regular. Notice how there are multiple edges between certain pairs of vertices. We will allow this.

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

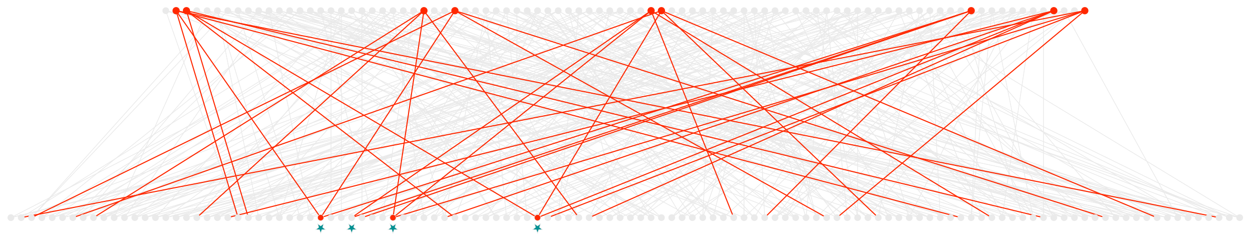


In the homework you will argue that when b is a sufficiently large constant, the probability that a random regular bipartite graph is *not* an expander is strictly less than 1.

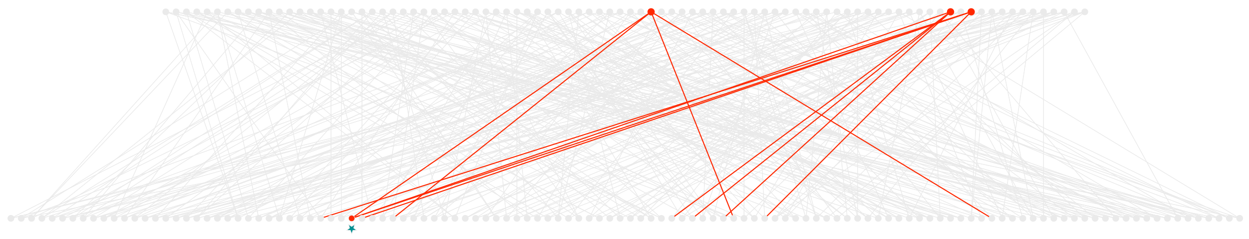
Now here is an example of the decoding algorithm "in action". The corrupted codeword bits are marked in red.



To check if the codeword is valid, the decoder evaluates the parity checks, and 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:



After one iteration of the decoding procedure, we end up with the following.



We got rid of some corruptions in the original word, but may have introduced some new ones. However, the number of violated constraints is smaller, and we repeat one more time until we get rid of all the violations: