Codes: How to Protect Your Data

Michael Mitzenmacher

1 Introduction

This chapter is going to be about *error-correcting codes*. Just to set the stage, we'll start with a quick little puzzle for you to think about (if you don't like puzzles, you can just read the answer). Suppose that, after meeting you at a conference, I find we have more to talk about, so I want to give you my telephone number – ten digits. (If you're single, feel free to put this story in the context of meeting a special someone in a bar or other locale, but being married, these days I just meet colleagues at conferences.) Sadly, my handwriting is rather messy, and I have a tendency to smudge, so I am worried you will not be able to read one (or more) of the digits later. Let us assume that if you can't clearly read a digit, you will just ignore it, so you will not end up making a mistake, deciding a 3 is a 7. But you may end up not knowing all the digits in my phone number. You might read

$$617 - 555 - 0?23,$$

where we use the question mark to mean you were not sure what that number was.

I could write my phone number down for you twice, or even three times, and then you would be much more likely to be able to determine my phone number. By just repeating the phone number twice, I could guarantee you would know it if any single digit was erased; in fact, you would know the number as long as both copies of a digit were not erased. Unless I expect a great number of messy smudges, though, repeating the number seems like overkill. So here is the challenge – can I write down an eleventh digit that will allow you to correct for *any single* missing digit? You might first want to consider a slightly easier problem: can I write down an extra number between 1 and 100 that will allow you to correct for any single missing digit?

In this puzzle, we are trying to come up with a *code*. Generally, a code is used to protect data during transmission from specific types of errors. In this example, a number that is so messy or smudged that you cannot tell what it is would be called an *erasure* in coding terminology, so our resulting code would be called an *erasure code*. There are many different types of errors that can be introduced besides erasures. I might write down (or you might read) a digit incorrectly, turning a 7 into a 4. I might transpose two digits, writing 37 when I meant 73. I might forget to write a number,

so you only have nine digits instead of the ten you would expect. There are codes for these and other more complicated types of errors.

Codes protect data by adding redundancy to it. Perhaps the most basic type of coding is just simple repetition: write everything down two or three or more times. Repetition can be effective, but it is often very expensive. Usually, each piece of data being transmitted costs something – time, space, or actual money – so repeating everything means paying at least twice as much. Because of this, codes are generally designed to provide the most bang for the buck, solving as many or as many different kinds of errors as possible with the least additional redundancy.

This brings us back to the puzzle. If I wanted to make sure you could correct for the erasure of any single digit, I could provide you with the sum of the digits in my phone number. If one digit then went missing, you could subtract the other numbers to find it. For example, if I wrote

$$617 - 555 - 0123 \;\; 35,$$

and you read

$$617 - 555 - 0?23 \;\; 35,$$

you could compute

$$35 - (6 + 1 + 7 + 5 + 5 + 5 + 0 + 2 + 3) = 1$$

to find the missing number.

We can reduce the amount of information passed even further because, in this case, you really do not even need the tens digit of the sum. Instead of writing 35, I could just write down 5. This is because no matter what digit is missing, the sum you will get from all of the remaining digits will be between 26 and 35. You will therefore be able to conclude that the sum must have been 35, and not 25 or smaller (too low) or 45 or larger (too big). Just one extra digit is enough to allow you to handle any single erasure.

It is interesting to consider how helpful this extra information would be in the face of other types of errors. If you misread exactly one of the digits in the phone number, you would see that there was a problem somewhere. For example, if you thought what I wrote was

$$617 - 855 - 0123 \;\; 5,$$

you would see the phone number and the ones digit of sum did not match, since the sum of the digits in the phone number is 38. In this case, the extra information allows you to *detect* the error, but it does not allow you to *correct* the error, as there are many ways a single digit could be have been changed to end up with this sequence. For example, instead of

$$617 - 555 - 0123,$$

my phone number might originally have been

$$617 - 852 - 0123,$$

which also differs from what you received in just one digit, and also has all the digits sum to 35, matching the extra digit 5 that was sent. Since without additional information you cannot tell what my original number was, you can only detect but not correct the error. In many situations, detecting errors can be just as or almost as valuable as correcting errors, and generally detection is less expensive than correction, so sometimes people use codes to detect rather than correct errors.

If there were two changed digits, you might detect that there is an error. Or you might not! If you read

$$617 - 556 - 0723\ \ 5,$$

the extra sum digit does not match the sum, so you know there is an error. But if you read

$$617 - 555 - 8323\ \ 5,$$

you would think everything seemed fine. The two errors match up in just the right way to make the extra digit match. In a similar manner, providing the sum does not help if the error is a transposition. If instead of

$$617 - 555 - 0123\ \ 5,$$

you thought I wrote

$$617 - 555 - 1023\ \ 5,$$

that would seem perfectly fine, since transpositions do not change the sum.

1.1 Where are codes used?

You are probably using codes all the time, without even knowing it. For example, every time you take your credit card out of your wallet and put the bill on the plastic, you are using a code. The last digit of your credit card number is derived from all of the previous digits, in a manner very similar to the scheme we described to handle erasures and errors in the telephone number puzzle. This extra digit prevents people from just making up a credit card number off the top of their heads; since one must get the last digit right, at most only one in ten numbers will be valid. The extra digit also prevents mistakes when transcribing a credit card number. Of course, transposition is a common error, and as we've seen, using the sum of the digits cannot handle transposition errors. A more difficult calculation is made for a credit card, using a standard called the *Luhn formula*, which detects all single-digit errors (like the sum) and most transpositions. Additional information of this sort, which is used to detect errors instead of correct them, is typically called a *checksum*.

Error-correcting codes are also used to protect data on compact discs (CDs) and digital video discs (DVDs, which are also sometimes called digital versatile discs, since they can hold more than video). CDs use a method of error-correction known as a cross-interleaved Reed-Solomon code (CIRC). We'll say a bit more about Reed-Solomon codes later on. This version of Reed-Solomon code is especially designed to

handle bursts of errors, which might arise from a small scratch on the CD, so that the original data can be reconstructed. The code can also correct for other errors, such as small manufacturing errors in the disc. Roughly one fourth of the data stored on a CD is actually redundancy due to this code, so there is a price to pay for this protection. DVDs use a somewhat improved version known as a Reed-Solomon product code, which uses about half as much redundancy as the original CIRC approach. More generally, error-correcting codes are commonly used in various storage devices, such as computer hard drives. Coding technology has proven key to fulfilling our desire for easy storage and access to audio and video data.

Of course, codes for error correction and error detection also come into play in almost all communication technologies. Your cell phone, for example, uses codes. In fact, your cell phone uses multiple complex codes for different purposes at different points. Your iPod uses codes. Computer modems, fax machines, and high definition television use error-correction techniques. Moving from the everyday to the more esoteric, codes are also commonly used in deep space satellites to protect communication. When pictures are sent back to NASA from the far reaches of our solar system, they come protected by codes.

In short, coding technology provides a linchpin for all manner of communication technology. If you want to protect your data from errors, or just detect errors when they occur, you apply some sort of code. The price for this protection is paid with redundancy and computation. One of the key things people who work on codes try to determine is how cheap we can make this protection while still making it as strong as possible.

2 Reed-Solomon codes

The invention of Reed-Solomon codes revolutionized the theory and practice of coding, and they are still in widespread use today. Reed-Solomon codes were invented around 1960 by two scientists, Irving Reed and Gustave Solomon. The codes can be used to protect against both erasures and errors. Also importantly, there are efficient algorithms for both *encoding*, or generating the information to send, and *decoding*, or reconstructing the original message from the information received. Fast algorithms for decoding Reed-Solomon codes were developed soon after the invention of the codes themselves by Berlekamp and Welch. Most of the coding circuits that have ever been built implement Reed-Solomon codes, and use some variation of Berlekamp-Welch decoding.

The basic idea behind Reed-Solomon codes can be explained with a simple example. I would like to send you just two numbers, for example a 3 followed by a 5. Suppose that I want to protect against erasures. We will think of these numbers as not just being numbers, but being points on a line: the first number becomes the point $(1, 3)$, to denote that the first number is a 3. The second number becomes the

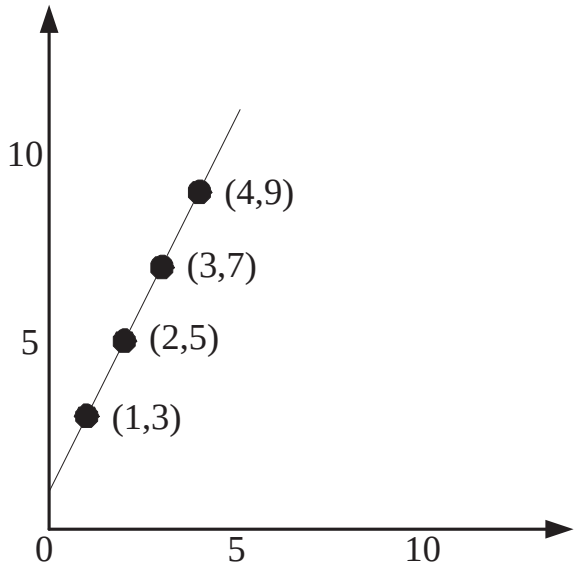

Figure 1: An example of Reed-Solomon codes. Given the two numbers 3 and 5, we construct the line between the two points $(1, 3)$ and $(2, 5)$ to determine additional numbers to send. Receiving any two points, such as the points $(3, 7)$ and $(4, 9)$, will allow us to reconstruct the line and determine the original message!

point $(2, 5)$, to denote that the second number is a 5. The line between these points can be pictured graphically, as in Figure 1, or can be thought of arithmetically: the second coordinate is obtained by doubling the first, then adding 1. *My goal will be to make sure that you obtain enough information to reconstruct the line.* Once you can reconstruct the line, you can determine the message, simply by finding the first two points $(1, 3)$ and $(2, 5)$. You then know I meant to send a 3 and a 5. The key idea is that instead of thinking about the data itself, we think about a line that encodes the data, and focus on that.

To cope with erasures, I can just send you other points on the line! That is, I can send you extra information by finding the next points on the line, $(3, 7)$ and $(4, 9)$. I would send the second coordinates, in order:

$$3, 5, 7, 9.$$

I could send more values – the next would be 11, for $(5, 11)$ – as many as I want. The more values I send, the more erasures you can tolerate, but the cost is more redundancy.

Now, as long as *any* two values make it to you, you can find my original message. How does this work? Suppose you receive just the 7 and 9, so what you receive looks like

$$?, ?, 7, 9,$$

where the '?' again means the number was erased. You know that those last two numbers correspond to the points $(3, 7)$ and $(4, 9)$, since the 7 is in the 3rd spot on

your list and the 9 is in the 4th. Given these two points, you can yourself draw the line between them, because *two points determine a single line*! The line is exactly the same line as in Figure 1. Now that you know the line, you can determine the message.

The key fact we are using here is that two points determine a line, so once you have any two points, you are done. There is nothing particularly special about the number two here. If I wanted to send you three numbers, I would have to use a parabola, instead of a line, since any three points determine a parabola. If I wanted to send you 100 numbers, I could build a curve determined by any 100 points. Such curves are written as polynomials: to handle 100 points, I would use curves with points (x, y) satisfying an equation looking like $y = a_{99}x^{99} + a_{98}x^{98} + \ldots + a_1x + a_0$, where the a_i are appropriately chosen numbers so that all the points satisfy the equation.

Reed-Solomon codes have an amazing property, with respect to erasures: if I am trying to send you 100 numbers, we can design a code so that you get the message as soon as you receive any 100 numbers I send. And there is nothing special about 100; if I am trying to send you k numbers, all you need to receive are k numbers, and any k will do. In this setting, Reed-Solomon codes are optimal, in the sense that if I want to send you 100 numbers, you really have to receive some 100 numbers from me to have a chance to get the message. What is surprising is that it does not matter which 100 they are! Also, importantly, there are efficient algorithms for both *encoding*, or generating numbers to send, and *decoding*, or determining the original message from the numbers sent.

An important detail you might be wondering about is what happens if one of the numbers I am supposed to send ends up not being an integer. Things would become a lot more complicated in practice if I had to send something like 16.124875. Similar problems might arise if the numbers I could send could become arbitrarily long; this too would be impractical. To avoid this, all work can be done in *modular* or *clock* arithmetic. In modular arithmetic, we always take the remainder after dividing by some fixed number. If I work "modulo 17", instead of sending the number 47, I would send the remainder after dividing 47 by 17. This remainder would be 13, since $47 = 2 \times 17 + 13$. Modular arithmetic is also called clock arithmetic, because it works like counting on a clock. In Figure 2 we see a clock corresponding to counting modulo 5. After we get to 4, when we add 1, we go back to 0 again (since the remainder when dividing 5 by 5 is 0). We have already seen an example of modular arithmetic in our original puzzle. Instead of sending the entire sum, I could get away with just sending the ones digit, which corresponds to working "modulo 10". It turns out that all the arithmetic for Reed-Solomon codes can be performed modulo a big prime number, and with this, all the numbers that need to be sent will be suitably small integers. (Big primes are nice mathematically for various reasons. In particular, each non-zero number has a *multiplicative inverse*, which is a corresponding number whose product with the original number is one. For example, 6 and 2 are inverses modulo 11, since $6 \times 2 = 12 = 11 + 1$, so 6×2 is equivalent to 1 modulo 11. In practice, things are slightly more complicated; one often does not work modulo some prime, but uses

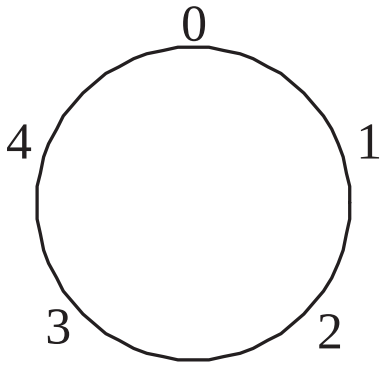

Figure 2: Modular arithmetic: counting "modulo 5" is like counting on a clock that goes back to zero after four. Equivalently, you just consider the remainder when dividing by 5, so 7 is the same as 2 modulo 5.

a number system with similar properties, including the property that each non-zero number has a multiplicative inverse.)

What about dealing with errors, instead of erasures? As long as the number of errors is small enough, all is well. For example, suppose again I sent you the numbers

$$3, 5, 7, 9,$$

but you received the numbers

$$3, 4, 7, 9,$$

so that there is one error. If we plot the corresponding points, $(1, 3), (2, 4), (3, 7)$, and $(4, 9)$, you can see that there is only one line that manages to pass through three of the four points, namely the original line. Once you have the original line, you can correct the point $(2, 4)$ to the true point $(2, 5)$, and recover the message. If there were too many errors, it is possible that we would obtain no line at all passing through three points, in which case we would detect that there were too many errors. For example, see Figure 4. Another possibility is that if there were too many errors, you might come up with the wrong line, and you would decode incorrectly. See Figure 4 for an example of this case. Again, there is nothing special about wanting to send two numbers. If I wanted to send you three numbers, and cope with one error, I would send you five points on a parabola. If there was just one error, there would be just one parabola passing through four of the five points. The idea extends to larger messages, and the Berlekamp-Welch algorithm decodes efficiently even in the face of such errors.

In general, if I am trying to send you a message with k numbers, in order for you to cope with e errors, I need to send you $k + 2e$ numbers using a Reed-Solomon code. That is, each error to be handled requires sending *two* additional numbers. Therefore, to send you two numbers and handle one error I needed to send $2 + 2 \cdot 1 = 4$ symbols. Also, there are efficient algorithms for both encoding and decoding in the face of errors.

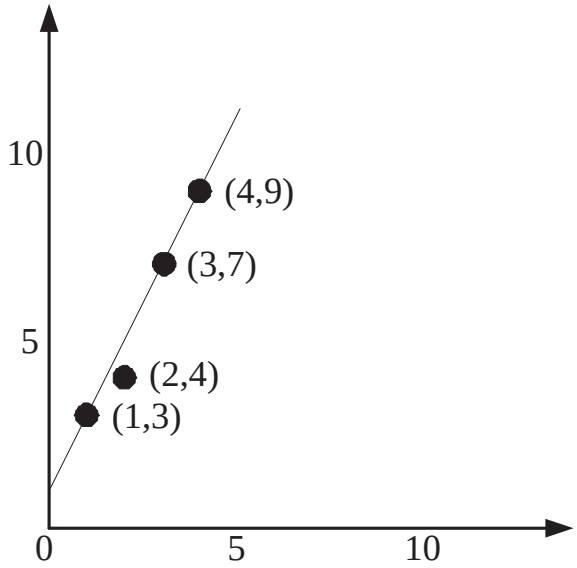

Figure 3: An example of decoding Reed-Solomon codes when there are errors. Given the four points, one of which is in error, there is just one line that goes through three of the four points, namely the line corresponding to the original message. Determining this line allows us to correctly reconstruct the message.

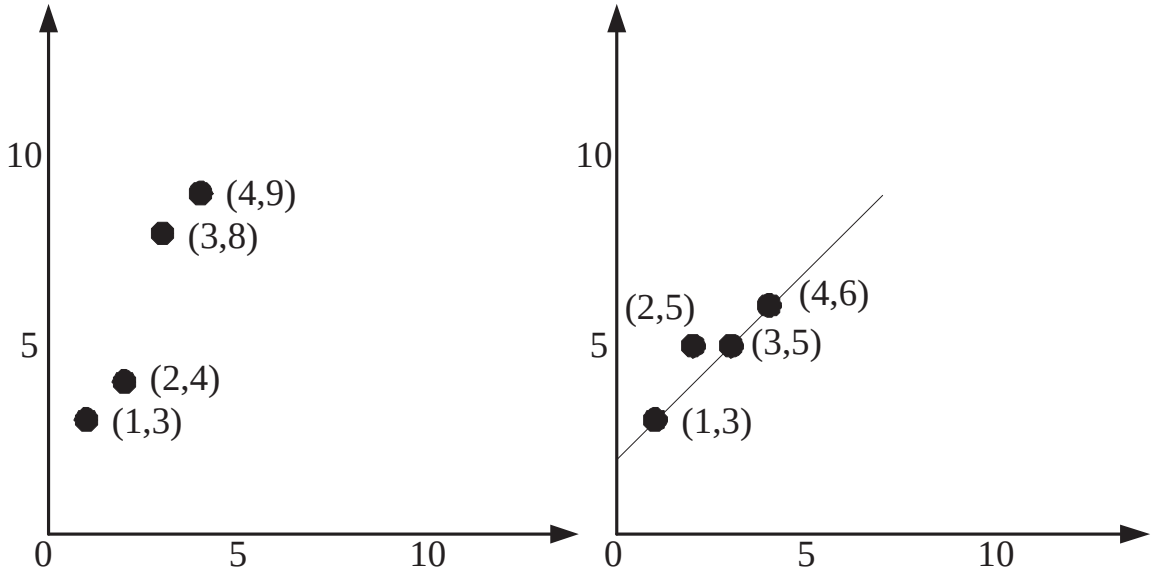

Figure 4: Examples of decoding Reed-Solomon codes when there are too many errors. Given the four points, when there are two or more errors, there may be no line that goes through three or more of the points, as in the example on the left, in which case one can detect there is an error but not correct it. Alternatively, when there are two errors, three of the points may lie on an incorrect line. On the right hand side, one could find a line going through three of the points, and incorrectly conclude that the point $(2, 5)$ should have been $(2, 4)$, giving an incorrect decoding.

Because Reed-Solomon codes have proven incredibly useful and powerful, for many years, it proved hard to move beyond them, even though there were reasons to do so. On the theoretical side, Reed-Solomon codes were not optimal for many types of errors. Theoreticians always like to have the best answer, or at least something very close to it. On the practical side, although Reed-Solomon codes are very fast for small messages, they are not very efficient when used for larger messages. This is in part because of the overhead of using modular arithmetic, which can be non-trivial, and because the decoding schemes just take longer when used for messages with lots of numbers and lots of erasures or errors. Specifically, the decoding time is roughly proportional to the product of the length of the message and the number of errors, or ke. This means that if I want to double the message length and handle twice as many errors, so that the overall percentage of errors remains the same, the time to decode increases by roughly a factor of four. If I want the message length to increase by a factor of one hundred, and handle the same percentage of errors, the time to decode increases by roughly a factor of 10,000, which substantial! The problems of dealing with larger messages were not too important until computers and networks became so powerful that sending huge messages of megabytes or even gigabytes became ordinary. These problems could be dealt with using tricks like breaking up a large message into several small messages, but such approaches were never entirely satisfactory, and people began looking for other ways to code to protect against erasures and errors.

3 New Coding Techniques: Low-Density Parity-Check Codes

Over the past fifteen years, a new style of coding has come into play. The theory of these codes has become quite solidly grounded, and the number of systems using these codes have been growing rapidly. Although there are many variations on this style, they are grouped together under the name Low-Density Parity-Check codes, or LDPC codes for short. LDPC codes are especially good for situations where you want to encode large quantities of data, such as movies or large software programs.

Like Reed-Solomon codes, LDPC codes are based on equations. With Reed-Solomon codes, there was one equation based on *all* of the data in the message. Because of this, Reed-Solomon codes are referred to as *dense*. LDPC codes generally work differently, using lots of small equations based on small parts of the data in the message. Hence they are called low-density.

The equations used in LDPC codes are generally based on the exclusive-or (XOR) operation, which works on individual bits. If the two bits are the same, the result is 0, otherwise it is 1. So writing XOR as $\oplus$, we have

$$0 \oplus 0 = 0; 1 \oplus 1 = 0; 1 \oplus 0 = 1; 0 \oplus 1 = 1.$$

Another way of thinking about the XOR operation is that it is like counting modulo

2, which corresponds to a clock with just two numbers, 0 and 1. This makes it easy to see that for example

$$1 \oplus 0 \oplus 0 \oplus 1 = 0.$$

We can extend XOR operations to bigger strings of bits of the same length, by simply XORing the corresponding bits together. For example, we would have

$$10101010 \oplus 01101001 = 11000011,$$

where the first bit of the answer is obtained from taking the XOR of the first bits of the two strings on the left, and so on.

The XOR operation also has the pleasant property that for any string S, $S \oplus S$ consists only of zeroes. This makes it easy to solve equations using XOR, like the equations you may have seen in algebra. For example, if we have

$$X \oplus 10101010 = 11010001,$$

we can "add" 10101010 to both sides to get

$$X \oplus (10101010 \oplus 10101010) = 11010001 \oplus 10101010.$$

This simplifies since $10101010 \oplus 10101010 = 00000000$ to

$$X = 01111011.$$

We will now look at how LDPC codes work in the setting where there are erasures. LDPC codes are based on the following idea. I will split my message up into smaller blocks. For example, when you send data over the Internet, you break it up into equal-sized packets. Each packet of data could be a block. Once the message is broken into blocks, I repeatedly send you the XOR of a small number of random blocks. One way to think of this is that I am sending you a bunch of equations. For example, if you had blocks of eight bits, labeled $X_1, X_2, X_3, X_4 \ldots$, taking on the values

$$X_1 = 01100110, X_2 = 01111011, X_3 = 10101010, X_4 = 01010111, \ldots,$$

I might send you $X_1 \oplus X_3 \oplus X_4 = 10011011$. Or I could just send you $X_3 = 10101010$. I could send you $X_5 \oplus X_{11} \oplus X_{33} \oplus X_{74} \oplus X_{99} \oplus X_{111} = 10111111$. Notice that whenever I send you a block, I have to choose *how many* message blocks to XOR together, and then *which* message blocks to XOR together.

This may seem a little strange, but it turns out to work out very nicely. In fact, things work out remarkably nicely when I do things *randomly*, in the following way. Each time I want to send out an encoded block , I will randomly pick how many message blocks to XOR together according to a distribution: perhaps 1/2 of the time I send you just 1 message block, 1/6 of the time I send you the XOR of 2 message

blocks, 1/12 of the time I send you the XOR of 3 message blocks, and so on. Once I decide how many blocks to send you, I choose which blocks to XOR uniformly, so each block is equally likely to be chosen.

With this approach, some of the information I send could be redundant, and therefore useless. For example, I might send you $X_3 = 10101010$ twice, and certainly you only need that information once. This sort of useless information does not arise with Reed-Solomon codes, but in return LDPC codes have an advantage in speed. It turns out that if I choose just the right random way to pick how many message blocks to XOR together, then there will be very little extra useless information, and you will almost surely be able to decode after almost just the right number of blocks. For example, suppose I had a message of 10,000 blocks. On average, you might need about 10,250 blocks to decode with a well-designed code, and you would be almost surely be done after 10,500 blocks.

LDPC codes can be decoded in an interesting and quite speedy way. The basic idea works like this: suppose I receive an equation $X_3 = 10101010$, and another equation $X_2 \oplus X_3 = 11010001$. Since I know the value of X_3, I can substitute the value of X_3 into the second equation so that it becomes $X_2 \oplus 10101010 = 11010001$. Now this equation has just one variable, so we can solve it, to obtain $X_2 = 01111011$. We can then substitute this derived value for X_2 into any other equations with the variable X_2, and look for further equations with just one variable left that can be solved. If all works out, we can just keep substituting values for variables, and solving simple equations with just one variable, until everything is recovered. At some point, we get a chain reaction, where solving for one variable lets us solve for more variables, which lets us solve for more variables, until we end up solving everything!

Just as Reed-Solomon codes can also be used to deal with errors instead of simply erasures, variations of LDPC codes can also be used to deal with errors. One of the amazing things about LDPC codes is that for many basic settings one can prove that they perform almost optimally. That is, the theory tells us that if we send data over a channel with certain types of errors, there is a limit to how much useful information we can expect to get. For erasures, as we saw with Reed-Solomon codes, this limit is trivial and can be achieved: to obtain a message of 100 numbers, you need to receive at least 100 numbers, and Reed-Solomon codes allow you to decode once you receive any 100 numbers. LDPC codes, in many situations with other types of errors, brush right up against the theoretical limits of what is possible! LDPC codes are so close to optimal we can hope to do very little better in this respect.

4 Network Codes

One thought that might be going through your mind is that if the recent success of low-density parity-check codes means we can reach the theoretical limits in practice, is coding research essentially over? This is not at all a strange idea. It comes up as

a point of discussion at scientific conferences, and certainly it is possible for scientific subfields to grow so mature that there seem to be few good problems left.

I am happy to report that coding theory is far from dead. There are many areas still wide open, and many fundamental questions left to resolve. Here I will describe one amazing area that is quite young and currently full of tremendous energy.

For the next decade, a key area of research will revolve around what is known as *network coding*. Network coding has the potential to transform the underlying communication infrastructure, but it is too early to tell if the idea will prove itself in the real world, or simply be a mathematical curiosity.

To understand the importance of network coding, it is important to understand the way networks have behaved over the last few decades. Computer networks, and in particular the Internet, have developed using an architecture based on routers. If I am sending a file from one side of the country to the other, the data is broken into small chunks, called packets, and the packets are passed through a series of specialized machines, called routers, that try to move the packets to the end destination. While a router could, conceivably, take the data in the packets and examine, change, or re-organize them in various ways, the Internet was designed so that all a router had to do is look at where the packet is going, and pass it on to the next hop on its route. This step is called *forwarding* the packet. By making the job of the routers incredibly simple, companies have been able to make them incredibly fast, increasing the speed of the network and making it possible to download Web pages, movies, or academic papers at amazing speeds.

The implications of this design for coding was clear: if you wanted to use a code to protect your data, the encoding should be done at the machine sending the data, called the *source*, before the packets were put on the network. Decoding should be done at the destination, after the packets were taken off the network. The routers would be uninvolved, so they could concentrate on *their* job, passing on the data.

Currently, this network design principle is being re-examined. Scientists and the people who run networks have been considering whether routers can do more than simply route. For example, it would be nice if routers could find and remove harmful traffic, such as viruses and worms, as soon as possible. Or perhaps routers could monitor traffic and record statistics that could be use for billing customers. The reason for this change in mindset has to do with how the Internet has developed and changes in technology. As issues like hacking attacks and e-commerce have moved to the forefront, there has been a push to think about what more the network can do. And as routers have grown in speed and sophistication, it becomes natural to question whether in the future they should be designed to do more, instead of just forwarding packets even faster.

Once you lose the mindset that all the router should do is forward packets, all sorts of questions arise. In the context of coding, we might ask if the routers can usefully be involved with coding, or if coding can somehow help the routers. These

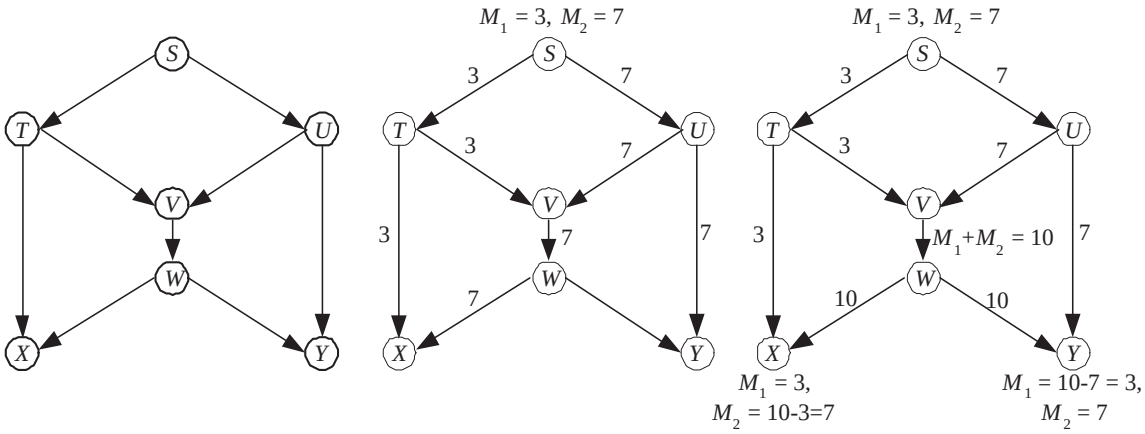

Figure 5: Network coding in action. The left hand side is the original network. The middle shows what happens if you just use forwarding; there is a bottleneck from V to W, slowing things down. I can get both numbers to X or to Y, but not to both simultaneously. The right hand side demonstrates network coding. I avoid the bottleneck by sending the sum of M_1 and M_2 to both X and Y by way of V.

questions represent a fundamental shift in how scientists usually think about coding, since usually we think of encoding as being done by the source and decoding being done by the receiver, without any coding being done along the way. Once people started asking these questions, interesting and innovative results started to arise, leading to the new field of network coding.

What sort of coding could you do on a network? There is a nice prototypical example, given pictorially in Figure 5. To explain the example, it helps to start by thinking about the connections as pipes, carrying commodities like water and oil. Suppose that I control the supplies of oil and water, located at S, and I want to ship water and oil along the connections shown, which are my pipes. I can send one gallon per minute of either on any pipe, but I can't send both water and oil on the same pipe at the same time. After all, oil and water do not mix! Can I get two gallons of water and oil per minute flowing at each the final destinations X and Y?

Obviously, the answer is no. The problem is that I can only get two gallons of stuff per minute out of S in total, so there is no way I can get two gallons of both oil and water to each of X and Y.

Now suppose that instead of sending oil and water I was sending data – numbers – corresponding to movies that people want to download. The network would look the same as my original network in Figure 5, but now all my pipes are really fiber-optic cables, and I can carry 10 Megabits per second along each cable. Can I get both movies to both final destinations each at a rate of 10 Megabits per second? If I just use forwarding, it does not seem possible, as shown in the middle of Figure 5. To simplify things, suppose I just have two numbers, $M_1 = 3$ and $M_2 = 7$, that need to get to each destination X and Y, without any interference anywhere along the path.

Because I can copy numbers at intermediate points, the fact that there are just two pipes out of S is not immediately a problem. There are two pipes into X and two pipes into Y, but it does not seem like I can make effective use of all of them, because of the bottleneck from V to W. I can send $M_1 = 3$ to X by going from S to T to X, and similarly I can send $M_2 = 7$ to Y by going from S to U to Y. But if I then try to send M_1 to Y by way of V, and M_2 to X by way of V, the two messages will meet at V, and one will have to wait for the other to be sent, slowing down the network.

One way to fix this would just be to speed up the link between V and W, so that both messages could be sent. If I could send two messages on that link at a time, instead of just one, the problem would disappear. Equivalently, I could build a second link between V and W.

Is there any way around the problem without adding a link between V and W or speeding up the link that is already there? Amazingly, the answer is *yes*! The reason why is that, unlike in the case of shipping physical commodities like oil and water, data can be mixed! To see how this would work, consider the right side Figure 5. I start sending my data as before, but now, when the 3 and the 7 get to V, I add the two numbers and transport their sum to X and Y through W. This mixes the data together, but when everything gets to the destinations, the data can be unmixed just by subtracting. At X, we take the 10 and subtract off $M_1 = 3$ to get $M_2 = 7$, and at Y, we take the 10 and subtract off $M_2 = 7$ to get $M_1 = 3$. A little addition, and a little subtraction, and the bottleneck disappears! The magic is that the sum $M_1 + M_2$ is useful at both X and Y.

Network coding is naturally much harder than this simple example suggests. In a large network, determining the best ways to mix things together can require some work! In many cases, we do not yet know what gains are possible using network coding. But there is a great deal of excitement as new challenges arise from our new view of how the network might work. Our understanding of network coding is really just beginning, and its practical impact, to this point, has been negligible. But there is a great deal of potential. Perhaps someday most of the data traversing networks will be making use of some form of network coding, and what seems novel today will become commonplace.

5 Places to Start Looking for More Information

- `http://en.wikipedia.org/wiki/Luhn_algorithm`
- `http://en.wikipedia.org/wiki/Checksum`
- `http://en.wikipedia.org/wiki/Reed-Solomon_error_correction`
- `http://en.wikipedia.org/wiki/Cross-interleaved_Reed-Solomon_coding`
- `http://en.wikipedia.org/wiki/Low-density_parity-check_code`

- http://en.wikipedia.org/wiki/Network_coding
- *Information Theory, Inference, and Learning Algorithms*, by David MacKay. The book is also available online at http://www.inference.phy.cam.ac.uk/mackay/itila/book.html .
- The Network Coding Home Page, available at http://www.ifp.uiuc.edu/~koetter/NWC/index.html .