5 – Error Correcting Codes

Luc Lapointe
luc.lapointe@ens-paris-saclay.fr
home.lmf.cnrs.fr/LucLapointe/

Abstract

In most modern communication channels, when a message is sent, some bits are corrupted by noise. Similarly, in modern storage devices, some bits are lost overtime. *Error Correcting Codes* are tools designed to *detect* such corruption or loss, and sometimes even correct it.
They can also apply to quantum technologies, where qubits are even less reliable than classical bits.

Introduction riddle: Dwarves and Dragon

100 dwarves are the prisoners of a dragon. The dragon is hungry, but not enough to eat the full 100 dwarves, so he offers them to play a game to eat some, and release some other. The rules of the game are the following:

1. At the start of the game, there is one dwarf on each step of the stairs going outside of the dragon cave.
2. The dragon puts either a white hat or a black hat on the head of each dwarf. He choses the color at random. No dwarf can see the color of its hat, but the first one, up the stairs, can see the color of the 99 next ones. The second can see the 98 next ones, and so on.
3. The dragon asks the first dwarf the color of its hat. The dwarf is only allowed to say out loud black or white. If it says the color of its hat, the dragon releases the dwarf. If it is wrong, the dragon eats the dwarf. Then the dragon asks the second dwarf, and so on.

The dwarves discuss the offer, then agree to play without cheating. The dragon expects to eat around 50 dwarves, but in the end it eats way less than that. How did the dwarves do?

1. Context

Most everyday messages, files, and more generally data are encoded by large sequences of bits. For this reason, in this lecture, we only consider error correction and detection in sequences of bits.

1.1. Modern data is compressed

Most encodings of messages or files are designed to use as few bits as possible. For example, to encode one of the 26 letters of the usual alphabets, one possible encoding is to use 5 bits, because $2^5 = 32 \geq 26$, to encode the letters as follow:

$$00000 \to \mathrm{A} \qquad 00001 \to \mathrm{B} \qquad 00010 \to \mathrm{C} \qquad \ldots \qquad 11010 \to \mathrm{Z}$$

Sending n letters with this encoding takes $5n$ bits.

The efficiency of the encoding depends of how many different messages you want to be able to write. For example, if you only want to communicate two different words: "HAPPY" or "SAD", you can use previous encoding to send a 25-bit long or a 15-bit long message. But you can also design a specific 1-bit long encoding: 1 to say you're happy, and 0 to say you're sad.

An encoding that can describe exactly 2^n different messages with n bits is a *perfect* encoding. It is *very* efficient: you can describe exponentially more messages than the amount of bits you're sending. The 5-bits long encoding of letters described previously is not a perfect encoding, but using the last 6 combinations to describe typographic characters, e.g. `␣ . , ' ! ?` can make it perfect.

Modern file formats such as `mp3`, `jpeg`, `gif` and so on are not perfect encodings, because the data they're encoding is too complex, but one of their aim is to use as efficiently as possible every single bit, so that they can describe a lot of different things with as few bits as possible.

To do so, they use *compression algorithms* and some other tricks. For example, encoding a square of $n \times n$ black pixels might take n^2 bits (i.e. an amount of bits quadratic in n), or you could write "black square n". As the binary writing of n only uses $\log_2(n)$ bits, describing a black square with this technique uses $C + \log_2(n)$ bits, where C is the amount of bits used to describe "black square". When n is big, this technique with $C + \log_2(n)$ bits uses way less bits than the one with n^2 bits. This technique can be used to describe with few bits pictures with uniform backgrounds, or videos where the image is not changing much (seeing successive black squares a black cube).

1.2. Detecting and correcting errors

When sending messages, either via wifi, cellular data, cables or other means, it is always possible that noise between the sender and the receiver changes the content of the message. Similarly, storing information on some device can be seen as sending a message from past to future, and this message may also be affected by noise.

When using a (close to) perfect encoding, every single bit of the message is useful. This means that any small noise can give the message a very different meaning.

Examples

- In the 1-bit long code describing "HAPPY" or "SAD", flipping the bit changes one mood to its opposite.
- When encoding binary numbers in 32 bits, flipping the 20th bit adds or removes 2^{20} to the number it is encoding; and flipping the sign bit changes a positive value to a negative one.
- When communicating a room coordinate at the ENS with a binary encoding, flipping a bit can change one letter to a different one (e.g. 1Z31 $\rightarrow$ 1P31).

This weakness can be addressed by adding *redundancy* in the encodings. *Error correcting codes* are how we call the tools with redundancy to detect or correct errors.

Exercise: My first error correction code

1. Think of a very simple way to add redundancy in an encoding.

Most error correcting codes can detect *up to* a given amount of errors. It is impossible to detect *for sure* that there has been no modification in the message, as a massive corruption can change a valid message to another valid message. But with good codes, you can detect errors if there are less than n_d, and even correct them if there are less than n_c.

2. How many redundancy do you need to add in your simple code to be able to detect errors if there are less than n_d?
3. How many redundancy do you need to add in your simple code to be able to correct errors if there are less than n_c?

2. Error correcting codes

2.1. Examples of codes

QR codes are commonly used to encode URLs. But it is less known that they can be used to encode any kind of text, of any size, and not necessarily links!

QR codes include in their design an error correction code, that can correct up to 30% of the content of the message. Try to hide part of the the code when scanning it: it should work!

Another kind of error detection technique is using **hash functions**.

Definition: Hash function

A **hash function** is a mathematical function that maps any kind of data, seen as a long sequence of 0 and 1, to a fixed-size "short" and "human-readable" sequence of numbers, called the **signature** of the file, with the following properties:

- Using twice the function on the same data must give the same result,
- Using the function on different data, even when they are very close, must give very different results,
- The risk of having two different files with the same signature must be very low.

Example

Signatures are of common use when downloading huge datasets or programs. If the server shares the signature of the files, it is possible to check once the data is downloaded that the signature is the same. If not, it means that the data has been corrupted while downloading.

Exercise: Risk of collision

When using a hash function h, a *hash collision* occurs when two different files $f_1 \neq f_2$ have the same hash: $h(f_1) = h(f_2)$.

Assume the hash function returns a n-digit long sequence, uniformly distributed among all the files. What is the risk of a hash collision between two files? How safe is it to hash 1 billion files with $n = 32$?

Same question if the hash function returns not only digits but also the 26 letters.

An example of tool that can calculate and verify signatures is md5sum.

2.2. Parity check

We have seen previously that a first simple idea to detect one error in a message is to write it twice. This method works, but doubles the size of each message.

It is however possible to detect one error in a message by using only one more bit, instead of doubling the size of the message: the *parity bit*.

Definition: Parity bit

In a message, seen as a sequence of bits, the number of 0 is either even or odd. Adding a **parity bit** can make it always even: if the original parity of 0 is odd, add a 0 parity bit. If it is even, add 1.

Example

- If the original message is 01101001, add a 1 in front of the message to preserve the even number of 0.
- If the original message is 10011101, add a 0 instead.

When you receive a message that uses the parity bit technique, you can never be sure that there is no error in it. Indeed, if two different bits are corrupted, the number of 0 is still even.

What you know precisely is:

- If you receive a message with an *even* number of 0, the number of errors in the message is either 0 or $2, 4, 6\ldots$.

- If you receive a message with an *odd* number of 0, the number of errors in the message is either 1 or $3, 5, 7\ldots$ and, in particular, **there is at least one error in the message**.

This technique is very simple but has the nice property that it uses only one bit, and can be extended to not only detect but also locate and correct errors, with *Hamming Codes*.

2.3. Hamming codes

Inspired by 3Blue1Brown video on this topic [1].

Hamming Codes is a natural extension of parity bit, designed to locate an error in some data. The main idea is that it uses *slightly* more than one parity bit, and each bit makes the parity of the number of 0 in one zone of the data even. With this technique, an error can be located by excluding the zones without an error and intersecting the zones with one.

The code applies more effectively to packs of data that that contain a square number of bits, and is described more easily when data is arranged in a square.

Exercise: From lines to squares

1. Find a simple bijective map from $[\![1, k^2]\!]$ to $[\![1, k]\!] \times [\![1, k]\!]$ that transforms a line of k^2 bits into a square of bits.
2. How would you store a data of arbitrary size n bits to square(s) with sides that are power of 2 long ?

In the following, we will only consider data arranged in a 4×4 square of bits, rather than a line of 16 bits.

$$1100110001000000 \quad \mapsto \quad \begin{matrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{matrix}$$

For a square of size 2^k sides, hamming code uses $2k + 1$ parity bits. This means that when you double the size of your data, you only need 2 parity bits more!

In a 4×4 square, the first 4 parity bits are located at very specific positions:

$$\begin{matrix} \cdot & \text{X} & \text{X} & \cdot \\ \text{X} & \cdot & \cdot & \cdot \\ \text{X} & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{matrix}$$

The second parity bit on the first line makes the number of 0 in the two last columns even. The first parity bit on the first line makes the number of 0 in second and fourth column.

·	·	X	·
·	·	·	·
·	·	·	·
·	·	·	·

·	X	·	·
·	·	·	·
·	·	·	·
·	·	·	·

Parity bits on the first column are similar: the first ones make the number of 0 in the second and fourth column even, and the second one in the third and fourth.

·	·	·	·
·	·	·	·
X	·	·	·
·	·	·	·

·	·	·	·
X	·	·	·
·	·	·	·
·	·	·	·

Example Consider the following square of data:

1			1
	1	0	0
	1	0	0
0	0	1	0

After adding the parity bits, it looks like this:

1	**1**	**0**	1
0	1	0	0
0	1	0	0
0	0	1	0

After corrupting the bit on the 3rd line and 3rd column, the number of zeros on columns 2 and 4 is still even, but not anymore on columns 3 and 4. As there is an error in column 3 or 4 but neither in column 2 nor 4, you know there is one in column 3.

1	1	0	1
0	1	0	0
0	1	**1**	0
0	0	1	0

1	1	0	1
0	1	0	0
0	1	**1**	0
0	0	1	0

Similarly, you can detect that there is an error in line 3 or 4, but not in line 2 nor 4; and conclude that there is an error in line 3, column 3.

Exercise: Test the Hamming code

1. Fill a square of size 4×4 with bits, except at the parity bits positions. Then apply parity bits accordingly.
2. Trade your square with one of your classmates, copy it but with one bit changed, and give the copy back.
3. Can you guess where your classmate changed the bit in your square?

Exercise: Large squares Hamming code

1. Guess the position of the parity bits with a square of size 32, and the columns and lines they cover.
2. Guess the position of the parity bits with a square of size 2^k, and the columns and lines they cover.

Exercise: Top left corner

One of the two has a mistake in the top left corner. Can you tell which one? Why?

$$\begin{matrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{matrix} \qquad \begin{matrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{matrix}$$

Haming Codes actually use one more bit than the ones described previously: a *parity bit* on the top left corner. This means that in addition to the four parity bits described previously, one more bit is used for redundancy:

$$\begin{matrix} \mathbf{X} & \mathrm{X} & \mathrm{X} & \cdot \\ \mathrm{X} & \cdot & \cdot & \cdot \\ \mathrm{X} & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{matrix}$$

When data is given, the four parity bits described previously are set first, and then the one on the top left corner last.

Exercise: Top left corner

Same question as previous exercise but with the new version of the Haming Code.

Exercise: New data, new errors

You receive the following data. Is it correct? Where are the error(s)?

$$\begin{matrix} 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{matrix}$$

Exercise: The limits of the code

1. From a given correctly encoded data, try to flip 4 different bits to reach some other correctly encoded data.
2. Is it possible, from a correctly encoded data, to reach another correctly encoded one by flipping 3 bits?
3. Find two different correctly encoded data d_1 and d_2 such that flipping 1 bit in d_1 and 3 bits in d_2 leads to the same data d_3.
   This means that the code can not distinguish between 1 or 3 errors. It is designed to make the correction assuming there is only 1, and is not correct when there are 3 or more errors.

3. Hypercubes, sphere packing, and making eccentric maths actually useful

Inspired by a comic on this topic [2].

We just saw that no matter how smart a code is, it can sometimes miss errors in messages that are corrupted enough. What a code can do, however, is ensure that if there are less than k errors in a message, they are always detected, or even corrected.

In this section we describe a generic way to build a code that can correct messages that have k errors or less, and to establish how efficient it is.

3.1. Hypercubes

To describe precisely what flipping many bits can do, one mathematical tool we can use is *hypercubes*.

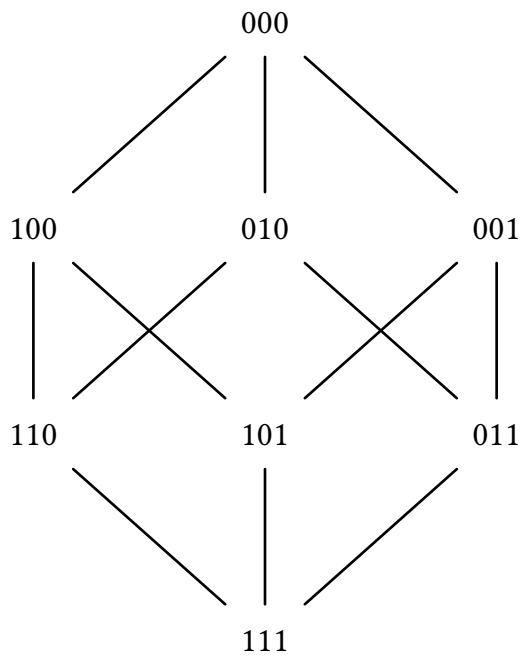
Definition: Hypercubes

A **hypercube** is a generalization of cubes to higher (and lower) dimensions. A hypercube of dimension 3 is a cube, a hypercube of dimension 2 is a square, but hypercubes can be defined for an arbitrarily large dimension.

As messages of n bits can be seen as elements of the set $\{0,1\}^n$, we will use a hypercube of dimension n to describe the set $\{0,1\}^n$.

- Vertices of the cube are elements of $\{0,1\}^n$,
- There is an edge between x and y if and only if x can be transformed to y by flipping exactly one bit.

With this representation, messages that can be produced by corrupting exactly k bits from a starting message m are exactly vertices that can be reached from m in the hypercube by using exactly k edges.



3.2. Sphere packing

Sphere packing is a mathematical problem about storing as many spheres as possibles in a given box. At the time it was first studied, in the early 17th, applications included the storage of cannonballs on a ship or, more simply, of oranges in crates.

Mathematicians, shopkeepers and pirates all found that the best way to do so is the *hexagonal close packing*: around one sphere, store 6 others on the same plane, and repeat in layers. Mathematicians, and in particular Kepler, conjectured that it is the best way to store them but couldn't prove for centuries that it indeed is[1].

During the process of trying to prove it, they redefined the problem to some more abstract ones, in particular of filling spaces of n dimensions with as many spheres as possible. At the time it was defined this alternative problem had no application.

3.3. Designing error correcting codes with hypercubes

Back to correcting codes now.

Designing an efficient code that can always correct a message if there ar less than k corrupted bits but with as few redundancy as possible can be achieved by solving a *sphere packing problem in a hypercube of n dimension*. How so?

Remember that the set of messages that can be produced by flipping k bits of m is exactly the set of vertices of the hypercube that can be reached from m with k edges. We say that these messages are *inside the sphere of radius k centered in m*.

[1] The Kepler conjecture has been proven on paper in 1998 [3], and with a proof assistant in 2014 [4].

Definition: Distance and sphere in hypercubes

The **distance** $d(x, y)$ between two vertices x, y in a hypercube is the least number of edges in a path from x to y. A **sphere** of radius k centered in x is the set of vertices of the hypercube at a distance k from x:

$$\text{sphere}(x, k) = \{y \mid d(x, y) = k\}$$

The inside of the sphere is the same but with an inequality in the definition. We also call it a **ball**:

$$\text{ball}(x, k) = \{y \mid d(x, y) \leq k\}$$

Examples

- This defines a distance in the mathematical sense:
  - Positivity: $\forall x \forall y, d(x, y) \geq 0$
  - Symmetry: $\forall x \forall y, d(x, y) = d(y, x)$
  - Separation: $\forall x \forall y, d(x, y) = 0 \Leftrightarrow x = y$
  - Triangular inequality: $\forall x \forall y \forall z, d(x, z) \leq d(x, y) + d(y, z)$
- In the hypercube of dimension 3, the ball of radius 1 centered in 000 is $\{000, 100, 010, 001\}$.
- In a hypercube of dimension n, a ball of radius n contains the whole hypercube.

To correct a message m that has k errors, one possibility is to ban from correct encodings all messages that are at a distance k or less from m in the hypercube, i.e. all messages of the ball of radius k centered in m, except its center.

With this encoding, if a message inside the ball is received, it necessarily was the message at the center of the ball when sent.

Example in 3D One possible design of a 3-bit long encoding that can correct up to 1 corrupted bit is:

- First possible message is 100. If any of the string in $\{100, 000, 010, 110\}$ is received, the original message was 100.
- Second possible message is 011. If any of the string in $\{011, 001, 101, 111\}$ is received, the original message was 011.

This method uses a lot of redundancy to make the correction possible. The problem of minimizing the redundancy... actually is a sphere packing problem in a n-dimension discrete space!

Designing a code on messages of n bits that can both correct up to k corruptions and encode as many different messages as possible
is the same as
solving a sphere packing problem in a hypercube of dimension n with sphere of radius k.

Bibliography

[1] G. Sanderson, “But what are Hamming codes? The origin of error correction.” [Online]. Available: https://www.youtube.com/watch?v=X8jsijhllIA

[2] T. Tao and Z. Weinersmith, “The Music of the Spheres.” [Online]. Available: https://www.smbc-comics.com/comic/spheres-part-1

[3] T. C. Hales, “The kepler conjecture,” *arXiv preprint math.MG/9811078*, 1998.

[4] T. Hales *et al.*, “A formal proof of the Kepler conjecture,” in *Forum of mathematics, Pi*, 2017, p. e2.

I hope you liked this course! I would like to have your anonymous opinion about it.

https://framaforms.org/avis-cours-1-a-5-informatique-arteq-2026-1791883142