Explain it like I'm 2: Hamming and SECDED

Sam Nolan

May 2, 2018

Hamming and SECDED are 2 very similar approaches to detecting errors in a string of bits.

In order to compare them to unchecked transmission or a simple parity, a ball and bucket metaphor can be used.

They do this through the use of parity bits. Which are redundant bits added to determine whether the rest of the bits are valid.

The bucket metaphor

Say that you are living in a small cottage on a mountain. You have a friend that lives slightly below you on this mountain, and you often need to convey simple messages such as:

- You left your car lights on
- You should come over!
- Your dog made his way into my yard again
- I won't be home today

However, the reception is really poor, so you can't text or call them to tell them this important information, and being very lazy, you can't be bothered to go down the mountain to talk to them.

So you devise a genius plan! You will set up a straight ramp that goes from your property to theirs. At the bottom of the ramp, they will place a buckets labelled with the messages above (and more if deemed fit). Side by side so that a ball that goes down the ramp must fall into one of the buckets. That way, you can drop the ball down the ramp and it will go into the bucket of the message that you want to convey.

This is what the bucket configuration looks like: "UUUU"

With no spacing in-between buckets at all. In this configuration, there is a "distance" of 1, meaning that the distance between each bucket is 1 bucket-length.

There's only one problem with this, although you are pretty good with rolling these balls into the correct buckets, you're not perfect. Sometimes the wrong message is conveyed as the ball strays from where you wanted it to go by one bucket or so.

As the buckets are side by side, there's no way for your friend to detect whether there was an error in transmission or not.

Back to bits

This configuration is similar to sending information among an unreliable medium with no error correction. If any of the bits change in a message, it is impossible to detect whether it was sent in error or it is a legitimate message.

The Hamming distance is the minimum distance between 2 valid codes. In this case, as any change to the data will still be valid, the Hamming distance would be 1. This Hamming distance corresponds to the separation of the buckets.

One error in a bit corresponds to the ball going right or left by one bucket-length.

Chapter 2 of our bucket metaphor (Parity)

In order to fix this, your friend could add spacing between the buckets, so that the buckets are further apart. The buckets would look like "U U U U". That way, if there was a single error in transmission (i.e. the ball moved one bucket-length right or left) The error can be detected. If one error in transmission was made, the ball would not fall into a bucket, it would fall onto the floor.

In this case, the distance between the buckets becomes 2, and there is an invalid message that is in between the buckets.

Parity bits

So how can be replicate this using our bits?

We can devise a very simple mechanism for detecting one bit errors using a single parity bit.

In math, parity means the state of being odd or even. For instance, the parity of 4 is even, and the parity of 5 is odd. From this point on however, when I say parity I will be referring to a parity bit

Take a look at the following string.

01101

By using what is called even parity bit, we can add a bit onto the end of the string in order to make the amount of 1's even.

011011

Now, if an error is made, either a 1 will be removed or added. This means that if the amount of 1's was before even, with 1 error it will be odd.

010011

We can tell that these bits is invalid. As there is an odd number of 1s in the string.

Keep in mind that using this configuration, every time a bit is changed, the message changes from valid to invalid or invalid to valid. For instance, 1 change makes the string invalid (As above) and 2 changes makes the string valid, 3 makes it invalid and 4 makes it valid again and so on. This is because you are always either adding a 1 (by changing a 0 to a 1) or removing a 1 (by changing a 1 to a 0). That operation will always change the parity (evenness/oddness).

If the amount of errors that have been made is even, then it will pass, if it is odd, then it will fail.

This is similar to our situation with the buckets, every time an error is made, the message goes from becoming a valid to an invalid message or goes from an invalid to a valid one.

Remember, the Hamming distance is defined as the minimum distance between 2 valid states. The simple parity has a Hamming distance of 2.

Problems with Parity bits

If we go back to the metaphor, if an error is made and detected, your friend needs to hike up the mountain to tell you that an error was made and ask for the message again. This is analogous to asking for the message to be sent again in computing, and when the message is coming from the other side of the world, this can take a lot of time (this time in computing would actually be measured in milliseconds, but computers are rather impatient).

The other problem is that any time there is an even amount of errors made, it is deemed valid. Although more than 1 error is quite rare, it's still possible, and should be considered in systems that have a low fault tolerance.

Funnelling and Hamming code

Your friend, being very smart and not wanting to hike up the mountain to tell you of your errors, decided he would come up with an alternative. He would put ramps so that if a ball falls 1 bucket length away from any particular bucket, it would hit the ramp and fall into the bucket instead.

The configuration of the buckets now looks like this:

```
\ /\ /\ /\ /
 U  U  U  U
```

The distance between the buckets is now 3, but it also has the added bonus that if there is only 1 error in transmission, it can be corrected.

Hamming code

Can we do this with bits? These ramps require something clever.

As we have seen before, we can use parity bits to detect whether an error has been made.

Let's get a bit smarter, and make it so our parity bits cover different aspects of the string. We could have more than 1 parity bit in order to better find where the error is.

Let's invent a scheme. Say that the first 7 bits of a byte is legitimate data, and the last bit is an even parity for the byte. Meaning that if any particular byte of data has an odd number of 1s, it is in error.

Take a look at the following string, which byte is in error using this scheme?

11010111 10000101 11100100 10110001

Got it? It's the second one, it has 3 1 bits, there must have been some error in transmission for that particular byte.

As you can see, by using more parity bits, we can narrow where in the message an error was found. But we can do better than that, we can use this concept and be able to identify which individual bit is in error using the Hamming code.

We can get information about where the faulty bit is depending on if the parity check failed or not. A parity check either fails, or it does not. Like a bit itself, it only contains 2 states. We want it so that a certain combination of fails and passes gives us the point where the bit is in error.

So how many positions can we specify with these pass/check combinations? Well say you have 2 parity bits and therefore 2 parity checks. You can have the following combinations:

- Pass Pass
- Pass Fail
- Fail Pass
- Fail Fail

Looks sort of like bits yes? That means that with p parity bits (and therefore p parity checks), you can express 2^p different positions that might be in error.

So how many bits do we need anyway?

How many parity bits do we need to specify a position in a bit string?

Well, if we have 3 bits, there are 8 (2^3) possible states that they can be in. This means that 3 bits could specify 8 different locations in a bit string. Say for instance, 001 could mean be the the bit at index 1 and 010 could be the bit at index 2, etc. We'll reserve 000 for later.

This means that given p bits, you can specify 2^p positions in a bit string. Meaning to be able to express all possible positions in a message of size m:

$$2^p \geq m$$

Now, the parity bits expressing the location of the error actually need to be sent with the message. There could be an error in a parity bit. This means that the 2^p also needs cover the parity bits. This gives us:

$$2^p \geq m + p$$

Finally, we are forgetting something. There is not always an error in transmission! We need a place that our bits can point to to indicate that there was no error. Let's add this phantom position to our equation.

$$2^p \geq m + p + 1$$

So there you have it, the amount of parity bits need to satisfy this equation for there to be enough, otherwise they will not be able to express every position in the string plus the fact that there could be no error!

This is the theoretical minimum possible number of parity bits needed in order to correct a bit that is in error. This theoretical minimum is actually achieved through the use of a Hamming code! Hamming was a smart guy.

Placing these bits

So now that we have worked out the amount of parity bits we will need, we need to work out a way that we can position them intuitively.

One way we can do this, we can map a pass to mean 0 and a fail to mean 1. This means that if we have 2 parity bits, the results can be represented as

- 00
- 01
- 10
- 11

These can be then interpreted as decimal digits 0,1,2,3 to mean positions.

But hold on a second, if all the parity checks pass, doesn't it mean that there is no error? You would be right. Because of this, we'll reserve 0 for when we don't get an error, meaning that 01 will indicate an error in the first bit (Least Significant bit) and 10 will indicate an error in the 2nd bit and so on.

As a side note an easy way to remember which bit is the LSB and which is the MSB is that it is exactly the same as it is in the decimal system. The number on the far right of number is the least significant and the number on the far left is the most.

So, lets number our bits off:

111	110	101	100	011	010	001
7	6	5	4	3	2	1

We have 3 bits of data that give us a position. Which means 3 parity check and therefore 3 parity bits. So this piece of data has 3 parity bits and the other 4 can be data.

Now all we need to do is choose the parity check positions.

Let's take a look at the first bit. This first bit will be indicated only if the first parity check fails (As shown by 001).

If there is an error in this bit, only the first parity check will fail, meaning that only parity check 1 has this bit in it's check.

This logic can extend to all the other bits. For instance, for bit 3, if it is error, parity check 1 and parity check 2 will fail, meaning bit 3 is covered by parity check 1 and 2 but not 3.

Because bit 1 only has 1 parity check, and the parity bit always needs to be inside the parity check, this bit has to be the first parity bit.

The same logic applies for 2 and 4, as there is only 1 parity check that will fail in those bits, so they must respectively be parity bits 2 and 3

D4 D3 D2 P3 D1 P2 P1

In the same way that in the decimal system, only powers of 10 have one 1 and the rest 0s, in binary, only powers of 2 have one 1 and rest 0s, so the parity bits will be in those positions among the string.

Now all we need to do is fill in which bits are covered by which numbers:

111	110	101	100	011	010	001
7	6	5	4	3	2	1
321	32	3 1	3	21	2	1

And there we have it! A hamming code!

Features of a Hamming code

Like our buckets and ramps, the hamming code can correct an error when there is only one bit that is different than it should. It has a hamming distance of 3, meaning it takes a minimum of 3 changes to make it from 1 valid bit string to another valid bit string.

However, if we make 2 changes, as of our bucket metaphor, it will correct itself to another bucket, and therefore giving the wrong message. Therefore, Hamming codes work to correct only 1 bit errors.

SECDED

Now, this is much more efficient than your friend hiking up the mountain to see you. However, on rare occasions, you do make 2 errors and the message gets converted to something you didn't want to convey. In light of this problem, your friend does exactly what he did before, he is going to add a bit more spacing between the buckets like so:

```
\ / \ / \ / \ /
 U   U   U   U
```

Now, the buckets have a distance of 4. If 2 errors are made, the ball will fall in between the ramps and not go into a bucket.

How can we do this with our bits? Exactly the same way we separated the buckets earlier, with an extra parity

Hamming and the extra parity

We know from before that if we have a parity bit that covers the entire bit string, if there is an even number of errors, the check will pass and if we have an odd number of errors, the check will fail.

So if we were to add a parity bit at position 0 of the string, we could tell if there has been an odd number or an even number of errors.

Let's check out the different situations and what that infers for us:

Hamming passed and the extra parity passed: What are we waiting for? There's no errors! Accept the string

Hamming passed and the extra parity failed: Well, Hamming does not cover the extra parity bit, so therefore this tells us that the only place that could be in error is that parity bit. Funnily enough, when Hamming gets all passes, it points to position 0 to be in error, which would be the extra parity! That being said, that logic can't be made when there are no errors at all.

Hamming failed and the extra parity failed: Well, this tells us that there is an odd number of errors that is within the hamming code. This situation is exactly the same as the hamming code, and we should run Hamming code logic in order to find the error and correct it

Hamming failed and the extra parity passed: In this case, there is an even number of errors, and hamming is telling us that there is an error somewhere in the bit string, therefore it's likely that there was 2 errors, we should ask for the bit string again.

This extra parity allows us to detect 2 errors and correct 1. However, SECDED cannot correct 2 errors, and if in the nearly impossible situation where there are 3 errors, Situation 3 is likely to be run and correct it to the wrong message.

This has a Hamming distance of 4, meaning that it takes a minimum of 4 errors to get from one valid bit string to another.

Limitations of the metaphor

Like scientific metaphors, this bucket metaphor is only a model, and can be used to effectively represent what Hamming, parity, and SECDED can do. That being said, it is only a model, and Hamming and SECDED work a little differently.

What you can take from the model is that if you are in one state and make an error, you will move to a state that is either right or left of it. These movements are not of equal probability mind you. If you are using SECDED and have a valid bit string and make one error, then the "ball" will move onto a ramp and it can be corrected. However, when the second error is made, it will only go back into the bucket if the second error cancels out the first (e.g. They both flip the same bit and result in a valid message being received). On the second error, it is for more likely that it will be another bit that is tampered with, and therefore dropping into an invalid uncorrectable state.

With our 1 dimensional bucket arrangement, it seems that there is only 2 different valid states that 2 errors could bring you to. This is not the case, with 2 errors, the ball can make it to a large number of different buckets.

Another thing is that it seems that adding 1 extra parity bit corresponds to moving the buckets further away from each other by 1 bucket-length. This is not necessarily true. Simply tacking a parity bit onto the end of a message that already has a parity bit that used in the same way doesn't help anything. It's just that adding a parity check onto unchecked and hamming both happen to have that effect.

Hope that made sense! If it didn't, I would love some feedback. My email is s3723315@student.rmit.edu.au.

If there is anything in this course of other courses in computer science that need a decent explanation, just ask! Writing these helps give me a good understanding of the course material.