CPS311 Lecture: Error Detecting and Correcting Codes

Revised September 4, 2019

Materials: Hamming Code Projectable and posted on web

I. Introduction

A. With any technology, there is a danger that information will be corrupted due to physical imperfections in storage media or electronic "noise". Since an undetected error of even 1 bit can be catastrophic, we wish to take measures to detect such errors as might occur. As a minimum, we seek error detection; but error correction is even better.

B. The same principles are often used when transmitting data from one place to another over a network - the other place where data is especially vulnerable to corruption (even more so.)

II. Simple Error detecting/correcting codes

A. All error detection/correction schemes depending on using more bits to store information than what one actually needs.

Of course, the simplest such scheme is parity. With each data item, we store an extra bit called the parity bit.

1. One convention, called odd parity, specifies that the parity bit be set so that the total number of 1 bits (including the parity bit) is odd Examples:

```
Data item       01010101
Item + parity  101010101

Data item       10000011
Item + parity  010000011
```

2. An alternate convention, called even parity, sets the parity bit so that the total number of 1 bits (including the parity bit) is even. The choice of which convention to use depends on what sort of catastrophic error is judged most likely to occur. For example, if a complete system failure would normally result in all data being converted to 0's, then odd parity would report something wrong but even would not. On the other hand, if system failure would turn all data to 1's (and the total length of data plus parity is odd) then even parity would be preferred,

3. To check the correctness of an item, the receiver simply counts the number of 1 bits and ensures that they are odd or even, as the case may be. (Of course, both the originator and the receiver of the item must use the same convention as to odd/even parity.)

4. Parity is what we call a 1-bit error detecting code

a. It can tell us that some bit is corrupt - but not which one is corrupt. When a parity error occurs, any data bit might be in error, or the data might itself be fine and the parity bit corrupt!

b. It can be fooled by a multiple error. If 2 bits are corrupted, a parity test will tell us everything is ok.

B. More sophisticated schemes provide for not only detecting but also correcting errors, and may be able to detect multi-bit errors. One rather simple example is a scheme once used on magnetic tape.

1. Data on tape is written in blocks, composed of frames. Each frame typically has 8 data bits plus a frame parity bit. Each block has a an extra frame called the block checksum, each of whose bits is a parity check on one row position throughout the length of the block - e.g.

```
d d d d d d d d d d l <- this bit checks all the d's to its left
d d d d d d d d d d l
d d d d d d d d d d l
d d d d d d d d d d l
d d d d d d d d d d l   d = data
d d d d d d d d d d l   f = frame parity
d d d d d d d d d d l   l = longitudinal parity
d d d d d d d d d d l
f f f f f f f f f f l <- this bit checks both all the l's above
^                        it and all the d's to its left
|
This bit checks all d's above it
```

2. Consider what would happen if there were a single bit in error in the block. When the user reconstructs the f's and l's, he would find that one f and one l are incorrect. Together, these two would isolate the one bit in error, which could be corrected by inverting it. Thus, we have 1 bit error correction.

3. Now suppose two bits were corrupted.

a. If they were in the same frame, we would have 2 l's in error.

b. If they were in the same row, we would have 2 f's in error.

c. If they were in different frames and different rows, we would have two errors each in f's and l's.

d. In any case, would be able to detect a 2 bit error (though we could not correct it.)

4. Finally, consider the effect of a 3 bit error. In most cases, we would get multiple error indications in both f's and l's - letting us know that the data is corrupt. However, there is one pathological case to be aware of. Let e be the bits in error:

```
        e                       e       <- l ok here

        e                               <- l signals error here

        ^                       ^
        |                       |
        f ok here               f signals error here
```

Unfortunately, this would look like a (correctable) 1 bit error. But we would correct the wrong bit, thus ending up with a 4 bit error!

5. For this reason, this encoding scheme is called a one-bit error correcting, two-bit error detecting scheme. Its usefulness is based on the assumption that errors involving three or more bits are so improbable as to be allow us to assume such errors won't occur.

   a. Suppose that the probability of a bit being corrupted is 10^{-9}. (One per billion).

      i. Such an error can be corrected and processing can proceed. If we process a billion bits per second, such a situation will arise, on the average, once per second.

      ii. The probability of a two bit error is 10^{-18}. Such an error can be detected - and some alternate path can be pursued to get the correct value. (E.g. recomputing it from an old value and a log of transactions, or retransmitting if we are dealing with a message over a network.) If we process a billion bits per second, this will occur, on the average, once every 317 years.

      iii. The probability of a three bit error is 10^{-27}. Such an error might be detected (many would be), but could escape detection. If we process a billion bits/second, this amounts to no more than one undetected error in 317 billion years - probably fairly safe!

   b. Note that neither this scheme, nor any other, can GUARANTEE that an undetected error won't occur - it can only make such an error improbable enough to make us willing to trust the system.

   c. The fact that error-correcting and detecting schemes are only probably correct means that, in some sense, computer-processed data is never ABSOLUTELY GUARANTEED to be accurate.

III. Hamming Codes

A. Now we consider a scheme that can be used for error detection/correction in a single word of data. The scheme is called a Hamming code. The example we will develop is for a word length of 11 bits - but the idea could be extended to any word size.

   SHOW PROJECTABLE PAGE 1 (Use pdf; set to View Slideshow; use arrows on bottom)

1. Let there be n data bits (numbered d_0 .. d_{n-1})

2. We will add m correcting bits, where m is the smallest integer such that 2^m >= (n + m + 1). (Example: 11 data bits - let m = 4 - 2^4 = 16 = (11 + 4 + 1). We number these bits c_0 .. c_{m-1}

3. We will logically, though not necessarily physically, intersperse the correcting bits so that their positions in the overall word are powers of 2. (We number bits in the overall word 1 .. m + n).

Ex:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
	d_{10}	d_9	d_8	d_7	d_6	d_5	d_4	c_3	d_3	d_2	d_1	c_2	d_0	c_1	c_0

4. Let each c_i be a parity check on the remaining bits in logical positions which contain 2^i in their binary representation - e.g.

   c_0 checks all bits in odd numbered positions
   c_1 checks bits in positions 3, 6, 7, 10, 11, 14, 15
   c_2 checks bits in positions 5, 6, 7, 12, 13, 14, 15
   c_3 checks bits in positions 9 .. 15

   (Observe: no c checks any other c - only d's. Thus, the c's can be computed knowing only the d's.)

5. To store a word, add in the necessary correcting bits and store the combination. To check a word retrieved from storage:

   a. Extract the d bits.

   b. Compute what the c bits should be.

   c. XOR the computed c bits with the actual c bits.

      i. If the result is all 0's, the stored word is ok.
      ii. If the result is non-0, treat the XOR result as the binary representation of an integer. This is the number of the bit position where the error occurred (assuming a 1 bit error), where bit positions must be numbered starting at 1, not 0.

   Example (using odd parity): data originally:

   10101010101 => 1010101_010_1__ w/slots for c's   PROJECTABLE 2

   Correcting bits: $c_0 = 0$
   $c_1 = 1$
   $c_2 = 0$
   $c_3 = 1$

```
Stored word is 101010110100110                    PROJECTABLE 3,4
                      -   - --
```

Assume data bit 0 (slot 3 if bit slots are numbered starting at 1) is corrupted in storage, so that word as read is 101010110100010

The receiver would extract the data bits 10101010100, and would compute the c bits on this basis to be

$c_0 = 1$ PROJECTABLE 5,6

$c_1 = 0$

$c_2 = 0$

$c_3 = 1$

XORing with the c bits extracted from the stored data yields 0011 - indicating that the error is in slot 3 of the stored data, as desired.

PROJECTABLE 7

B. The code we have presented gives 1 bit error correction, but could be fooled by a 2 bit error. (We would, in fact, create a 3-bit error by "correcting" the wrong bit!)

C. To add 2-bit error detection, we simply add a conventional parity bit.

1. In the absence of error, the parity bit will indicate no error, and the correcting bits will show no error (exact match between computed and stored values.)

2. A 1-bit error can always be corrected.

   a. If it is in the data or correcting bits, it will cause both the parity bit and the correcting bits to report an error, and the correcting bits can be used to correct it.

   b. If it is in the parity bit, it will cause the parity bit to report an error, but the correcting bits will report no error. In this case, we know that the parity bit is wrong.

3. A 2-bit error will always be detected but cannot be corrected.

   a. If two data and/or correcting bits are wrong, this will cause the parity bit to NOT report an error, but the correcting bits WILL report an error - this is taken as an indication of a double error that we can detect but not fix.

   (There is no way to corrupt exactly two bits in such a way as to cause the correcting bits to report no error, because each data bit affects at least two correcting bits, and no two data bits affect the same set of correcting bits.)

b. If one data or correcting bit plus the parity bit are corrupted, once again the parity bit will NOT report an error, but the correcting bits WILL report an error (in fact, the right one). However, since we can’t distinguish this case from the previous one, we will simply have to recognize a double error and let it go at that.

IV. More Sophisticated Schemes

A. There are, of course, other error detecting and correcting schemes. The most widely used scheme is based on cylic-redundancy polynomials, which we can't go into here. Such schemes can be made to not only yield 1-bit error correction, 2-bit error detection, but also can detect longer error bursts (successive corrupted bits).

B. A final general observation on the cost of error detection/correction in terms of added bits. Some terminology:

1. Any error detection/correction scheme operates by creating an extended encoding for the data in such a way that certain bit patterns are legal and others are not. (For example, odd parity creates an encoding in which words having an odd number of 1's are legal and those having an even number are not.)

2. For any such encoding, we can define the distance between two legal words as the number of bits that would need to change to go from one to the other. For example, the distance from 101010 to 101100 is 2.

3. For any encoding scheme, we can define the minimum distance to be the smallest distance between any two legal words. For example, for simple parity the minimum distance is two.

4. Now observe:

a. For a 1-bit error detecting code, a minimum distance of 2 is necessary.

b. For a 1-bit error correcting code, a minimum distance of 3 is necessary. This guarantees that for any erroneous word there will be at most 1 word that has a distance of 1 from it. We take that correct word to be the original data that was corrupted.

c. For 1-bit error correction plus 2 bit error detection, we need a minimum distance of 4.

d. In general, for n-bit error correction we need a minimum distance of 1 + 2*n, and for m-bit error detection (m >= n) we need to increase the distance by m-n.

5. We can reduce the sheer number of added bits by generating the detecting/correcting bits for longer blocks of data.

Example: assume we want 1-bit error correction 2-bit error detection. we need a minimum distance of 4.

If we have a 128 byte message, we could achieve this by adding 5 error-detecting/correcting bits to each byte, for a total of 128 * (8 + 5) = 1664 bits. (2^4 >= 8+4+1; add 1 for parity)

Or, we could treat the message as 32 "words" of 4 bytes each, appending 7 error-detecting/correcting bits to each word, for a total of 32 * (32 + 7) = 1248 bits. (2^6 >= 32+6+1; add 1 for parity)

Or, we could treat the message as 16 "words" of 8 bytes each, appending 8 error-detecting/correcting bits to each word, for a total of 16 * (64 + 8) = 1152 bits. (2^7 >= 64+7+1; add 1 for parity)

Or, we could treat the entire message a single long "word", appending 12 error-detecting/correcting bits to it, for a total of 1036 bits! (2^{11} >= 1024+11+1; add 1 for parity)