Hamming code

From Wikipedia, the free encyclopedia

In telecommunication, a **Hamming code** is a linear error-correcting code named after its inventor, Richard Hamming. Hamming codes can detect and correct single-bit errors, and can detect (but not correct) double-bit errors. In contrast, the simple parity code cannot detect errors where two bits are transposed, nor can it correct the errors it can find.

Contents

[hide]

History

Hamming worked at Bell Labs in the 1940s on the Bell Model V computer, an electromechanical relay-based machine with cycle times in seconds. Input was fed in on punch cards, which would invariably have read errors. During weekdays, special code would find errors and flash lights so the operators could correct the problem. During after-hours periods and on weekends, when there were no operators, the machine simply moved on to the next job.

Hamming worked on weekends, and grew increasingly frustrated with having to restart his programs from scratch due to the unreliability of the card reader. Over the next few years he worked on the problem of error-correction, developing an increasingly powerful array of algorithms. In 1950 he published what is now known as Hamming Code, which remains in use in some applications today.

Codes predating Hamming

A number of simple error-detecting codes were used before Hamming codes, but none were nearly as effective as Hamming codes in the same overhead of space.

Parity

Parity adds a single bit that indicates whether the number of 1 bits in the preceding data was even or odd. If a single bit is changed in transmission, the message will change parity and the error can be detected at this point. (Note that the bit that changed may have been the parity bit itself!) The most common convention is that a parity value of **1** indicates that there is an **odd** number of ones in the data, and a parity value of **0** indicates that there is an **even** number of ones in the data. In other words: The data and the parity bit **together** should contain an even number of 1s.

Parity checking is not very robust, since if the number of bits changed is even, the check bit will be valid and the error will not be detected. Moreover, parity does not indicate which bit contained the error, even when it can detect it. The data must be discarded entirely, and re-transmitted from scratch. On a noisy transmission medium a successful transmission could take a long time, or even never occur. While parity checking is not very good, it uses only a single bit, resulting in the least overhead, and does allow for the restoration of a missing bit, when which bit is missing is known.

Two-out-of-five code

In the 1940s Bell used a slightly more sophisticated code known as the two-out-of-five code. This code ensured that every block of five bits (known as a *5-block*) had exactly two 1s. The computer could tell if there was an error if in its input there were not exactly two 1s in each block. Two-of-five was still only able to detect single bits; if one bit flipped to a 1 and another to a 0 in the same block, the two-of-five rule remained true and the error would go undiscovered.

Repetition

Another code in use at the time repeated every data bit several times in order to ensure that it got through. For instance, if the data bit to be sent was a 1, an n=3 *repetition code* would send "111". If the three bits received were not identical, an error occurred. If the channel is clean enough, most of the time only one bit will change in each triple. Therefore, 001, 010, and 100 each correspond to a 0 bit, while 110, 101, and 011 correspond to a 1 bit, as though the bits counted as "votes" towards what the original bit was. A code with this ability to reconstruct the original message in the presence of errors is known as an *error-correcting* code.

Such codes cannot correctly repair all errors, however. In our example, if the channel flipped two bits and the receiver got "001", the system would detect the error, but conclude that the original bit was 0, which is incorrect. If we increase the number of times we duplicate each bit to four, we can detect all two-bit errors but can't correct them (the votes "tie"); at five, we can correct all two-bit errors, but not all three-bit errors.

Moreover, the repetition code is extremely inefficient, reducing throughput by three times in our original case, and the efficiency drops drastically as we increase the number of times each bit is duplicated in order to detect and correct more errors.

Hamming codes

If more error-correcting bits are included with a message, and if those bits can be arranged such that different incorrect bits produce different error results, then bad bits could be identified. In a 7-bit message, there are seven possible single bit errors, so three error control bits could potentially specify not only that an error occurred but also which bit caused the error.

Hamming studied the existing coding schemes, including two-of-five, and generalized their concepts. To start with he developed a nomenclature to describe the system, including the number of data bits and error-correction bits in a block. For instance, parity includes a single bit for any data word, so assuming ASCII words with 7-bits, Hamming described this as an *(8,7)* code, with eight bits in total, of which 7 are data. The repetition example would be *(3,1)*, following the same logic. The *information rate* is the second number divided by the first, for our repetition example, 1/3.

Hamming also noticed the problems with flipping two or more bits, and described this as the "distance" (it is now called the *Hamming distance*, after him). Parity has a distance of 2, as any two bit flips will be invisible. The (3,1) repetition has a distance of 3, as three bits need to be flipped in the same triple to obtain another code word with no visible errors. A (4,1) repetition (each bit is repeated four times) has a distance of 4, so flipping two bits in the same group will no longer go undiscovered.

Hamming was interested in two problems at once; increasing the distance as much as possible, while at the same time increasing the information rate as much as possible. During the 1940s he developed several encoding schemes that were dramatic improvements on existing codes. The key to all of his systems was to have the parity bits overlap, such that they managed to check each other as well as the data.

The algorithm for use of parity bits for the *general* 'Hamming code' is simple:

1. All bit positions that are powers of two are used as parity bits. (positions 1, 2, 4, 8, 16, 32, 64, etc.)
2. All other bit positions are for the data to be encoded. (positions 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 17, etc.)
3. Each parity bit calculates the parity for some of the bits in the code word. The position of the parity bit determines the sequence of bits that it alternately checks and skips.
    - Position 1 (n=1): skip 0 bit (0=n-1), check 1 bit (n), skip 1 bit (n), check 1 bit (n), skip 1 bit (n), etc.
    - Position 2 (n=2): skip 1 bit (1=n-1), check 2 bits (n), skip 2 bits (n), check 2 bits (n), skip 2 bits (n), etc.
    - Position 4 (n=4): skip 3 bits (3=n-1), check 4 bits (n), skip 4 bits (n), check 4 bits (n), skip 4 bits (n), etc.
    - Position 8 (n=8): skip 7 bits (7=n-1), check 8 bits (n), skip 8 bits (n), check 8 bits (n), skip 8 bits (n), etc.
    - Position 16 (n=16): skip 15 bits (15=n-1), check 16 bits (n), skip 16 bits (n), check 16 bits (n), skip 16 bits (n), etc.
    - Position 32 (n=32): skip 31 bits (31=n-1), check 32 bits (n), skip 32 bits (n), check 32 bits (n), skip 32 bits (n), etc.
    - General rule for position *n*: skip *n*-1 bits, check *n* bits, skip *n* bits, check *n* bits...
    - And so on.

In other words, the parity bit at position 2^k checks bits in positions having bit k set in their binary representation. Conversely, for instance, bit 13, i.e. $1101_{(2)}$, is checked by bits $1000_{(2)} = 8$, $0100_{(2)}=4$ and $0001_{(2)} = 1$.

Example using the (11,7) Hamming code

Consider the 7-bit data word "0110101". To demonstrate how Hamming codes are calculated and used to detect an error, see the tables below. They use **d** to signify data bits and **p** to signify parity bits.

Firstly the data bits are inserted into their appropriate positions and the parity bits calculated in each case using *even* parity.

	p_1	**p_2**	**d_1**	**p_3**	**d_2**	**d_3**	**d_4**	**p_4**	**d_5**	**d_6**	**d_7**
Data word (without parity):			**0**		**1**	**1**	**0**		**1**	**0**	**1**
p_1	**1**		0		1		0		1		1
p_2		**0**	0			1	0			0	1
p_3				**0**	1	1	0				
p_4								**0**	1	0	1
Data word (with parity):	**1**	**0**	0	**0**	1	1	0	**0**	1	0	1

Calculation of Hamming code parity bits

The new data word (with parity bits) is now "10001100101". We now assume the final bit gets corrupted and turned from 1 to 0. Our new data word is "10001100100"; and this time when we analyse how the Hamming codes were created we flag each parity bit as 1 when the even parity check fails.

	p_1	**p_2**	**d_1**	**p_3**	**d_2**	**d_3**	**d_4**	**p_4**	**d_5**	**d_6**	**d_7**	**Parity check**	**Parity bit**
Received data word:	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**0**		
p_1	**1**		0		1		0		1		0	**Fail**	**1**
p_2		**0**	0			1	0			0	0	**Fail**	**1**
p_3				**0**	1	1	0					Pass	0
p_4								**0**	1	0	0	**Fail**	**1**

Checking of parity bits (switched bit highlighted)

The final step is to evaluate the value of the parity bits (remembering the bit with lowest index is the least significant bit, i.e., it goes furthest to the right). The integer value of the parity bits is 11, signifying that the 11th bit in the data word (including parity bits) is wrong and needs to be flipped.

	p_4	**p_3**	**p_2**	**p_1**	
Binary	1	0	1	1	
Decimal	**8**		**2**	**1**	**= 11**

Flipping the 11th bit gives changes 1000110010**0** back into 1000110010**1**. Removing the Hamming codes gives the original data word of 0110101.

Note that as parity bits do not check each other, if a single parity bit check fails and all others succeed, then it is the parity bit in question that is wrong and not any bit it checks.

Finally, suppose two bits change, at positions *x* and *y*. If *x* and *y* have the same bit at the 2^k position in their binary representations, then the parity bit corresponding to that position checks them both, and so will remain the same. However, *some* parity bit must be altered, because *x* != *y*, and so some two corresponding bits differ in *x* and *y*. Thus, the Hamming code detects all two bit errors -- however, it cannot distinguish them from 1-bit errors.

Hamming code (7,4)

Today, **Hamming code** really refers to a specific (7,4) code Hamming introduced in 1950. Hamming Code adds three additional check bits to every four data bits of the message. Hamming's (7,4) algorithm can correct any single-bit error, or detect all single-bit and two-bit errors. This means that for transmission medium situations where burst errors do not occur, Hamming's (7,4) code is effective (as the medium would have to be extremely noisy for 2 out of 7 bits to be flipped).

Hamming matrices

Hamming codes work through extending the idea of parity by multiplying matrices called **Hamming matrices**. For the Hamming (7,4) code, we use two closely related matrices, the *code generator matrix* **G** and the *parity-check matrix* **H**:

$$\mathbf{G} := \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix}$$

and

$$\mathbf{H} := \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}.$$

The first four rows of **G** represent the 4 x 4 identity matrix $\mathbf{I}_4$, while the last three rows consist of a 4 x 3 mapping from the 4 source bits to the 3 parity bits. The column vectors in **G** form the basis of the kernel of **H**. The identity matrix passes the data vector on in the multiplication step. Unlike the explanation above, the data bits are in the first 4 positions while the parity bits are in the last 3 positions. Although these matrices are somewhat different from the actual Hamming matrices, they convey the essential details of the Hamming code in a form that is easier to understand.

Similarly, the last three columns of **H** represent the 3 x 3 identity matrix $\mathbf{I}_3$, while the first four columns consist of the identical 4 x 3 mapping between the source data bits and the parity checks.

We use a block of four payload data bits (hence the 4 in the name), and accrue another three redundant data bits (hence the 7 in the name as 4+3=7). To send the data, we consider the block of data bits we wish to send as a

vector, for example, for "1011", the vector is

$$\mathbf{p} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

Channel coding

Suppose we want to transmit this data over a noisy communications channel. We take the product of **G** and **p**, with entries modulo 2, to determine the transmitted codeword **x**:

$$\mathbf{Gp} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \mathbf{x}$$

Parity check

If no error occurs during transmission, then the received codeword **r** is identical to the transmitted codeword **x**:

$$\mathbf{r} = \mathbf{x}$$

The receiver multiplies **H** and **r** to obtain the *syndrome* vector **z**, which indicates whether an error has occurred, and if so, for which codeword bit. Performing this multiplication (again, entries modulo 2):

$$\mathbf{Hr} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} = \mathbf{z}$$

Since the syndrome **z** is the zero vector, the receiver can conclude that no error has occurred. This conclusion is based on the observation that when the data vector is multiplied by **G**, a change of basis occurs into a vector subspace that is the kernel of **H**. As long as nothing happens during transmission, **r** will remain in the kernel of **H** and the multiplication will yield the zero vector.

Error correction

Otherwise, suppose a single bit error has occurred. Mathematically, we can write

$$\mathbf{r} = \mathbf{x} + \mathbf{e}_i$$

modulo 2, where $\mathbf{e}_i$ is the *i*th unit vector, that is, a zero vector with a 1 in the *i*th place, counting from 1. Thus the above expression signifies a single bit error in the *i*th place.

Now, if we multiply this vector by **H**:

$$\mathbf{Hr} = \mathbf{H}(\mathbf{x} + \mathbf{e}_i) = \mathbf{Hx} + \mathbf{He}_i$$

Since **x** is the transmitted data, it is without error, and as a result, the product of **H** and **x** is zero. Thus

$$\mathbf{Hx} + \mathbf{He}_i = 0 + \mathbf{He}_i = \mathbf{He}_i$$

Now, the product of **H** with the *i*th standard basis vector picks out that column of **H**, we know the error occurs in the place where this column of **H** occurs. As we have constructed **H** in a particular way, we can interpret this column as a binary number -- for example, (1, 0, 1) is a column of **H** and thus corresponds to the 5th place, and thus know where the error has occurred and can correct it.

For example, suppose we have

$$\mathbf{r} = \mathbf{x} + \mathbf{e}_2 = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

Now,

$$\mathbf{Hr} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = \mathbf{z}$$

which corresponds to the second column of **H**. Thus, an error has been detected in position 2, and can be corrected.

It is not difficult to show that only single bit errors can be corrected using this scheme. Alternatively, Hamming codes can be used to detect single and double bit errors, by merely noting that the product of **H** is nonzero whenever errors have occurred. However, the Hamming (7,4) and similar Hamming codes cannot distinguish between single-bit errors and two-bit errors.

Hamming codes with additional parity

Hamming codes can be used together with an extra parity bit. The extra parity bit applies to all bits after the Hamming code check bits have been added. This increases the Hamming distance between valid codes from 3 to 4. Then all single-bit, two-bit and three-bit errors can be detected[1]. Also, two-bit errors can be distinguished from single-bit and three-bit errors. So single-bit errors can be corrected.

Two-bit errors can at least be recognized as uncorrectable: When using correction, if a parity error is detected and the Hamming code indicates that there is an error, this error can be corrected. However, if a parity error is not detected but the Hamming code indicates that there is an error, this is assumed to have been due to a two-bit error, which is detected but cannot be corrected.

See also

- Hamming distance
- Golay code
- Reed-Muller code
- Reed-Solomon code
- Turbo code
- Low-density_parity-check_code

References

- David J.C. MacKay (2003) Information theory, inference and learning algorithms, CUP, ISBN 0521642981, (also available online)

PDC Live

Hamming Code

Several Teletext packets contain information protected by *hamming*. This is a method for encoding data such that errors in reception can be detected, and if the error is sufficiently small, corrected. Teletext uses two hamming codes.

The simpler and more robust version encodes 4 bits of data in one 8-bit byte. This code is used extensively to protect the fields that have meaning to the Teletext system itself. The more efficient code encodes 18 bits of data in three 8-bit bytes. It is used for to protect some of the data fields inside Teletext packets. Both codes are designed such that any one-bit error can be corrected, and any 2-bit error can be detected.

8/4 Hamming

The 8/4 code is quite simple; if the input nibble is the 4-bit value:

```
b3, b2, b1, b0
```

with `b3` being the most significant bit, then the output hammed byte is the 8-bit value:

```
b3, b3^b2^b1, b2, !b2^b1^b0, b1, !b3^b1^b0, b0, !b3^b2^b0
```

where ^ represents bitwise exclusive-or and ! is bitwise not.

This code has the property that every value is four bits different from all other such values. A one-bit error is therefore unambiguously correctable, being only one bit away from a valid code. A two-bit error is detectable but not correctable, being equidistant between two valid codes.

Here is the table of all 16 encoded nibbles:

```
Data nibble     Hammed byte
0 = 0 0 0 0     15 = 0 0 0 1 0 1 0 1
1 = 0 0 0 1     02 = 0 0 0 0 0 0 1 0
2 = 0 0 1 0     49 = 0 1 0 0 1 0 0 1
3 = 0 0 1 1     5E = 0 1 0 1 1 1 1 0
4 = 0 1 0 0     64 = 0 1 1 0 0 1 0 0
5 = 0 1 0 1     73 = 0 1 1 1 0 0 1 1
6 = 0 1 1 0     38 = 0 0 1 1 1 0 0 0
7 = 0 1 1 1     2F = 0 0 1 0 1 1 1 1
8 = 1 0 0 0     D0 = 1 1 0 1 0 0 0 0
9 = 1 0 0 1     C7 = 1 1 0 0 0 1 1 1
A = 1 0 1 0     8C = 1 0 0 0 1 1 0 0
B = 1 0 1 1     9B = 1 0 0 1 1 0 1 1
C = 1 1 0 0     A1 = 1 0 1 0 0 0 0 1
D = 1 1 0 1     B6 = 1 0 1 1 0 1 1 0
E = 1 1 1 0     FD = 1 1 1 1 1 1 0 1
F = 1 1 1 1     EA = 1 1 1 0 1 0 1 0
    | | | |         | | | | | | | |
   b3b2b1b0         b3 |b2 |b1 |b0 |
                       |   |   |   |
                      321 !210 !310 !320
```

Decoding a hammed byte back in to a 4-bit nibble can be done using the table below, or by a bit-twiddling algorithm. In the table, the hammed byte value is used to index the row (most significant 4 bits) and column (least significant four bits), and the decoded nibble is read out of the table. If a single bit error has been detected and corrected, the nibble is followed by ``!". If an uncorrecteable error has occurred, the table cell is ``.".

```
     LSB
   | 0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F |
MSB+-------------------------------------------------+---
 0 | 1! .  1  1! .  0! 1! .  .  2! 1! .  A! .  .  7!| 0
 1 | .  0! 1! .  0! 0  .  0! 6! .  .  B! .  0! 3! . | 1
 2 | .  C! 1! .  4! .  .  7! 6! .  .  7! .  7! 7! 7 | 2
 3 | 6! .  .  5! .  0! D! .  6  6! 6! .  6! .  .  7!| 3
 4 | .  2! 1! .  4! .  .  9! 2! 2  .  2! .  2! 3! . | 4
 5 | 8! .  .  5! .  0! 3! .  .  2! 3! .  3! .  3  3!| 5
 6 | 4! .  .  5! 4  4! 4! .  .  2! F! .  4! .  .  7!| 6
 7 | .  5! 5! 5  4! .  .  5! 6! .  .  5! .  E! 3! . | 7
 8 | .  C! 1! .  A! .  .  9! A! .  .  B! A  A! A! . | 8
 9 | 8! .  .  B! .  0! D! .  .  B! B! B  A! .  .  B!| 9
 A | C! C  .  C! .  C! D! .  .  C! F! .  A! .  .  7!| A
 B | .  C! D! .  D! .  D  D! 6! .  .  B! .  E! D! . | B
 C | 8! .  .  9! .  9! 9! 9  .  2! F! .  A! .  .  9!| C
 D | 8  8! 8! .  8! .  .  9! 8! .  .  B! .  E! 3! . | D
 E | .  C! F! .  4! .  .  9! F! .  F  F! .  E! F! . | E
 F | 8! .  .  5! .  E! D! .  .  E! F! .  E! E  .  E!| F
---+-------------------------------------------------+---
   | 0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F |
```

The algorithmic approach shows what is going on more clearly. Assuming the hammed byte is

```
h7, h6, h5, h4, h3, h2, h1, h0
```

with `h7` being the most significant bit, we compute

```
p  = h7 ^ h6 ^ h5 ^ h4 ^ h3 ^ h2 ^ h1 ^ h0
c0 = h7 ^ h5 ^ h1 ^ h0
c1 = h7 ^ h3 ^ h2 ^ h1
c2 = h5 ^ h4 ^ h3 ^ h1
```

If the parity, `p`, is correct (equal to 1) then either 0 or 2 errors occurred. If all the check bits, `c0`, `c1`, `c2` are correct (equal to 1) then the byte was received intact, (no errors) otherwise it was damaged beyond repair (two errors).

If `p` is 0, then there was a single bit error which can be recovered:

```
c0 c1 c2    meaning
 1  1  1    error in bit h6
 1  1  0    error in bit h4
 1  0  1    error in bit h2
 0  1  1    error in bit h0
 0  0  1    error in bit h7
 0  1  0    error in bit h5
 1  0  0    error in bit h3
 0  0  0    error in bit h1
```

The erroneous bit should be flipped. Note that there is actually no need to fix errors in bits `h6`, `h4`, `h2` and `h0` since they are not used in the decoded byte.

After flipping bits if necessary, the decoded byte is then:

```
h7, h5, h3, h1
```

This page was created on 12 August 1996. It was last updated 09 September 1998.
Please send comments to Robin O'Leary **pdc at ro dot nu**
