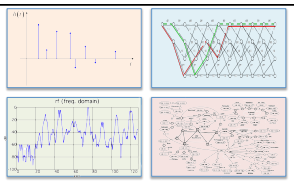


INTRODUCTION TO EECS II

DIGITAL COMMUNICATION SYSTEMS

6.02 Fall 2010
Lecture #7: Channel Coding (cont.)

Hari Balakrishnan

- Linear (n,k) block codes & Hamming codes
- Combating burst errors: interleaving
- Error detection: cyclic redundancy codes

9/29/2010

How Many Parity Bits Do We Need?

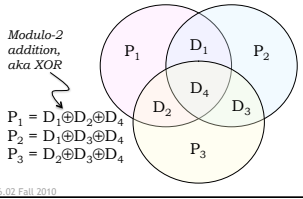
- We have n-k parity bits, which collectively can represent 2^{n-k} possibilities
- For single-bit error correction, parity bits need to represent two sets of cases:
  - Case 1: No error has occurred (1 possibility)
  - Case 2: Exactly one of the code word bits has an error (n possibilities, not k)
- So we need $n+1 \le 2^{n-k}$

$$n \le 2^{n-k} - 1$$

- Hamming codes correct single errors with this minimum number of parity bits (7,4,3), (15,11,3), …

Example: (7,4,3) Hamming Code

- Use multiple parity bits, each covering a subset of the data bits.
- No two message bits belong to exactly the same subsets, so a single-bit error will generate a unique set of parity check errors.



Suppose we check the parity and discover that P2 and P3 indicate an error? bit D3 must have flipped

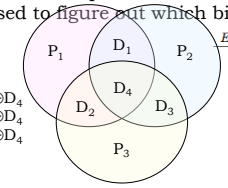
What if only P3 indicates an error? P3 itself had the error!

Syndrome Decoding

- After receiving the (possibly corrupted) message, compute a **syndrome** bit (E_i) for each parity bit

$$E_1 = D_1 \oplus D_2 \oplus D_4 \oplus P_1$$
$$E_2 = D_1 \oplus D_3 \oplus D_4 \oplus P_2$$
$$E_3 = D_2 \oplus D_3 \oplus D_4 \oplus P_3$$

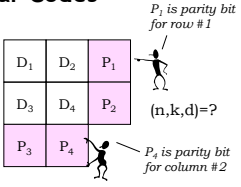
- If all the E_i are zero: no errors
- Otherwise the particular combination of the E_i can be used to figure out which bit to correct



$E_3E_2E_1$	Corrective Action
000	no errors
001	p_1 has an error, flip to correct
010	p_2 has an error, flip to correct
011	d_1 has an error, flip to correct
100	p_3 has an error, flip to correct
101	d_2 has an error, flip to correct
110	d_3 has an error, flip to correct
111	d_4 has an error, flip to correct

Rectangular Codes

Idea: start with rectangular array of data bits, add parity checks for each row and column. Single-bit error in data will show up as parity errors in a particular row and column, pinpointing the bit that has the error.



0 1 1
1 1 0
1 0

Parity for each row and column is correct ⇒ no errors

0 1 1
1 0 0
1 0

Parity check fails for row #2 and column #2 ⇒ bit D_4 is incorrect

0 1 1
1 1 1
1 0

Parity check only fails for row #2 ⇒ bit P_2 is incorrect

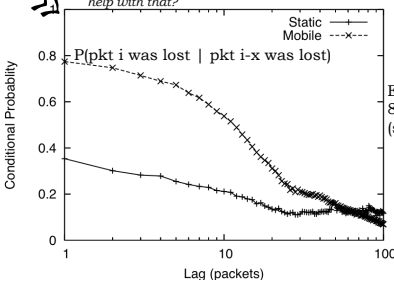
If we add an overall parity bit P_5, we get a (9,4,4) code!

Linear Block Codes: Wrap-Up

- (n,k,d) codes with rate k/n
- Code words are linear operations over message bits: sum of any two code words is a code word
- Message + 1 parity bit: (n+1,n,2) code
  - Good code rate, but only 1-bit error detection
- Replicating each bit c times is a (c,1,c) code
  - Simple way to get great error correction; poor code rate
- Hamming single-error correcting codes are (n,n-p,3) where $n = 2^p-1$ for $p > 1$
  - Adding an overall parity bit makes the code (n+1,n-p,4)
- Rectangular parity codes are (rc+r+c, rc, 3) codes
  - Rate not as good as Hamming codes

Burst Errors

Correcting single-bit errors is nice, but in many situations errors come in bursts many bits long (e.g., damage to storage media, fading or burst of interference on wireless channel, etc.). How does single-bit error correction help with that?

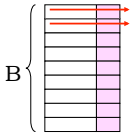


Experiment on 802.11g network (static and mobile)

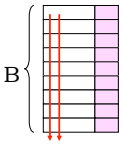
Note: P(pkt lost) ≈ s*P(bit error) where s = pkt size

Coping with Burst Errors

Well, can we think of a way to turn a B-bit error burst into B single-bit errors?



Row-by-row transmission order



Col-by-col transmission order

Problem: Bits from a particular codeword are transmitted sequentially, so a B-bit burst produces multi-bit errors.

Solution: interleave bits from B different codewords. Now a B-bit burst produces 1-bit errors in B different codewords.

Error Detection Schemes

- Simple checksum
  - Add up all the message units and send along sum
  - Easy for two errors to offset one another
    - Some 0 bit changed to 1; 1 bit in same position in another message unit changed 0… sum is unchanged
- Weighted checksum
  - Add up all the message units, each weighted by its index in the message, send along sum
  - Still too easy for two errors to offset one another
- Both! Adler-32
  - A = (1 + sum of message units) mod 65521
  - B = (sum of A_i after each message unit) mod 65521
    = $(n + n*m_1 + (n-1)*m_2 + \ldots + 1*m_n)$ mod 65521
  - Send 32-bit quantity (B << 16) + A
  - Cheaper in SW, but not quite as good as a CRC
  - Not good for short messages

Cyclic Redundancy Check (CRC)

- Cyclic code: If w is a code word, any cyclic shift of w is also a code word
- Think of bits in w as coefficients of polynomial of degree n-1. Add with GF(2) rules (i.e, XOR)
  - E.g., 11000101 would be $x^7 + x^6 + x^2 + 1$
- Each valid code word w(x) is a multiple of a generator polynomial, g(x)
- Given a message m(x), construct w(x) as
  $w(x) = x^{n-k}\, m(x) + R\{x^{n-k}\, m(x) \,/\, g(x)\}$
  where R{a/b} is "remainder when a is divided by b"
- The CRC check bits (there are n-k of them) are given by R{…}
- Receiver gets r(x), divides by g(x), and sees if remainder is 0 (if 0, then no errors detected)

CRC Example

M = 1101011011, G = 10011

Message bits

W = 11010110111110

```
          1100001010
10011 | 11010110110000
        10011
        10011
        10011
        00001
        00000
        00010
        00000
        00101
        00000
        01011
        00000
        10110
        10011
        01010
        00000
        10100
        10011
        01110
        00000
        1110
```

Remainder bits

Good CRC Generator Polynomials

- Receiver gets w(x) + e(x), where e() is the error term
- If non-zero e(x) evenly divides g(x), then we have an undetected error
- If (1+x) is a factor of g(x), then all odd numbers of errors will be detected
  - Because (1+x)(any polynomial) will have an even number of terms, so all valid codewords will have even parity
- Double error pattern has $e(x) = x^i + x^j$
  - If g(x) does not have x as a factor and if it does not evenly divide $1 + x^{j-i}$ for $1 <= j-i <= n-1$, then can detect 2 errors
- Can extend rule to detect burst errors of bounded size (burst size < n-k bits)
- Many CRC polynomials exist, used very widely