EECS 122, Lecture 6

Kevin Fall

kfall@cs.berkeley.edu

Errors

- Errors occur due to noise or interference on a communication channel
- Error detection and correction
  - error detecting (correcting) codes
  - retransmission (ARQ)
- Usually, codes are used for bit errors, ARQ is used for packets

Channel Coding

- Codes to correct for errors in channel (versus source coding--compression)
- Benefits due to these phenomena
  - Redundancy
  - Noise averaging (over long time spans)
- Types of codes
  - block codes, tree codes

Block and Tree Codes

- Block codes
  - k input bits -> n output bits; an "(n,k) code"
  - memoryless process, simple mapping
  - code rate R = k/n [typ 0.25< R 0.875]
- Tree codes (incl. convolutional codes)
  - k input, n output, n is f(v+k input bits)
  - v > 0 implies process has memory

Where are Codes Used?

- Used on storage media (magnetic tape, CDs, etc)
- Common examples
  - Parity bits
  - Cyclic redundancy check (CRC)
  - Internet checksum
- (we will look briefly at block codes)

Basics

- Hamming weight is # of 1's in a word
- Hamming distance (d) is # of differences
  - 110101, 111001 have d = 2
  - (also the Ham. weight of their XOR!)
- At least some errors can be detected or corrected if, for a code with HD d:

  d >= (# errors that can be detected) + (# errors that can be corrected) + 1

Basics 2

- A pattern of t or fewer errors can be detected *and* corrected if:
  - d >= 2t + 1 (use closest code word)
- The minimum distance of the code is the smallest d of any codeword pairs
- Want codes with as large as possible minimum distance

Simple Parity

- Starting with n-1 information bits, construct the nth bit so that the Hamming weight is even (even parity)
- Will detect an odd number of bit errors
- Does not handle even # of errors
- Does not correct

Parity Check Code

- Consider a codeword to be of form:

$$i_1 i_2 i_3 \ldots i_k p_1 p_2 p_3 p_r$$

  - (symmetric form...info comes first)
  - then for (n,k) block code, n = k + r
- We can think of selecting a codeword **c** as a matrix multiplication (w/mod-2 +):

c = **mG**

- **m** is message, **G** is *generator matrix*

Parity Generation

- **G** is a k x n (k rows) matrix:

$$G = [I \mid Z^T] = \begin{bmatrix} 1 & 0 & 0 & 0 & \ldots & 0 & z_{11} & z_{21} & \ldots & z_{r1} \\ 0 & 1 & 0 & 0 & \ldots & 0 & z_{12} & z_{22} & \ldots & z_{r2} \\ . & & & & & & . & & & \\ . & & & & & & . & & & \\ 0 & 0 & 0 & 0 & \ldots & 1 & z_{1k} & z_{2k} & \ldots & z_{rk} \end{bmatrix}$$

The Z Matrix

- entries in Z are binary numbers specified to give the desired codewords in the (n,k) block code [Hamming is 1 example]
- Want this relationship:

$$\begin{aligned} p_1 &= z_{1,1} i_1 \oplus z_{1,2} i_2 \oplus \ldots z_{1,k} i_k \\ p_2 &= z_{2,1} i_1 \oplus z_{2,2} i_2 \oplus \ldots z_{2,k} i_k \\ &\ldots \\ p_r &= z_{r,1} i_1 \oplus z_{r,2} i_2 \oplus \ldots z_{r,k} i_k \end{aligned}$$

Parity Generation Example

- **c = mG** for (7,4) systematic code word:

$$c = [1011] \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} = [1011010]$$

Parity Checking

- **H** is a (n-k) x n matrix:

$$H = [Z \mid I] = \left[\begin{array}{cccc|cccccc} z_{11} & z_{12} & \cdots & z_{1k} & 1 & 0 & 0 & 0 & \ldots & 0 \\ z_{21} & z_{22} & \cdots & z_{2k} & 0 & 1 & 0 & 0 & \ldots & 0 \\ \cdot & & & & \cdot & & & & & \\ \cdot & & & & \cdot & & & & & \\ \cdot & & & & \cdot & & & & & \\ z_{r1} & z_{r2} & \cdots & z_{rk} & 0 & 0 & 0 & 0 & \ldots & 1 \end{array}\right]$$

- Syndrome: $\mathbf{s} = \tilde{\mathbf{c}}\,\mathbf{H}^T \neq \mathbf{0}$ indicates error

Hamming Codes [BSTJ-4/50]

- Special block codes with d = 3
- Because d >= 2t + 1, t = 1 (Single EC)
- Requires: $(n,k) = (2^m - 1, 2^m - 1 - m)$
- where integer m >= 3
- So, allowable codes include (7,4), (15,11), (31, 26), (63, 57), (127, 120)

Cyclic Redundancy Check (CRC)

- Block based error detection commonly used in link-layer networks
- Idea: Given a k-bit message, generate an n-bit frame check sequence (FCS) so that a combined k+n bit frame is evenly divisible by some pre-defined number
- On receipt, no remainder means no error

Messages as Polynomials

- Consider n bit message as corresponding to an (n-1) degree polynomial with the message bits as coefficients
- Example:
  - m = 10011010 $M(x) = x^7 + x^4 + x^3 + x^1$

What to Send

- Let C(x) be our divisor polynomial
  - example: $C(x) = x^3 + x^2 + x^1$ (degree 3)
- So, first scale M(x) by multiplying by degree of C(x): $M(x) = x^{10} + x^7 + x^6 + x^4$
- Now, compute remainder of M(x)/C(x)

Polynomial Division

```
          111110011
      ______________
1101 | 10011010000
       1101
       ----
        1001
        1101
        ----
         1000
         1101
         ----
          1011
          1101
          ----
           1100
           1101
           ----
              1000
              1101
              ----
               101
```

Remainder Calculation

- So, we see that 101 is the remainder
- Thus, M(x) - 101 would be evenly divisible by C(s)
- So, just subtract off 101 (remember, we pre-multiplied leaving room for it)
- Then, new message is 10011010101

Where did C(x) Come From?

- C(x) is standardized to be small but typically produce remainders. Detects:
  - all single bit errors
  - all double-bit errors if C(x) has a factor with at least 3 terms
  - any odd number of errors, if (x+1) divides C(x)
  - any burst error of length < len of FCS
  - most large burst errors

Standard CRC Polynomials

- CRC-8: 100000111
- CRC-10: 11000110011
- CRC-12: 1100000001111 (text is wrong)
- CRC-16: 11000000000000101
- CRC-CCITT: 10001000000100001
- CRC-32: 100000100110000010001110110110111

The Internet Checksum

- Used in IP, ICMP, TCP, UDP, ...
- Alg: 1's complement of the 1's complement sum of data interpreted 16 bits at a time. In 1's comp., two zeros!
- 1's complement addition is "end-round-carry" addition. Why?
  - 2's complement carry is a zero-crossing; account for -0 by adding one

Internet Checksum Example

- Message: e3 4f 23 96 44 27 99 f3
- 2's comp sum is: 1e4ff
- 1's comp sum is: e4ff + 1 = e500
- So, Internet cksum is 1aff
- Note that message + cksum = ffff
- Thus, cksum(msg+cksum) = 0000

Interesting Properties (are these good for a checksum?)

- <{0001..ffff}, +> forms Abelian Group:
  - for all X,Y (X+Y) is in {0001...ffff} [closure]
  - A + (B + C) = (A + B) + C [assoc]
  - e + X = X + e = X (for all X), e = ffff [ident]
  - for all X, X' exists where X + X' = e [inverse]
  - for all X,Y, X+Y = Y+X [commutativity]
  - not closed under complement!
  - only trivial payload results in ffff cksum

Other Characteristics

- easy to compute and check in software
- amenable to incremental updates
- not as strong as CRC
  - assume any bit error results in uniform csum value on [0000..fffe], then Prob(cksumvalid|error) = 1 in 65536, about 3x10^-5.....ok if errors are rare

Incremental Updates

- Possible to determine new cksum without touching all data...only need sum of areas being changed (from and to)
- Why useful? [for small changes]
  - Network Address Translation (NAT)
  - IP forwarding (TTL decrement)