

Text Compression Systems

- Systems
 - combinations of modeling and coding
 - PC system performance
 - on Calgary and Canterbury corpora
- Error control
- CRC

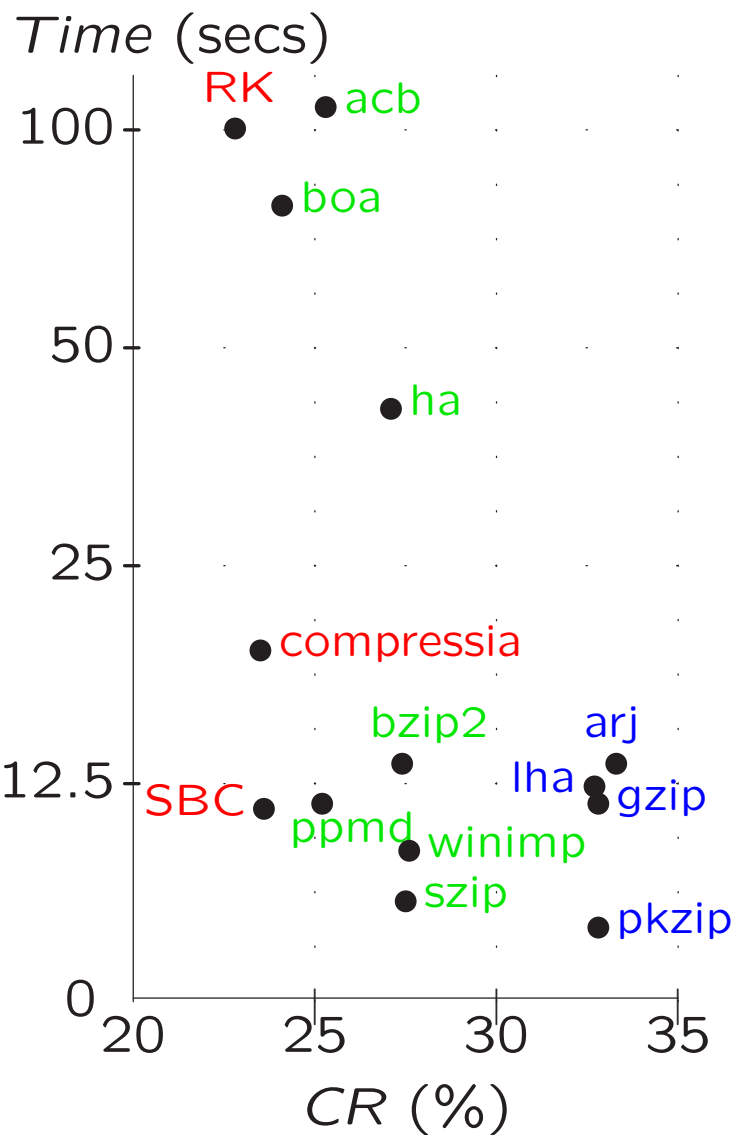
Systems

<i>method</i>	<i>model</i> <i>data struct</i>	<i>code</i>
MTF	def-word self-org list	static
LZC	dictionary hash	fixed-length
LZFG	dictionary trie	start-step-stop
SCM	context,ch hash, self-org	AC
PPMC	context,ch scaled counts	AC
WORD	context,wd wd/non-wd contexts	AC

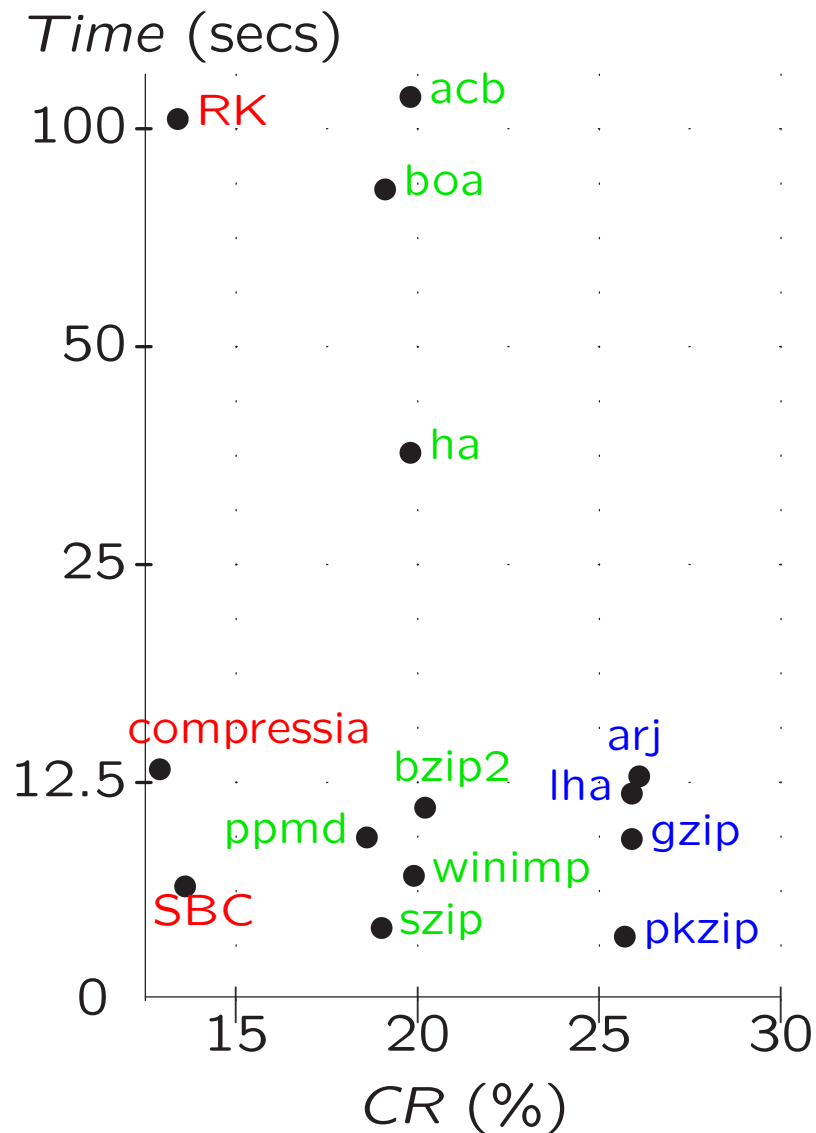
PC Compression Software

<i>program</i>	<i>method</i>	<i>code</i>
compress	LZC(hash)	fixed-length
pkarc	LZC(hash)	fixed-length
squeeze	LZ77(hash)	
pak	LZ77	blocked static Huffman
gzip	LZ77	blocked static Huffman
pkzip 1.10	LZ77	Shannon-Fano
pkzip 2.04g	LZ77(hash)	blocked Huffman
lha,zoo	LZ77(trie)	blocked adaptive Huffman
arj	LZ77(hash)	blocked adaptive Huffman
hpack	PPMC	AC
ha	0-4 Markov model	
bzip2	BWT	
winimp	BWT	
szip	BWT variation	
SBC	BWT with var blocksize	
RK	reduced-offset LZ, PPMZ	

PC Compression Software



3.2 Mb Calgary Corpus



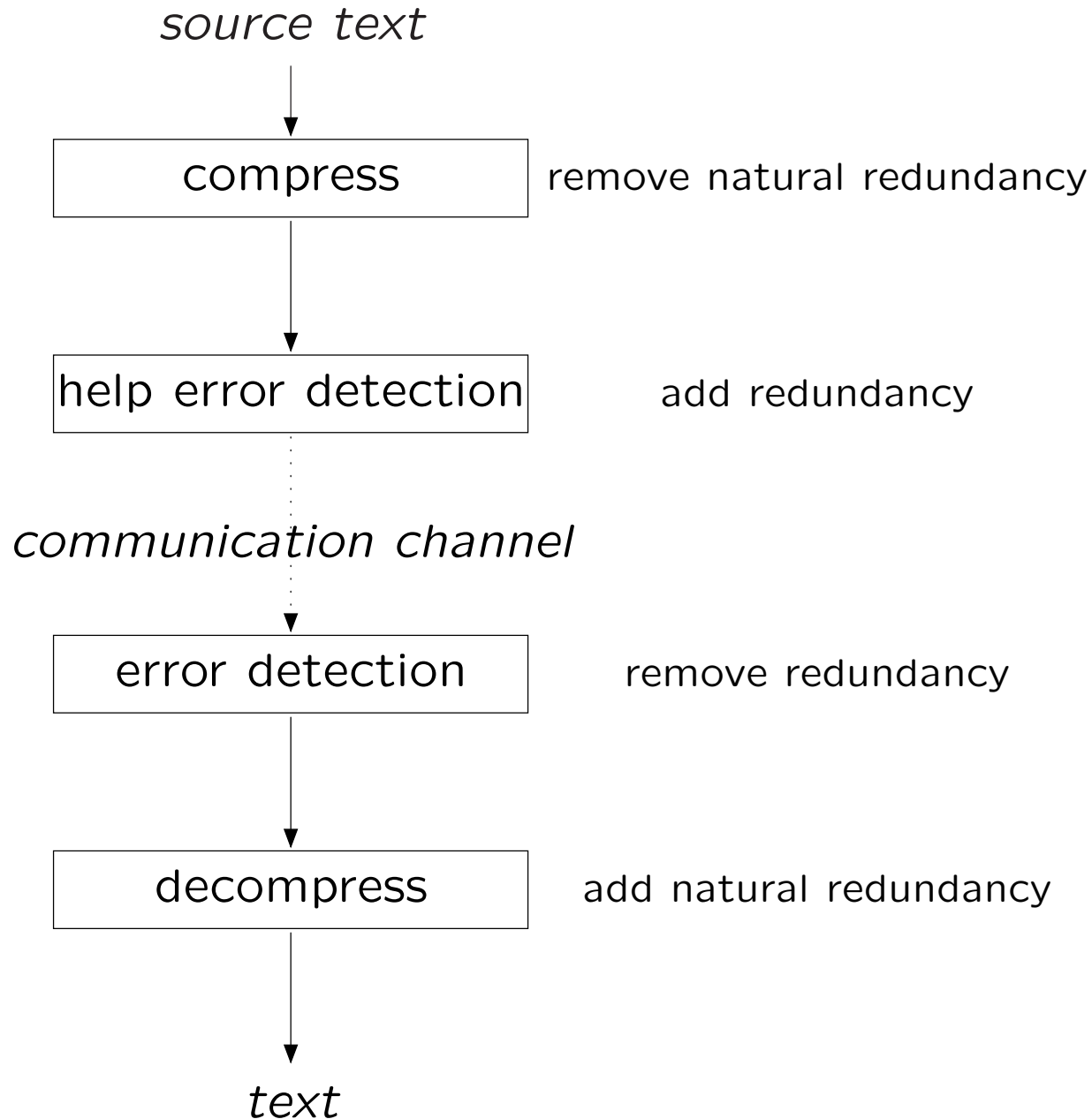
2.8 Mb Canterbury Corpus

compression ratio vs. compress time
P-200, 64 Mb, Win 98

Error Detection/Correction

- can apply CRC, etc., post compression
- essentially contrary to compression
- robustness in face of errors
 - adaptive methods most susceptible
 - a single error in LZ'77 can corrupt $O(n^{2/3})$ phrases
(see *IEEE Trans Info Th*, 2006 pp.1974-1989)
 - static Huffman codes *tend* to re-synchronize
 - Note:** this fact enables parallel decoding
(see DCC'00, pp.383-392)
 - Fibonacci codes limit error propagation

Error Detection/Correction



CRC - Cyclical Redundancy Control

- better than simple additive “checksum”
as each input byte affects many CRC bits
- CRC algorithm
 - append k 0's to input bit string
 - divide by CRC- k value using mod-2 arithmetic
(no carries/borrows: $+$, $-$, \oplus are equivalent)
the remainder is the CRC code, having k bits
(the high bit in the CRC- k value is 2^k)
 - transmit input string and the k -bit CRC code
 - receiver divides transmission by CRC- k value and should net zero
 - efficient implementation is table-driven
- CRC values
 - CRC-32 = 100000100110000010001110110110111
bits on: 32, 26, 23, 22, 16, 12, 11, 10, 8, 7, 5, 4, 2, 1, 0
 - X25 standard = bits 16, 12, 5, 0
 - CRC-16 = bits 16, 15, 2, 0 CRC-12 = bits 12, 3, 1, 0

Example of Division using mod 2 arithmetic

```

          11010
          -----
10011 ) 110001101
        10011 . .
        ----- . .
         10111 . .
         10011 . .
         ----- . .
           01001 .
           00000 .
           ----- .
             10010 .
             10011 .
             ----- .
               00011
               00000
               -----
                 0011 = Remainder = THE CHECKSUM
```

Implementing CRC Division

- Simple approach

augment input bitstream by appending k zero bits

$CRC \leftarrow CRC-k$ value without leading bit (2^k)

initialize register $R \leftarrow 0$

while more input bits

 oldtop \leftarrow hi-bit of R

 shift R one bit left

R bit-position 0 \leftarrow next augmented input bit

if oldtop=1

then $R \leftarrow R \oplus CRC$

R now contains the remainder

- in effect this subtracts various shifts of the $CRC-k$ value

Analysis of Simple CRC Algorithm

Let CRC bits = $\mathbf{g}_1 \mathbf{g}_2 \dots$, R bits = $\mathbf{t}_1 \mathbf{t}_2 \dots$

Define $R^{(k)}$ to be value of R before iteration k

- in iteration 1, CRC is \oplus into R only if $\mathbf{t}_1=1$
 after iteration 1, $R^{(2)} = \mathbf{t}_2 \mathbf{t}_3 \mathbf{t}_4 \dots$
 $\quad \quad \quad + \mathbf{t}_1 * (\mathbf{g}_1 \mathbf{g}_2 \mathbf{g}_3 \mathbf{g}_4 \dots)$
- $R^{(2)}$'s top bit = $\mathbf{t}_2 + \mathbf{t}_1 * \mathbf{g}_1$ controls iteration 2
 can calculate $R^{(2)}$'s top bit from R 's top 2 bits
- can calculate $R^{(k)}$'s top bit from R 's top k bits
- after 8 iterations using calculated control bits
 - R 's top byte now doesn't matter, R 's other bits will be shifted left one byte
 - one byte of input will be shifted into R
 - R will have been subjected to a series of \oplus 's
 in accordance with the calculated control bits
- the effect of \oplus CRC at various offsets to R
 is the same as \oplus just one constant value to R

Table-driven CRC Calculation

- byte-oriented CRC algorithm

while augmented message not exhausted

calculate control byte from R 's top byte

$X \leftarrow$ sum of CRC at various offsets that are to be \oplus -ed into R
in accordance with the control byte

shift R left one byte

read new input byte into rightmost byte of R

$R \leftarrow R \oplus X$

- most of the calculation can be pre-computed

while augmented message not exhausted

$idx \leftarrow$ leftmost byte of R

shift R left by one byte

read in a new input byte

use idx to index table of 256 32-bit values

$R \leftarrow R \oplus \text{table_value}$

Table-driven CRC Calculation

- C code

```
reg = 0;
while (len--)
{
    byte temp = (reg >> 24) & 0xFF;
    reg = (reg << 8) | *input_ptr++;
    reg ^= table[temp];
}
```

- equivalent C code

```
r=0; while (len--)
    r = ((r<<8)|*p++) ^ t[(r>>24)&0xFF];
```

Implementing CRC Division

Problem

the loop operates on *augmented* input
need to append $k/8$ zero-bytes before pointing p at it
problem if the data block given by other code

One solution:

append the following loop (after other loop)

```
for (i=0; i<k/8; i++)  
    r = (r<<8) ^ t[(r>>24)&0xFF];
```

Implementing CRC Division

Can avoid need to (1) augment input with 0-bytes
or to (2) explicitly process 0-bytes at the end

- **TAIL**

The augmented 0-bytes have no effect on R

because \oplus with 0 does not change the target.

All they do is drive the the calculation for another $k/8$ byte-cycles.

- **HEAD**

If initially $R = 0$, the only effect of the first 4 loop iterations is to shift in the first 4 input bytes,

because the first 32 control bits are all 0 and so nothing is \oplus -ed into R .

If initial value $\neq 0$, the first 4 byte-iterations have the effect of shifting 4 input bytes into R and then \oplus them with some constant value (that is a function of the initial value of R).

Implementing CRC Division

- **solution uses \oplus property:** $(A \oplus B) \oplus C = A \oplus (B \oplus C)$
input bytes need not actually travel thru R
 for the $k/8$ augmented-byte-cycles,
they can be \oplus into the top byte
 just before it is used to index the lookup table
- **modified table-driven algorithm**
initialize R
while message not exhausted
 shift R left by one byte
 read in new input byte
 \oplus top byte just rotated out of R with next input byte
 to yield an index into the table ($[0,255]$)
 \oplus table value into R

Initial R value is one used for previous algorithm fed thru the table 4 times.
(The table is such that if prev algorithm used 0, the new algorithm will too.)

Implementing CRC Division

- C code that is likely found inside current CRC implementations

```
r=0; while (len--)  
    r = (r<<8) ^ t[(r>>24) ^ *p++];
```

CRC - Example table-driven program

```
u_long table[256];
u_long crc32(u_char *buf, int len) {
    u_char *p;
    u_long  crc;
    if (!crc32_table[1])  init_crc32();
    crc = 0xffffffff;      /* preload shift reg */
    for (p = buf; len > 0; ++p, --len)
        crc = (crc << 8) ^ table[(crc >> 24) ^ *p];
    return ~crc;          /* transmit complement */
}

/* * Build auxiliary table for CRC-32.  * */
#define CRC32_POLY 0x04c11db7
init_crc32() {
    int i, j;
    u_long c;
    for (i = 0; i < 256; ++i) {
        for (c = i << 24, j = 8; j > 0; --j)
            c = c & 0x80000000 ?
                (c << 1) ^ CRC32_POLY : (c << 1);
        table[i] = c;
    }
}
```