Compression

CISC489/689-010, Lecture #6

Wednesday, Feb. 25th

Ben Carterette

Recall

- Compression
  - Fixed length, variable length
  - Information theory:
    - Very frequent symbols = high probability symbols
    - High probability symbols = very predictable symbols
    - Very predictable = low information content (low entropy)
    - Low entropy = few bits needed to transmit/encode
    - Few bits = short code.

Inverted Lists

and	1,15
aquarium	3,5
are	3,3 4,14
around	1,9
as	2,21
both	1,13
bright	3,11
coloration	3,12 4,5
derives	4,7
due	3,7
environments	1,8
fish	1,2 1,4 2,7 2,18 2,23 3,2 3,6 4,3 4,13
fishkeepers	2,1
found	1,5
fresh	2,13
freshwater	1,14 4,2
from	4,8
generally	4,15
in	1,6 4,1
include	1,3
including	1,12
iridescence	4,9

marine	2,22
often	2,2 3,10
only	2,10
pigmented	4,16
popular	3,4
refer	2,9
referred	2,19
requiring	2,12
salt	1,16 4,11
saltwater	2,16
species	1,18
term	2,5
the	1,10 2,4
their	3,9
this	4,4
those	2,11
to	2,8 2,20 3,8
tropical	1,1 1,7 2,6 2,17 3,1
typically	4,6
use	2,3
water	1,17 2,14 4,12
while	4,10
with	2,15
world	1,11

Compression

- Inverted lists are very large
  - When term positions are stored, inverted file can be as large as original data
- Compression of indexes saves disk and/or memory space
  - Typically have to decompress lists to use them
  - Best compression techniques have good *compression ratios* and are easy to decompress
- *Lossless* compression – no information lost

Inverted List Compression

- *Basic idea*: Variable-length codes.
- Common data elements use short codes while uncommon data elements use longer codes
  - Example: coding numbers
    - number sequence: $0, 1, 0, 3, 0, 2, 0$
    - possible encoding: $00\ 01\ 00\ 10\ 00\ 11\ 00$
    - encode 0 using a single 0: $0\ 01\ 0\ 10\ 0\ 11\ 0$
    - only 10 bits, but...

Compression Example

- *Ambiguous* encoding – not clear how to decode
    - another decoding:

      $$0\ 01\ 01\ 0\ 0\ 11\ 0$$

    - which represents:

      $$0, 1, 1, 0, 0, 3, 0$$

    - use unambiguous code:

Number	Code
0	0
1	101
2	110
3	111

    - which gives:

      $$0\ 101\ 0\ 111\ 0\ 110\ 0$$

Inverted List Compression

- Inverted list compression should be lossless, unambiguous, and use variable-length codes.
- The longest lists take up the most space.
  - They have the most common words.
  - They should compress very well.
    - They are least informative.

Delta Encoding

- Word count data is good candidate for compression
  - many small numbers and few larger numbers
  - encode small numbers with small codes
- Document numbers are less predictable
  - but differences between numbers in an ordered list are smaller and more predictable
- *Delta encoding*:
  - encoding differences between document numbers (*d-gaps*)

Delta Encoding

- Inverted list (without counts)

  $1, 5, 9, 18, 23, 24, 30, 44, 45, 48$

- Differences between adjacent numbers

  $1, 4, 4, 9, 5, 1, 6, 14, 1, 3$

- Differences for a high-frequency word are easier to compress, e.g.,

  $1, 1, 2, 1, 5, 1, 4, 1, 1, 3, \ldots$

- Differences for a low-frequency word are large, e.g.,

  $109, 3766, 453, 1867, 992, \ldots$

Recall Huffman Codes

- Variable-length, prefix-free (unambiguous).
  – Space-efficient.
  – Not very time-efficient.
  – Requires a full tree to decode.
- Is there a more time-efficient variable-length prefix-free code that would work well for inverted files?

Bit-Aligned Codes

- Breaks between encoded numbers can occur after any bit position (like Huffman codes)
- *Unary* code
  - Encode *k* by *k* 1s followed by 0
  - 0 at end makes code unambiguous

Number	Code
0	0
1	10
2	110
3	1110
4	11110
5	111110

Unary and Binary Codes

- Unary is very efficient for small numbers such as 0 and 1, but quickly becomes very expensive
  - 1023 can be represented in 10 binary bits, but requires 1024 bits in unary
- Binary is more efficient for large numbers, but it may be ambiguous
  - Why?

Elias-γ Code

- To encode a number *k*, compute
  - $k_d = \lfloor \log_2 k \rfloor$
  - $k_r = k - 2^{\lfloor \log_2 k \rfloor}$
  - k_d is number of binary digits, encoded in unary

Number (k)	k_d	k_r	Code
1	0	0	0
2	1	0	10 0
3	1	1	10 1
6	2	2	110 10
15	3	7	1110 111
16	4	0	11110 0000
255	7	127	11111110 1111111
1023	9	511	1111111110 111111111

Elias-δ Code

- Elias-γ code uses no more bits than unary, many fewer for k > 2
  - 1023 takes 19 bits instead of 1024 bits using unary
- In general, takes $2 \lfloor \log_2 k \rfloor + 1$ bits
- To improve coding of large numbers, use Elias-δ code
  - Instead of encoding k_d in unary, we encode $k_d + 1$ using Elias-γ
  - Takes approximately $2 \log_2 \log_2 k + \log_2 k$ bits

Elias-δ Code

- $k_d = \lfloor \log_2 k \rfloor$
  - $k_{dd} = \lfloor \log_2(k_d + 1) \rfloor$
  - $k_{dr} = k_d - 2^{\lfloor \log_2(k_d+1) \rfloor}$
  - encode k_{dd} in unary, k_{dr} in binary, and k_r in binary

Number (k)	k_d	k_r	k_{dd}	k_{dr}	Code
1	0	0	0	0	0
2	1	0	1	0	10 0 0
3	1	1	1	0	10 0 1
6	2	2	1	1	10 1 10
15	3	7	2	0	110 00 111
16	4	0	2	1	110 01 0000
255	7	127	3	0	1110 000 1111111
1023	9	511	3	2	1110 010 111111111

```
 #
 # Generating Elias-gamma and Elias-delta codes in Python
 #

 import math

 def unary_encode(n):
  return "1" * n + "0"

 def binary_encode(n, width):
     r = ""
     for i in range(0,width):
      if ((1<<i) & n) > 0:
      r = "1" + r
      else:
      r = "0" + r
     return r

 def gamma_encode(n):
     logn = int(math.log(n,2))
     return unary_encode( logn ) + " " + binary_encode(n, logn)

 def delta_encode(n):
  logn = int(math.log(n,2))
if n == 1:
  return "0"
  else:
  loglog = int(math.log(logn+1,2))
  residual = logn+1 - int(math.pow(2, loglog))
         return unary_encode( loglog ) + " " + binary_encode( residual, loglog ) + " " + binary_encode(n, logn)

 if __name__ == "__main__":
     for n in [1,2,3, 6, 15,16,255,1023]:
         logn = int(math.log(n,2))
         loglogn = int(math.log(logn+1,2))
         print n, "d_r", logn
         print n, "d_dd", loglogn
         print n, "d_dr", logn + 1 - int(math.pow(2,loglogn))
         print n, "delta", delta_encode(n)
         #print n, "gamma", gamma_encode(n)
         #print n, "binary", binary_encode(n)
```

Byte-Aligned Codes

- Variable-length bit encodings can be a problem on processors that process bytes
- *v-byte* is a popular byte-aligned code
  - A type of *restricted variable length* encoding
  - Similar to UTF-8 for Unicode
- Shortest v-byte code is 1 byte
- Numbers are 1 to 4 bytes, with high bit 1 in the last byte, 0 otherwise

V-Byte Encoding

k	Number of bytes
$k < 2^7$	1
$2^7 \leq k < 2^{14}$	2
$2^{14} \leq k < 2^{21}$	3
$2^{21} \leq k < 2^{28}$	4

k	Binary Code	Hexadecimal
1	1 0000001	81
6	1 0000110	86
127	1 1111111	FF
128	0 0000001 1 0000000	01 80
130	0 0000001 1 0000010	01 82
20000	0 0000001 0 0011100 1 0100000	01 1C A0

V-Byte Encoder

```
public void encode( int[] input, ByteBuffer output ) {
    for( int i : input ) {
        while( i >= 128 ) {
            output.put( i & 0x7F );
            i >>>= 7;
        }
        output.put( i | 0x80 );
    }
}
```

Example: i = 104 (0x68; 01101000)
i < 128, so return (0x68 | 0x80) = (01101000 | 10000000) = 11101000 = 0xE8

Example: i = 165 (0xA5; 10101000)
i >= 128, so add (0xA5 & 0x7F) = (10101000 & 01111111) = 00101000 = 0x28 to output
rightshift i 7 bits: i is now 00000001
i < 128, so add (0x01 | 0x80) = (00000001 | 10000000) = 10000001 = 0x81 to output
return 0x28 0x81

V-Byte Decoder

```
public void decode( byte[] input, IntBuffer output ) {
    for( int i=0; i < input.length; i++ ) {
        int position = 0;
        int result = ((int)input[i] & 0x7F);

        while( (input[i] & 0x80) == 0 ) {
            i += 1;
            position += 1;
            int unsignedByte = ((int)input[i] & 0x7F);
            result |= (unsignedByte << (7*position));
        }

        output.put(result);
    }
}
```

Compression Example

- Consider inverted list with positions:

$$(1, 2, [1, 7])(2, 3, [6, 17, 197])(3, 1, [1])$$

- Delta encode document numbers and positions:

$$(1, 2, [1, 6])(1, 3, [6, 11, 180])(1, 1, [1])$$

- Compress using v-byte:

```
81 82 81 86 81 82 86 8B 01 B4 81 81 81
```

Comparison of Compression Methods

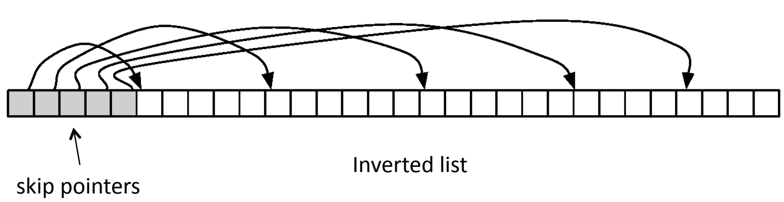
n	V-byte	gamma	delta
1	8	1	2
2	8	3	3
3	8	3	3
4	8	5	6
8	8	7	7
16	8	9	10
128	16	15	14
1,000	16	19	16
10,000	16	27	20
16,385	24	29	21
100,000	24	33	25
1,000,000	24	39	28

Skipping

- Search involves comparison of inverted lists of different lengths
  - Can be very inefficient
  - “Skipping” ahead to check document numbers is much better
  - Compression makes this difficult
    - Variable size, only d-gaps stored
- Skip pointers are additional data structure to support skipping

Skip Pointers

- A skip pointer (*d, p)* contains a document number *d* and a byte (or bit) position *p*
  - Means there is an inverted list posting that starts at position *p*, and the posting before it was for document *d*



Skip Pointers

- Example
  - Inverted list

$5, 11, 17, 21, 26, 34, 36, 37, 45, 48, 51, 52, 57, 80, 89, 91, 94, 101, 104, 119$

  - D-gaps

$5, 6, 6, 4, 5, 9, 2, 1, 8, 3, 3, 1, 5, 23, 9, 2, 3, 7, 3, 15$

  - Skip pointers

$(17, 3), (34, 6), (45, 9), (52, 12), (89, 15), (101, 18)$