Information Retrieval

WS 2015 / 2016

Lecture 4, Tuesday November 10\textsuperscript{th}, 2015
(Compression, Codes, Entropy)

Prof. Dr. Hannah Bast
Chair of Algorithms and Data Structures
Department of Computer Science
University of Freiburg
Overview of this lecture

- **Organizational**
  - Your experiences with ES3 Efficient List Intersection
  - Assistant: Björn → Elmar

- **Compression**
  - Motivation saves space and query time
  - Codes Elias, Golomb, Variable-Byte
  - Entropy Shannon's theorem

  - **Exercise Sheet 4:** prove the optimality of some of the codes + investigate the limits of what can be achieved

We take a break from implementation work this week
Experiences with ES3  1/2

Summary / excerpts

- Interesting exercise, many liked performance tweaking
- Submissions are getting less ... not unusual though
- Sentinels already implemented in C++, but not in Java
- Not so easy to improve on a tuned baseline
- Complex improvements can cost more than they help
  
  E.g. galloping always slower than binary search for some
- Skip pointers can be simulated via a simple offset
  
  Like a low-budget version of galloping search
- Seemingly insignificant code changes can affect runtime
Experiences with ES3  2/2

Results

- Three inverted lists of different lengths
  
<table>
<thead>
<tr>
<th>Category</th>
<th>Postings</th>
<th>Repeated</th>
</tr>
</thead>
<tbody>
<tr>
<td>film</td>
<td>171,951</td>
<td>100 times</td>
</tr>
<tr>
<td>comedy</td>
<td>27,706</td>
<td>100 times</td>
</tr>
<tr>
<td>2015</td>
<td>285</td>
<td>100 times</td>
</tr>
</tbody>
</table>

- Query film+2015, list length ratio = 603
  Any of galloping, skip ptrs, bin. search give large speedup

- Query comedy+2015, list length ratio = 97
  Skipping helps, but not too much

- Query film+comedy, list length ratio = 6
  Skipping costs more than it helps, switch to tuned baseline
Motivation

- Inverted lists can become very large

Recall: length of an inverted list of a word = total number of occurrences of that word in the collection

For example, in the English Wikipedia:

- film: 1,667,147 occurrences
- year: 2,052,964 occurrences
- one: 4,022,417 occurrences

- Compression potentially saves space and time
Index in **memory**

- Then compression saves memory (obviously)
- Also: the index might be too large to fit into memory without compression, and with compression it does fit

Fitting in memory is good because reading from memory is much much **much** faster than reading from disk

Transfer rate from memory  $\approx 2$ GB / second
Transfer rate from disk  $\approx 50$ MB / second
Index on disk:

- Then compression saves disk space (obviously)

- But it also saves query time, here is a realistic example:

  Disk transfer time: 50 MB / second
  Compression rate: Factor 5
  Decompression time: 30 MB / second
  Inverted list of size: 50 MB

  Reading uncompressed: 1.0 seconds → 50 MB
  Reading compressed: 0.2 seconds → 10 MB
  Decompressing: 0.3 seconds → 50 MB

  Reading compressed + decompression **twice faster** compared to reading uncompressed
Gap encoding

- Example inverted list (doc ids only):
  3, 17, 21, 24, 34, 38, 45, ..., 11876, 11899, 11913, ...

- Numbers small in the beginning, large in the end, using an `int` for each id would be **4 bytes per id**

- Alternative: store differences from one item to next:
  +3, +14, +4, +3, +10, +4, +7, ..., +12, +23, +14, ...

- This is called **gap encoding**

- Works as long as we process the lists from left to right

- Now we have a sequence of mostly (but not always) small numbers ... how do we store these in little space?
Binary representation

- We can write number $x$ in binary using $\lceil \log_2 x \rceil + 1$ bits

<table>
<thead>
<tr>
<th>$x$</th>
<th>binary</th>
<th>number of bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>3</td>
</tr>
</tbody>
</table>

- This encoding is optimal in a sense ... see later slides

- So why not just (gap-)encode like this and concatenate:

  $+3$, $+14$, $+4$, ... → 11, 1110, 100, ... → 111110100...
Prefix-free codes, definition

- Decode bit sequence from the last slide: 111110100

  This could be:  +3, +14, +4  →  11, 1110, 100

  Could also be: +7, +6, +4  →  111, 110, 100

  Or:  +3, +3, +2, + 4  →  11, 11, 10, 100

- Problem: we have no way to tell where one code ends and the next code begins

  Equivalently: some codes are prefixes of other codes

- In a **prefix-free code**, no code is a prefix of another

  Then decoding from left to right is unambiguous
Elias-Gamma ... from 1975

- Write \( \lfloor \log_2 x \rfloor \) zeros, then \( x \) in binary like on slide 9
- Prefix-free, because the number of initial zeros tells us exactly how many bits of the code come afterwards
- Code for \( x \) has a length of exactly \( 2 \cdot \lfloor \log_2 x \rfloor + 1 \) bits

\[
\begin{align*}
1 & \rightarrow 1 \\
2 & \rightarrow 010 \\
3 & \rightarrow 011 \\
4 & \rightarrow 00100 \\
& \vdots \\
10 & \rightarrow 0001010
\end{align*}
\]
Elias-Delta ... also from 1975

- Write \( \lfloor \log_2 x \rfloor + 1 \) in Elias-Gamma, followed by \( x \) in binary (like on slide 9) but **without** the leading 1

- Elias-Delta is also prefix-free and the length of the code length is \( \lfloor \log_2 x \rfloor + 2 \log_2 \log_2 x + O(1) \) bits

Pr. in Exercise 4.1

\[
\begin{align*}
1 & \rightarrow 1 \\
2 & \rightarrow 0100 \\
3 & \rightarrow 0101 \\
4 & \rightarrow 011000 \\
5 & \rightarrow 011010 \\
\vdots & \\
10 & \rightarrow 00100010
\end{align*}
\]
Codes 3/4

Golomb (not Gollum) ... from 1966

- Comes with an integer parameter $M$, called **modulus**
- Write $x$ as $q \cdot M + r$, where $q = x \text{ div } M$ and $r = x \text{ mod } M$
- The code for $x$ is then the concatenation of:
  - $q$ written in unary with 0s
  - a single 1 (as a delimiter)
  - $r$ written in binary

**Example:**

- $M = 16$, $x = 70 = 4 \cdot 16 + 6$
- Code for $x$ is 000010110

Solomon Golomb
1932 – still alive
Variable-Byte (VB)

- Idea: use **whole bytes**, in order to avoid the (expensive) bit fiddling needed for the previous schemes

  VB often used in practice, for exactly that reason

- Use one bit of each byte to indicate whether this is the last byte in the current code or not

- VB is also used for **UTF-8 encoding** ... see later lecture

\[ x = 521 = 4 \cdot 128 + 9 \]

**VB code for x**

000010000001001

**Should be 521 unibyte.**

**Byte 1**

1000000000000000

Indicates that code continues

**Byte 2**

0000000000000001

Indicates last byte of code

Uni-binery
Motivation

- Which code compresses the best?
  
  It depends!

  But on what?

- Roughly: it depends, on the relative frequency on the numbers/symbols we want to encode

  For example, in natural language, an "e" is much more frequent than a "z"  
  
  So we should encode "e" with less bits than "z"

- The next slides will make this more precise
Entropy

- **Intuitively:** the information content of a message = the optimal number of bits to encode that message
- **Formally:** defined for a discrete random variable $X$

Without loss of generality range of $X = \{1, ..., m\}$

Think of $X$ as generating the symbols of the message

Then the **entropy** of $X$ is written and defined as

$$H(X) = - \sum_i p_i \log_2 p_i$$

where $p_i = \text{Prob}(X = i)$

E.g. $m$ symbols, each equally likely $\Rightarrow p_i = \frac{1}{m}$

$\Rightarrow H(X) = \sum_{i=1}^{m} \frac{1}{m} \cdot \log_2 \frac{1}{m} = \log_2 m$
Shannon's source coding theorem ... from 1948

Let $X$ be a random variable with finite range

For an arbitrary prefix-free (PF) encoding, let $L(x)$ be the length of the code for $x \in \text{range}(X)$

(1) For any PF encoding it holds: $E L(X) \geq H(X)$

(2) There is a PF encoding with: $E L(X) \leq H(X) + 1$

where $E$ denotes the expectation

**In words:** no code can be better than the entropy, and there is always a code as good

Claude Shannon
1916 – 2001
Central Lemma ... to prove the source coding theorem

- Denote by $L_i$ the length of the code for the $i$-th symbol, then

1. Given a PF code with lengths $L_i$ \( \Rightarrow \sum_i 2^{-L_i} \leq 1 \)

2. Given $L_i$ with $\sum_i 2^{-L_i} \leq 1$ \( \Rightarrow \) exists PF code with length $L_i$

- Note: $\sum_i 2^{-L_i} \leq 1$ is known as "Kraft's inequality"
Proof of central lemma, part (1)

- To show: given a PF code with lengths $L_i \Rightarrow \sum_i 2^{-L_i} \leq 1$

- Consider the following random experiment:
  
  Generate a random binary sequence, and pick each bit independent from all other bits
  
  Stop when you have a valid code, or when no more code is possible ... well-defined for PF codes only!

- Let $C_i$ be the event that code $i$ is generated

  
  $\Pr(C_1 \cup C_2 \cup \ldots \cup C_m) = 1$
  
  $= \Pr(C_1) + \Pr(C_2) + \ldots + \Pr(C_m) = \sum_{i=1}^{m} 2^{-L_i}$
  
  $\Pr(C_i) = 2^{-L_i}$
Proof of central lemma, part (2)

- To show: \( L_i \) with \( \sum_i 2^{-L_i} \leq 1 \) \( \Rightarrow \) exists PF code with length \( L_i \)
- Consider a complete binary tree of depth \( \max L_i \)
- Mark all left edges 0, and all right edges 1
- Consider the code lengths \( L_i \) in sorted order, smallest first
- Then iterate: pick a path of length \( L_i \) from the root, with no previous path as prefix ... this gives a PF code for symbol \( i \)

\[
\begin{align*}
L_1 &= 1, L_2 = 2, L_3 = 3, L_4 = 3 \\
\sum 2^{-L_i} &= \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{8} = 1 \text{ ok.}
\end{align*}
\]
Proof of source coding theorem, part (1)

- To show: for any PF encoding \( E L(X) \geq H(X) \)
- By definition of expectation: \( E L(X) = \sum_i p_i \cdot L_i \) (1)
- By Kraft's inequality: \( \sum_i 2^{-L_i} \leq 1 \) (2)
- Using Lagrange, it can be shown that, under the constraint (2), (1) is minimized for \( L_i = \log_2 \frac{1}{p_i} \)

\[ E L(X) = \sum_i p_i \cdot L_i \Rightarrow \sum_i p_i \cdot \log_2 \frac{1}{p_i} = H(X) \]

LAGRANGE (exercise 4.4):

\[ L = \sum_{i=1}^m p_i \cdot L_i + \lambda (1 - \sum_{i=1}^m 2^{-L_i}) \]

\[ 2^{-L_i} = e^{-\lambda \cdot \ln 2 \cdot L_i} \]

\[ \frac{\partial L}{\partial L_i} = p_i + \lambda \cdot \ln 2 \cdot 2^{-L_i} = 0 \Rightarrow \ldots \Rightarrow L_i = \log_2 \frac{1}{p_i} \]
Proof of source coding theorem, part (2)

- Show: there is a PF encoding with $E L(X) \leq H(X) + 1$

- Let $L_i = \lfloor \log_2 \frac{1}{p_i} \rfloor$, then $\sum_i 2^{-L_i} \leq 1$

  Note that rounding is necessary because the code length must be an integer, and that we need to round upwards, so that Kraft's inequality holds

- By the central lemma, part (2), there then exists a PF code with code lengths $L_i$

- By definition of expectation: $E L(X) = \sum_i p_i \cdot L_i$

  $E L(X) = \sum_{i=1}^{m} p_i \cdot \log_2 \frac{1}{p_i} \leq \sum_{i=1}^{m} p_i \cdot \log_2 \frac{1}{p_i} + \sum_{i=1}^{m} p_i = H(X) + 1$
Entropy-optimal codes

- Consider a PF code with $L_i = \text{code length for symbol } i$ and $p_i = \text{probability for symbol } i$

- We say that the code is optimal for distribution $p_i$ if

$$L_i \leq \log_2 \frac{1}{p_i} + 1$$

Then $E L(X) \leq H(X) + 1$ and by Shannon's theorem this is the best we can hope for.

For the optimality proofs from Exercise Sheet 4, it suffices that you show $L_i \leq \log_2 \frac{1}{p_i} + O(1)$
Universal codes

- A prefix-free code is **universal** if for every probability distribution over the symbols to be encoded

  \[ \mathbb{E} L(X) = O(H(X)) \]

  That is, the expected code length is within a constant factor of the optimum for *any* distribution

- Elias-Gamma, Elias-Delta, Golomb, and Variable-Byte are all universal in this sense

  For a finer distinction, the definition of optimality from the previous slide is better

  \[ \mathbb{E} L(X) \leq H(X) + 1 \text{ versus } \mathbb{E} L(X) = O(H(X)) \]
Entrophy 11/12

- Optimality of Elias-Gamma
  - Recall: code length for Elias-Gamma is $L_i = 2 \lceil \log_2 i \rceil + 1$
  - For which probability distribution is this entropy-optimal?
  - We need $L_i = 2 \lceil \log_2 i \rceil + 1 \leq \log_2 1/p_i + 1$
  - This suggests something like $p_i \approx 1/i^2$
  - Let $p_i = 1/i^2$ for $i \geq 2$, and $p_1$ such that $\sum_i p_i = 1$
    That is, numbers $i \geq 2$ occur with probability $1/i^2$
Optimality of Golomb

- Consider the following random experiment for the generation of an inverted list $L$ of length $m$:
  
  Include each document $i$ in $L$ with probability $p = m/N$, independently of each other, where $N = \#\text{documents}$

- Let $G$ be a fixed gap in this inverted list, then
  
  $\Pr(G = i) = (1 - p)^{i - 1} \cdot p =: p_i$ for $i = 1, 2, 3, \ldots$

- Exercise 4.3: prove that Golomb is optimal for this distr.

- Bottom line: Golomb is optimal for gap-encoded lists

  But not practical, because of the bit fiddling, see slide 14
References

- **Textbook**
  
  Section 5: Index compression
  
  Section 5.3: Postings file compression  some codes only

- **Wikipedia**
  
  
  
  
  
  