Inverted Indexes
Compressed Inverted Indexes
Compressed Inverted Indexes

It is possible to combine index compression and text compression without any complication.

In fact, in all the construction algorithms mentioned, compression can be added as a final step.

In a full-text inverted index, the lists of text positions or file identifiers are in ascending order.

Therefore, they can be represented as sequences of gaps between consecutive numbers.

Notice that these gaps are small for frequent words and large for infrequent words.

Thus, compression can be obtained by encoding small values with shorter codes.
Compressed Inverted Indexes

A coding scheme for this case is the **unary code**

- In this method, each integer \( x > 0 \) is coded as \((x - 1)\) 1-bits followed by a 0-bit

A better scheme is the Elias-\(\gamma\) code, which represents a number \( x > 0 \) by a concatenation of two parts:

1. a unary code for \( 1 + \lfloor \log_2 x \rfloor \)
2. a code of \( \lfloor \log_2 x \rfloor \) bits that represents the number \( x - 2^{\lfloor \log_2 x \rfloor} \) in binary

Another coding scheme is the Elias-\(\delta\) code

Elias-\(\delta\) concatenates parts (1) and (2) as above, yet part (1) is not represented in unary but using Elias-\(\gamma\) instead
## Compressed Inverted Indexes

### Example codes for integers

<table>
<thead>
<tr>
<th>Gap $x$</th>
<th>Unary</th>
<th>Elias-$\gamma$</th>
<th>Elias-$\delta$</th>
<th>Golomb $(b = 3)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>100</td>
<td>1000</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>110</td>
<td>101</td>
<td>1001</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>1110</td>
<td>11000</td>
<td>10100</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>11110</td>
<td>11001</td>
<td>10101</td>
<td>1010</td>
</tr>
<tr>
<td>6</td>
<td>111110</td>
<td>11010</td>
<td>10110</td>
<td>1011</td>
</tr>
<tr>
<td>7</td>
<td>1111110</td>
<td>11011</td>
<td>10111</td>
<td>1100</td>
</tr>
<tr>
<td>8</td>
<td>11111110</td>
<td>1110000</td>
<td>11000000</td>
<td>11010</td>
</tr>
<tr>
<td>9</td>
<td>111111110</td>
<td>1110001</td>
<td>11000001</td>
<td>11011</td>
</tr>
<tr>
<td>10</td>
<td>1111111110</td>
<td>1110010</td>
<td>11000010</td>
<td>11100</td>
</tr>
</tbody>
</table>

Note: Golomb codes will be explained later.
Compressed Inverted Indexes

In general,

- Elias-$\gamma$ for an arbitrary integer $x > 0$ requires $1 + 2\lfloor \log_2 x \rfloor$ bits
- Elias-$\delta$ requires $1 + 2\lfloor \log_2 \log_2 2x \rfloor + \lfloor \log_2 x \rfloor$ bits

For small values of $x$, Elias-$\gamma$ codes are shorter than Elias-$\delta$ codes, and the situation is reversed as $x$ grows.

Thus the choice depends on which values we expect to encode.
Golomb presented another coding method that can be parametrized to fit smaller or larger gaps.

For some parameter $b$, let $q$ and $r$ be the quotient and remainder, respectively, of dividing $x - 1$ by $b$.

I.e., $q = \lfloor (x - 1)/b \rfloor$ and $r = (x - 1) - q \cdot b$.

Then $x$ is coded by concatenating

- the unary representation of $q + 1$
- the binary representation of $r$, using either $\lfloor \log_2 b \rfloor$ or $\lceil \log_2 b \rceil$ bits.
Compressed Inverted Indexes

If \( r < 2^{\lfloor \log_2 b \rfloor - 1} \) then \( r \) uses \( \lfloor \log_2 b \rfloor \) bits, and the representation always starts with a 0-bit.

Otherwise it uses \( \lceil \log_2 b \rceil \) bits where the first bit is 1 and the remaining bits encode the value \( r - 2^{\lfloor \log_2 b \rfloor - 1} \) in \( \lfloor \log_2 b \rfloor \) binary digits.

For example,

- For \( b = 3 \) there are three possible remainders, and those are coded as 0, 10, and 11, for \( r = 0 \), \( r = 1 \), and \( r = 2 \), respectively.
- For \( b = 5 \) there are five possible remainders \( r \), 0 through 4, and these are assigned the codes 00, 01, 100, 101, and 110.
Compressed Inverted Indexes

To encode the lists of occurrences using Golomb codes, we must define the parameter $b$ for each list.

Golomb codes usually give better compression than either Elias-$\gamma$ or Elias-$\delta$.

However, they need two passes to be generated as well as information on terms statistics over the whole document collection.

For example, in the TREC-3 collection, the average number of bits per list entry for each method is:

- Golomb = 5.73
- Elias-$\delta$ = 6.19
- Elias-$\gamma$ = 6.43

This represents a five-fold reduction in space compared to a plain inverted index representation.
Let us now consider inverted indexes for ranked search.

In this case the documents are sorted by decreasing frequency of the term or other similar type of weight.

Documents that share the same frequency can be sorted in increasing order of identifiers.

This will permit the use of gap encoding to compress most of each list.
Text Compression
Text Compression

- A representation of text using less space
- Attractive option to reduce costs associated with
  - space requirements
  - input/output (I/O) overhead
  - communication delays
- Becoming an important issue for IR systems
- *Trade-off*: time to encode versus time to decode text
Text Compression

Our focus are compression methods that

- allow random access to text
- do not require decoding the entire text

Important: compression and decompression speed

- In many situations, decompression speed is more important than compression speed
- For instance, in textual databases in which texts are compressed once and read many times from disk

Also important: possibility of searching text without decompressing

- faster because much less text has to be scanned
Compression Methods

Two general approaches

- statistical text compression
- dictionary based text compression

Statistical methods

- Estimate the probability of a symbol to appear next in the text
- **Symbol**: a character, a text word, a fixed number of chars
- **Alphabet**: set of all possible symbols in the text
- **Modeling**: task of estimating probabilities of a symbol
- **Coding** or **encoding**: process of converting symbols into binary digits using the estimated probabilities
Compression Methods

Dictionary methods

- Identify a set of sequences that can be referenced
- Sequences are often called **phrases**
- Set of phrases is called the **dictionary**
- Phrases in the text are replaced by pointers to dictionary entries
Statistical Methods

Defined by the combination of two tasks
- the **modeling task** estimates a probability for each next symbol
- the **coding task** encodes the next symbol as a function of the probability assigned to it by the model

A **code** establishes the representation (**codeword**) for each source symbol

The entropy $E$ is a **lower bound** on compression, measured in bits per symbol
Golden rule

Shorter codewords should be assigned to more frequent symbols to achieve higher compression.

If probability $p_c$ of a symbol $c$ is much higher than others, then $\log_2 \frac{1}{p_c}$ will be small.

To achieve good compression:

- Modeler must provide good estimation of probability $p$ of symbol occurrences.
- Encoder must assign codewords of length close to $\log_2 \frac{1}{p}$. 
Compression models can be adaptive, static, or semi-static, character-based or word-based.

**Adaptive models:**
- start with no information about the text
- progressively learn the statistical text distribution
- need only one pass over the text
- store no additional information apart from the compressed text

Adaptive models provide an inadequate alternative for full-text retrieval:
- decompression has to start from the beginning of text
Static models

- Assume an average distribution for all input texts.
- Modeling phase is done only once for all texts.
- Achieve poor compression ratios when data deviate from initial statistical assumptions.
- A model that is adequate for English literary texts will probably perform poorly for financial texts.
Semi-static models

Do not assume any distribution on the data

Learn data distribution (fixed code) in a first pass

Text compressed in a second pass using fixed code from first pass

Information on data distribution sent to decoder before transmitting encoded symbols

Advantage in IR contexts: direct access

Same model used at every point in compressed file
Semi-static models

- Simple semi-static model: use **global frequency** information

- Let $f_c$ be the frequency of symbol $c$ in the text
  \[ T = t_1 t_2 \ldots t_n \]

- The corresponding entropy is
  \[ E = \sum_{c \in \Sigma} \frac{f_c}{n} \log_2 \frac{n}{f_c} \]

- This simple modeling may not capture the redundancies of the text
Semi-static models

In the 2 gigabyte TREC-3 collection:

- Entropy under this simple model: 4.5 bits per character
- Compression ratio cannot be lower than 55%
- But, state-of-the-art compressors achieve compression ratios between 20% and 40%
Semi-static models

**Order $k$** of a model

- Number of symbols used to estimate probability of next symbol
- Zero-order model: computed independently of context
- Compression improves with higher-order models
  - Model of order 3 in TREC-3 collection
    - compression ratios of 30%
    - handling about 1.3 million frequencies
  - Model of order 4 in TREC-3 collection
    - compression ratio of 25%
    - handling about 6 million frequencies

In adaptive compression, a higher-order modeler requires much more memory to run.
Word-based Modeling

- **Word-based modeling** uses zero-order modeling over a sequence of words

- Good reasons to use word-based models in IR
  - Distribution of words more skewed than that of individual characters
  - Number of different words is not as large as text size
  - Words are the atoms on which most IR systems are built
  - Word frequencies are useful in answering queries
Word-based Modeling

Two different alphabets can be used

- one for words
- one for separators

In TREC-3, 70% – 80% of separators are spaces

Good properties of word-based models stem from well-known statistical rules:

- **Heaps’ law**: \( V = O(n^\beta) \),

- **Zipf’s law**: the \( i \)-th most frequent word occurs \( O(n/i^\alpha) \) times
Statistical Methods: Coding

- **Codeword**: representation of a symbol according to a model

- **Encoders**: generate the codeword of a symbol (coding)
  - assign short codewords to frequent symbols
  - assign long codewords to infrequent ones
  - entropy of probability distribution is lower bound on average length of a codeword

- **Decoders**: obtain the symbol corresponding to a codeword (decoding)

- Speed of encoder and decoder is important
Symbol code: an assignment of a codeword to each symbol

The least we can expect from a code is that it be uniquely decodable

Consider three source symbols $A$, $B$, and $C$

Symbol code: $A \rightarrow 0$, $B \rightarrow 1$, $C \rightarrow 01$

Then, compressed text 011 corresponds to $ABB$ or $CB$?
Consider again the three source symbols $A$, $B$, and $C$

Symbol code: $A \rightarrow 00$, $B \rightarrow 11$, $C \rightarrow 110$

This symbol code is uniquely decodable

However, for the compressed text $110000000$

we must count total number of zeros to determine whether first symbol is $B$ or $C$

A code is said to be **instantaneous** if every codeword can be decoded immediately after reading its last bit

**Prefix-free** or **prefix** codes: no codeword should be a prefix of another
Huffman coding

- a method to find the best prefix code for a probability distribution
- Let $\{p_c\}$ be a set of probabilities for the symbols $c \in \Sigma$
  - Huffman method assigns to each $c$ a codeword of length $\ell_c$
  - Idea: minimize $\sum_{c \in \Sigma} p_c \cdot \ell_c$
- In a first pass, the modeler of a semi-static Huffman-based compression method:
  - determines the probability distribution of the symbols
  - builds a coding tree according to this distribution
- In a second pass, each text symbol is encoded according to the coding tree
Huffman Codes

Figure below presents an example of Huffman compression

Original text:  

Compressed text:  

---

Documents: Languages & Properties, Baeza-Yates & Ribeiro-Neto, Modern Information Retrieval, 2nd Edition – p. 113
Huffman Codes

Given $V$ symbols and their frequencies in the text, the algorithm builds the Huffman tree in $O(V \log V)$ time.

Decompression is accomplished as follows:

- Stream of bits in file is traversed from left to right.
- Sequence of bits read is used to also traverse the Huffman coding tree, starting at the root.
- Whenever a leaf node is reached, the corresponding word or separator is printed out and the tree traversal is restarted.

In our example, the presence of the codeword 110 in the compressed file leads to the symbol for.
Huffman Codes

The Huffman tree for a given probability distribution is not unique.

Original text: for my rose, a rose is a rose
Compressed text: 110 010 00 011 10 00 111 10 00

Canonical tree

- right subtree of no node can be taller than its left subtree
- can be stored very efficiently
- allows faster decoding

Original text: for my rose, a rose is a rose
Compressed text: 010 000 10 001 11 10 011 11 10

Byte-Huffman Codes

- Original Huffman method leads to binary coding trees
- However, we can make the code assign a sequence of whole bytes to each symbol
  - As a result, Huffman tree has degree 256 instead of 2
  - This word-based model degrades compression ratios to around 30%
  - In exchange, decompression of byte-Huffman code is much faster than for binary Huffman code
In byte-Huffman coding, direct searching on compressed text is simpler.

To search for a word in the compressed text:

1. first find it in the vocabulary
2. for TREC-3, vocabulary requires just 5 megabytes
3. mark the corresponding leaf in the tree
4. proceed over text as if decompressing, except that no symbol is output
5. instead, report occurrences when visiting marked leaves
Byte-Huffman Codes

- Process is simple and fast: only 30% of the I/O is necessary.

- Assume we wish to search for a complex pattern including ranges of characters or a regular expression:
  - just apply the algorithm over the vocabulary
  - for each match of a whole vocabulary word, mark the word
  - done only on the vocabulary (much smaller than whole text)
  - once relevant words are marked, run simple byte-scanning algorithm over the compressed text
Byte-Huffman Codes

All complexity of the search is encapsulated in the vocabulary scanning.

For this reason, searching the compressed text is

- up to 8 times faster when complex patterns are involved
- about 3 times faster when simple patterns are involved
Byte-Huffman Codes

Although the search technique is simple and uniform, one could do better especially for single-word queries.

Concatenation of two codewords might contain a third codeword.

Consider the code: $A \rightarrow 0, B \rightarrow 10, C \rightarrow 110, D \rightarrow 111$

$DB$ would be coded as 11110.

If we search for $C$, we would incorrectly report a spurious occurrence spanning the codewords of $DB$.

To check if the occurrence is spurious or not, rescan all text from the beginning.
Dense Codes

An alternative coding simpler than byte-Huffman is dense coding

Dense codes arrange the symbols in decreasing frequency order

- Codeword assigned to the $i$-th most frequent symbol is, essentially, the number $i - 1$
- Number is written in a variable length sequence of bytes
- 7 bits of each byte are used to encode the number
- Highest bit is reserved to signal the last byte of the codeword
Dense Codes

- Codewords of symbols ranked 1 to 128 are 0 to 127.
  - They receive one-byte codewords.
  - Highest bit is set to 1 to indicate last byte (that is, we add 128 to all codewords).
  - Symbol ranked 1 receives codeword $\langle 128 \rangle = \langle 0 + 128 \rangle$.
  - Symbol ranked 2 receives codeword $\langle 129 \rangle = \langle 1 + 128 \rangle$.
  - Symbol ranked 128 receives codeword $\langle 255 \rangle$.

- Symbols ranked from 129 to 16,512 (i.e., $128 + 128^2$) are assigned two-byte codewords $\langle 0, 128 \rangle$ to $\langle 127, 255 \rangle$. 
Dense Codes

- **Stoppers**
  - these are those bytes with their highest bit set
  - they indicate the end of the codeword

- **Continuers**
  - these are the bytes other than *stoppers*

- Text vocabularies are rarely large enough to require 4-byte codewords
Dense Codes

Figure below illustrates an encoding with dense codes

<table>
<thead>
<tr>
<th>Word rank</th>
<th>Codeword</th>
<th>Bytes</th>
<th># of words</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>⟨128⟩</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>⟨129⟩</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>⟨255⟩</td>
<td>1</td>
<td>128</td>
</tr>
<tr>
<td>129</td>
<td>⟨0, 128⟩</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>130</td>
<td>⟨0, 129⟩</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>⟨0, 255⟩</td>
<td>2</td>
<td>128²</td>
</tr>
<tr>
<td>257</td>
<td>⟨1, 128⟩</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>16,512</td>
<td>⟨127, 255⟩</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>16,513</td>
<td>⟨0, 0, 128⟩</td>
<td>3</td>
<td>128³</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2,113,664</td>
<td>⟨127, 127, 255⟩</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Dense Codes

- Highest bit signals the end of a codeword
- A dense code is automatically a prefix code

Self-synchronization

- Dense codes are **self-synchronizing**
  - Given any position in the compressed text, it is very easy to determine the next or previous codeword beginning
  - Decompression can start from any position, be it a codeword beginning or not
- Huffman-encoding is **not self-synchronizing**
  - Not possible to decode starting from an *arbitrary position* in the compressed text
  - Notice that it is possible to decode starting at an *arbitrary codeword beginning*
Dense Codes

- Self-synchronization allows faster search
- To search for a single word we can
  - obtain its codeword
  - search for the codeword in the compressed text using any string matching algorithm
- This does not work over byte-Huffman coding
**Dense Codes**

An *spurious occurrence* is a codeword that is a suffix of another codeword

- assume we look for codeword \( a \ b \ c \), where we have overlined the stopper byte
- there could be a codeword \( d \ a \ b \ c \) in the code, so that we could find our codeword in the text \( \ldots e \ f \ g \ d \ a \ b \ c \ldots \)
- yet, it is sufficient to access the text position preceding the candidate occurrence, ‘\( d \)’, to see that it is not a stopper

Such a fast and simple check is not possible with Huffman coding

To search for phrases

- concatenate the codewords
- search for the concatenation