Lempel-Ziv Algorithms

LZ77 (Sliding Window)
- Variants: LZSS (Lempel-Ziv-Storer-Szymanski)
- Applications: gzip, Squeeze, LHA, PKZIP, ZOO

LZ78 (Dictionary Based)
- Variants: LZW (Lempel-Ziv-Welch), LZC (Lempel-Ziv-Compress)
- Applications: compress, GIF, CCITT (modems), ARC, PAK

Traditionally LZ77 was better but slower, but the gzip version is almost as fast as any LZ78.

LZ77: Sliding Window Lempel-Ziv

Cursor

Dictionary
(previously coded)

Lookahead
Buffer

Dictionary and buffer “windows” are fixed length and slide with the cursor

On each step:
- Output (p,l,c)
  - p = relative position of the longest match in the dictionary
  - l = length of longest match
  - c = next char in buffer beyond longest match
- Advance window by 1 + l

LZ77: Example

Dictionary (size = 6)

Longest match

Next character
LZ77 Decoding

Decoder keeps same dictionary window as encoder.

- For each message it looks it up in the dictionary and inserts a copy

What if \( l > p \)? (only part of the message is in the dictionary.)

- E.g. dict = abcd, codeword = (2, 9, e)
- Simply copy starting at the cursor
  
  \[
  \text{out[cursor+i]} = \text{out[cursor-offset+i]}
  \]

- Out = abcdcdcdcdcdc

LZ77 Optimizations used by gzip

LZSS: Output one of the following formats

- \((0, \text{position}, \text{length})\) or \((1, \text{char})\)

Typically use the second format if length < 3.

Optimizations used by gzip (cont.)

- Huffman code the positions, lengths and chars
- Non greedy: possibly use shorter match so that next match is better
- Use hash table to store dictionary.
  - Hash is based on strings of length 3.
  - Find the longest match within the correct hash bucket.
  - Limit on length of search.
  - Store within bucket in order of position

Theory behind LZ77

Sliding Window LZ is Asymptotically Optimal [Wyner-Ziv,94]

Will compress long enough strings to the source entropy as the window size goes to infinity.

\[
H_n = \sum_{x \in A^n} p(x) \log \frac{1}{p(x)}
\]

\[
H = \lim_{n \to \infty} H_n
\]

Uses logarithmic code for position.

Problem: “long enough” is really really long.
LZ78: Dictionary Lempel-Ziv

Basic algorithm:
- Keep dictionary of words with integer id for each entry (e.g. keep it as a trie).
- Coding loop
  - find the longest match \( S \) in the dictionary
  - Output the entry id of the match and the next character past the match from the input \((id,c)\)
  - Add the string \( Sc \) to the dictionary
- Decoding keeps same dictionary and looks up ids

LZ78: Coding Example

<table>
<thead>
<tr>
<th>Input</th>
<th>Dict.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, a</td>
<td>(0,a)</td>
</tr>
<tr>
<td>1, b</td>
<td>a b</td>
</tr>
<tr>
<td>1, a</td>
<td>a a b a a</td>
</tr>
<tr>
<td>0, c</td>
<td>a a b a c</td>
</tr>
<tr>
<td>2, c</td>
<td>a a b a a c a b c</td>
</tr>
<tr>
<td>5, b</td>
<td>a a b a a c a b c a b c b</td>
</tr>
</tbody>
</table>

LZ78: Decoding Example

<table>
<thead>
<tr>
<th>Input</th>
<th>Dict.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, a</td>
<td>1 = a</td>
</tr>
<tr>
<td>1, b</td>
<td>2 = ab</td>
</tr>
<tr>
<td>1, a</td>
<td>3 = aa</td>
</tr>
<tr>
<td>0, c</td>
<td>4 = c</td>
</tr>
<tr>
<td>2, c</td>
<td>5 = abc</td>
</tr>
<tr>
<td>5, b</td>
<td>6 = abcb</td>
</tr>
</tbody>
</table>

LZW (Lempel-Ziv-Welch) [Welch84]

Don’t send extra character \( c \), but still add \( Sc \) to the dictionary.

The dictionary is initialized with byte values being the first 256 entries (e.g. \( a = 112 \), ascii), otherwise there is no way to start it up.

The decoder is one step behind the coder since it does not know \( c \)
- Only an issue for strings of the form \( SS\) where \( S[0] = c \), and these are handled specially
LZW: Encoding Example

<table>
<thead>
<tr>
<th>Input</th>
<th>Dict.</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b a a c a b c a b c</td>
<td>112</td>
</tr>
<tr>
<td>a b a a c a b c a b c</td>
<td>112</td>
</tr>
<tr>
<td>a b a a c a b c a b c</td>
<td>113</td>
</tr>
<tr>
<td>a b a a c a b c a b c</td>
<td>256</td>
</tr>
<tr>
<td>a b a a c a b c a b c</td>
<td>257</td>
</tr>
<tr>
<td>a b a a c a b c a b c</td>
<td>258</td>
</tr>
<tr>
<td>a b a a c a b c a b c</td>
<td>259</td>
</tr>
<tr>
<td>a b a a c a b c a b c</td>
<td>114</td>
</tr>
<tr>
<td>a b a a c a b c a b c</td>
<td>260</td>
</tr>
</tbody>
</table>

LZ78 and LZW issues

What happens when the dictionary gets too large?
- Throw the dictionary away when it reaches a certain size (used in GIF)
- Throw the dictionary away when it is no longer effective at compressing (used in unix compress)
- Throw the least-recently-used (LRU) entry away when it reaches a certain size (used in BTLZ, the British Telecom standard)

LZW: Decoding Example

<table>
<thead>
<tr>
<th>Input</th>
<th>Dict.</th>
</tr>
</thead>
<tbody>
<tr>
<td>112 a b a a c a b c a b c</td>
<td>256</td>
</tr>
<tr>
<td>112 a b a a c a b c a b c</td>
<td>257</td>
</tr>
<tr>
<td>113 a b a a c a b c a b c</td>
<td>258</td>
</tr>
<tr>
<td>256 a b a a c a b c a b c</td>
<td>259</td>
</tr>
<tr>
<td>257 a b a a c a b c a b c</td>
<td>260</td>
</tr>
<tr>
<td>260 a b a a c a b c a b c</td>
<td>261</td>
</tr>
</tbody>
</table>

Lempel-Ziv Algorithms Summary

Both LZ77 and LZ78 and their variants keep a “dictionary” of recent strings that have been seen. The differences are:
- How the dictionary is stored
- How it is extended
- How it is indexed
- How elements are removed
Lempel-Ziv Algorithms Summary (II)

Adapt well to changes in the file (e.g. a Tar file with many file types within it).
Initial algorithms did not use probability coding and perform very poorly in terms of compression (e.g. 4.5 bits/char for English)
More modern versions (e.g. gzip) do use probability coding as “second pass” and compress much better.
Are becoming outdated, but ideas are used in many of the newer algorithms.

Burrows -Wheeler

Currently best “balanced” algorithm for text

Basic Idea:
- Sort the characters by their full context (typically done in blocks). This is called the block sorting transform.
- Use move-to-front encoding to encode the sorted characters.

The ingenious observation is that the decoder only needs the sorted characters and a pointer to the first character of the original sequence.

Burrows Wheeler: Example

Let’s encode: $d_1 e_2 c_3 o_4 d_5 e_6$
We’ve numbered the characters to distinguish them.
Context “wraps” around.

<table>
<thead>
<tr>
<th>Context</th>
<th>Char</th>
<th>Encode</th>
<th>Context</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ecode_6</td>
<td>d_1</td>
<td></td>
<td>dedec_3</td>
<td>o_4</td>
</tr>
<tr>
<td>coded_1</td>
<td>e_2</td>
<td>Sort</td>
<td>coded_2</td>
<td>e_2</td>
</tr>
<tr>
<td>odede_2</td>
<td>c_3</td>
<td></td>
<td>decod_6</td>
<td>e_6</td>
</tr>
<tr>
<td>dedec_3</td>
<td>o_4</td>
<td>Context</td>
<td>odede_5</td>
<td>c_3</td>
</tr>
<tr>
<td>edeco_4</td>
<td>d_5</td>
<td></td>
<td>ecode_6</td>
<td>d_1</td>
</tr>
<tr>
<td>decod_5</td>
<td>e_6</td>
<td></td>
<td>edeco_4</td>
<td>d_5</td>
</tr>
</tbody>
</table>

Burrows-Wheeler (Continued)

Theorem: After sorting, equal valued characters appear in the same order in the output as in the most significant position of the context.

Proof: Since the chars have equal value in the most-sig-position of the context, they will be ordered by the rest of the context, i.e. the previous chars. This is also the order of the output since it is sorted by the previous characters.
Burrows-Wheeler Decode

Function BW_Decode(In, Start, n)

\[ S = \text{MoveToFrontDecode(In,n)} \]
\[ R = \text{Rank}(S) \]
\[ j = \text{Start} \]
\[ \text{for } i=1 \text{ to } n \text{ do} \]
\[ \text{Out}[i] = S[j] \]
\[ j = R[i] \]

Rank gives position of each char in sorted order.

Decode Example

<table>
<thead>
<tr>
<th>$S$</th>
<th>Sort($S$)</th>
<th>Rank($S$)</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>o₄</td>
<td>c₃</td>
<td>6</td>
<td>e₆ d₁</td>
</tr>
<tr>
<td>e₂</td>
<td>d₁</td>
<td>4</td>
<td>d₁ e₂</td>
</tr>
<tr>
<td>e₆</td>
<td>d₅</td>
<td>5</td>
<td>e₂ c₃</td>
</tr>
<tr>
<td>c₃</td>
<td>e₂</td>
<td>1</td>
<td>c₃ o₄</td>
</tr>
<tr>
<td>d₁</td>
<td>e₆</td>
<td>2</td>
<td>o₄ d₅</td>
</tr>
<tr>
<td>d₅</td>
<td>o₄</td>
<td>3</td>
<td>d₅ e₆</td>
</tr>
</tbody>
</table>

ACB (Associate Coder of Buyanovsky)

Currently the second best compression for text (BOA, based on PPM, is marginally better)

Keep dictionary sorted by context
- Find best match for context
- Find best match for contents
- Code
  - Distance between matches
  - Length of contents match

Has aspects of Burrows-Wheeler, residual coding, and LZ77

Other Ideas?
More Ideas?