15-499: Algorithms and Applications

Data Compression III
- Lempel-Ziv algorithms
- Burrows-Wheeler

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), LZW
- Applications: compress, GIF, CCITT (modems), ARC, PAK

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

Compression Outline

Introduction: Lossy vs. Lossless, Benchmarks, ...
Information Theory: Entropy, etc.
Probability Coding: Huffman + Arithmetic Coding
Applications of Probability Coding: PPM + others
Lempel-Ziv Algorithms:
- LZ77, gzip,
- LZ78, compress
Other Lossless Algorithms: Burrows-Wheeler
Lossy algorithms for images: JPEG, MPEG, ...
Compressing graphs and meshes: BBK

LZ77: Sliding Window Lempel-Ziv

Cursor
\[ \text{Dictionary} \]
\[ \text{Lookahead} \]
\[ \text{Buffer} \]

Dictionary and buffer "windows" are fixed length and slide with the cursor.

Repeat:
Output \((p, l, c)\) where
- \(p\) = position of the longest match in the dictionary (relative to the cursor)
- \(l\) = length of longest match
- \(c\) = next char in buffer beyond longest match
Advance window by \(l + 1\)
LZ77: Example

```
> a c a a c a b c a b a a a c
( _, 0 , a )
> a c a a c a b c a b a a a c
( 1 , 1 , c )
> a c a a c a b c a b a a a c
( 3 , 4 , b )
> a c a a c a b c a b a a a c
?
> a c a a c a b c a b a a a c
?
```

Dictionary (size = 6)  Longest match
Buffer (size = 4)       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. \( \text{dict} = \text{abcd}, \text{codeword} = (2, 9, e) \)
What does this give?

LZ77 Optimizations used by gzip

LZSS: Output one of the following two formats
\((0, \text{position}, \text{length})\) or \((1, \text{char})\)
Uses the second format if length < 3.

```
> a c a a c a b c a b a a a c
( 1 , a )
> a c a a c a b c a b a a a c
( 1 , a )
> a c a a c a b c a b a a a c
( 1 , c )
> a c a a c a b c a b a a a c
?
```

Optimizations used by gzip (cont.)

1. Huffman code the positions, lengths and chars
2. Non greedy: possibly use shorter match so that
   next match is better
3. Use a hash table to store the dictionary.
   - Hash keys are all strings of length 3 in the
dictionary window.
   - Find the longest match within the correct
     hash bucket.
   - Puts a limit on the length of the search within
     a bucket.
   - Within each bucket store in order of position
The Hash Table

... 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ... ...
... a a c a a c a b c a b a a a c ...

LZ78: Dictionary Lempel-Ziv

Like LZ77, but “dictionary” is maintained differently

The dictionary maintains a mapping of words to ids
Typically stored as a trie, e.g:

```
   a
 /   |
 b   c
```

<table>
<thead>
<tr>
<th>Word</th>
<th>id</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>aa</td>
<td>2</td>
</tr>
<tr>
<td>ab</td>
<td>3</td>
</tr>
<tr>
<td>abc</td>
<td>4</td>
</tr>
</tbody>
</table>

Theory behind LZ77

Sliding Window LZ is Asymptotically Optimal
[Wyner-Ziv94]

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

\[
H_n = \sum_{X \subseteq X^*} p(X) \log \frac{1}{p(X)}
\]

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

Uses logarithmic code (e.g. gamma) for the position.
Problem: “long enough” is really really long.

LZ78: Encoding and Decoding

Keep a cursor as in LZ77

```
Cursor
```

```
\begin{tabular}{|c|}
\hline
\textbf{Repeat:} \\
\textbf{Output} \textbf{(id, c)} where \\
\textbf{id} \textbf{=} The id of the longest match \textbf{S} found in the \\
dictionary \\
c = next char in buffer beyond longest match \\
Add the string \textbf{Sc} to the dictionary with a new \textbf{id} \\
Move the cursor past \textbf{S} and \textbf{c} \\
\hline
\end{tabular}
```

Decoding keeps the same dictionary and looks up ids
**LZ78: Coding Example**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, a)</td>
<td>a a b a a c a b c a b c b</td>
</tr>
<tr>
<td>(1, b)</td>
<td>a a b a a c a b c a b c b</td>
</tr>
<tr>
<td>(1, a)</td>
<td>a a b a a c a b c a b c b</td>
</tr>
<tr>
<td>(0, c)</td>
<td>a a b a a c a b c a b c b</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>

**Dict.**

- (0, a): a
- (1, b): b
- (1, a): c
- (0, c): d
- (2, c): 3
- (5, b): 5

---

**LZ78: Decoding Example**

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

**Dict.**

- (0, a): a
- (1, b): b
- (1, a): c
- (0, c): d
- (2, c): 3
- (5, b): 5

---

**LZW (Lempel-Ziv-Welch) [Welch84]**

Don't send the extra character c, but still add Sc to the dictionary.

Initialize the dictionary with all possible character values. For bytes, the dictionary starts with 256 entries.

Decoder can only add Sc to the dictionary after decoding the next word since it does not know c.

---

**LZW : Possible Problem**

The decoder is one step behind the encoder since it does not know c.

Consider a string ...SSc... where S[0] = c

**The encoder:**
- Outputs the id_1 for S and enters Sc into dictionary creating a new id_2 for Sc.
- Outputs the id_2 for Sc

**The decoder:**
- Inputs id_1, looks it up and outputs S
- Inputs id_2, looks it up, but not in the dictionary

What do we do?
**LZW: Encoding Example**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Dict.</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b a a c a b c a b</td>
<td>112</td>
<td>256=aa</td>
</tr>
<tr>
<td>a b a a c a b c a b</td>
<td>112</td>
<td>257=ab</td>
</tr>
<tr>
<td>a b a a c a b c a b</td>
<td>113</td>
<td>258=ba</td>
</tr>
<tr>
<td>a b a a c a b c a b</td>
<td>256</td>
<td>259=aac</td>
</tr>
<tr>
<td>a b a a c a b c a b</td>
<td>114</td>
<td>260=ca</td>
</tr>
<tr>
<td>a b a a c a b c a b</td>
<td>257</td>
<td>261=abc</td>
</tr>
<tr>
<td>a b a a c a b c a b</td>
<td>260</td>
<td>262=cab</td>
</tr>
</tbody>
</table>

**LZW: Decoding Example**

<table>
<thead>
<tr>
<th>Input</th>
<th>Dict</th>
</tr>
</thead>
<tbody>
<tr>
<td>112</td>
<td>a b a a c a b c a b</td>
</tr>
<tr>
<td>112</td>
<td>a b a a c a b c a b</td>
</tr>
<tr>
<td>113</td>
<td>a b a a c a b c a b</td>
</tr>
<tr>
<td>256</td>
<td>a a b a a c a b c a b</td>
</tr>
<tr>
<td>257</td>
<td>a a b a a c a b c a b</td>
</tr>
<tr>
<td>258</td>
<td>a a b a a c a b c a b</td>
</tr>
<tr>
<td>114</td>
<td>a a b a a c a b c a b</td>
</tr>
<tr>
<td>259</td>
<td>a a b a a c a b c a b</td>
</tr>
<tr>
<td>260</td>
<td>a a b a a c a b c a b</td>
</tr>
<tr>
<td>261</td>
<td>a a b a a c a b c a b</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)

**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)**

Adapts 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 performed poorly in terms of compression. More modern versions (e.g. gzip) do use probability coding as "second pass" and compress much better. The algorithms are becoming outdated, but ideas are used in many of the newer algorithms.

**Compression Outline**

- **Introduction**: Lossy vs. Lossless, Benchmarks, ...
- **Information Theory**: Entropy, etc.
- **Probability Coding**: Huffman + Arithmetic Coding
- **Applications of Probability Coding**: PPM + others
- **Lempel-Ziv Algorithms**: LZ77, gzip, compress, ...
- **Other Lossless Algorithms**:
  - Burrows-Wheeler
  - ACB
- **Lossy algorithms for images**: JPEG, MPEG, ...
- **Compressing graphs and meshes**: BBK

**Burrows-Wheeler**

Currently best "balanced" algorithm for text

Breaks file into fixed-size blocks and encodes each block separately.

**For each block:**

- Sort each character by its full context. This is called the block sorting transform.
- Use move-to-front transform 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>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ecode</td>
<td>$d_1$</td>
<td>deced</td>
</tr>
<tr>
<td>coded</td>
<td>$e_2$</td>
<td>coded</td>
</tr>
<tr>
<td>odede</td>
<td>$c_3$</td>
<td>decod</td>
</tr>
<tr>
<td>dedec</td>
<td>$o_4$</td>
<td>dedec</td>
</tr>
<tr>
<td>edecod</td>
<td>$d_5$</td>
<td>ecode</td>
</tr>
<tr>
<td>decod</td>
<td>$e_6$</td>
<td>edeco</td>
</tr>
</tbody>
</table>

**Sort Context**

$\leftarrow$
**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 sketch:** Since the chars have equal value in the most-significant-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.

<table>
<thead>
<tr>
<th>Context</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>dedec3</td>
<td>o4</td>
</tr>
<tr>
<td>coded1</td>
<td>e2</td>
</tr>
<tr>
<td>decod5</td>
<td>e6</td>
</tr>
<tr>
<td>odede2</td>
<td>c3</td>
</tr>
<tr>
<td>edode2</td>
<td>d1</td>
</tr>
<tr>
<td>edeco</td>
<td>d5</td>
</tr>
</tbody>
</table>

---

**Burrows-Wheeler: Decoding**

Consider dropping all but the last character of the context.

- What follows the \( a \) with an arrow?
- What follows the first \( b \)?
- What is the whole string?

<table>
<thead>
<tr>
<th>Context</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>b a</td>
</tr>
<tr>
<td>b a</td>
<td>c a</td>
</tr>
</tbody>
</table>

---

**Burrows-Wheeler: Decoding**

What about now?

<table>
<thead>
<tr>
<th>Output Rank</th>
<th>c ( \leftarrow ) 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>4</td>
</tr>
<tr>
<td>b</td>
<td>5</td>
</tr>
<tr>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>3</td>
</tr>
</tbody>
</table>

Can also use the "rank". The "rank" is the position of a character if it were sorted using a stable sort.

---

**Burrows-Wheeler Decode**

Function \( BW_{-}\text{Decode}(In, \text{Start}, n) \)

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

Rank gives position of each char in sorted order.
### Decoding Example

<table>
<thead>
<tr>
<th>$S$</th>
<th>$\text{Sort}(S)$</th>
<th>$\text{Rank}(S)$</th>
<th>$\text{Out}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_4$</td>
<td>$c_3$</td>
<td>6</td>
<td>$e_6$</td>
</tr>
<tr>
<td>$e_2$</td>
<td>$d_1$</td>
<td>4</td>
<td>$d_1$</td>
</tr>
<tr>
<td>$e_6$</td>
<td>$d_3$</td>
<td>5</td>
<td>$e_2$</td>
</tr>
<tr>
<td>$c_3$</td>
<td>$e_2$</td>
<td>1</td>
<td>$c_3$</td>
</tr>
<tr>
<td>$d_1$</td>
<td>$e_6$</td>
<td>2</td>
<td>$o_4$</td>
</tr>
<tr>
<td>$d_5$</td>
<td>$o_4$</td>
<td>3</td>
<td>$d_5$</td>
</tr>
</tbody>
</table>

#### Overview of Text Compression

PPM and Burrows-Wheeler both encode a single character based on the immediately preceding context.

LZ77 and LZ78 encode multiple characters based on matches found in a block of preceding text.

Can you mix these ideas, i.e., code multiple characters based on immediately preceding context?

- BZ does this, but they don’t give details on how it works – current best compressor
- ACB also does this – close to best

### ACB (Associate Coder of Buyanovsky)

Keep dictionary sorted by context (the last character is the most significant)

- Find longest match for context
- Find longest match for contents
- Code
  - Distance between matches in the sorted order
  - Length of contents match

Has aspects of Burrows-Wheeler, and LZ77