CPS 296.3: Algorithms in the Real World

Data Compression III

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 (Not covered in class)
Other Lossless Algorithms: Burrows-Wheeler
Lossy algorithms for images: JPEG, MPEG, ...
Compressing graphs and meshes: BBK

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
  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

a | a | c | a | a | c | a | b | c | a | b | a | c

Dictionary
(previously coded)

Lookahead
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 that starts 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**

```
LZ77: Example

<table>
<thead>
<tr>
<th>Buffer (size = 4)</th>
<th>Longest match</th>
<th>Dictionary (size = 6)</th>
<th>Next character</th>
</tr>
</thead>
</table>
```

```
| | a | a | c | a | c | b | a | b | a | a | c |
| (-, 0, a) |
| | a | a | c | a | c | b | b | c | a | b | a |
| (1, 1, c) |
| | a | a | c | a | c | b | c | b | a | a | a |
| (3, 4, b) |
| | a | a | c | a | c | a | b | c | b | a | a |
| (3, 3, a) |
| | a | a | c | a | c | a | b | a | a | a | c |
| (1, 2, c) |
```

**LZ77 Decoding**

Decoder keeps same dictionary window as encoder. For each message it looks it up in the dictionary and inserts a copy at the end of the string.

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

E.g. dict = abcd, codeword = $(2, 9, e)$

- Simply copy from left to right
  ```
  for (i = 0; i < length; i++)
      out[+cursor+i] = out[+cursor-offset+i]
  ```
- Out = abcdcdcdcdcdce

**LZ77 Optimizations used by gzip**

LZSS: Output one of the following two formats

- $(0, \text{position, length})$ or $(1, \text{char})$
- Uses the second format if length < 3.
  ```
<table>
<thead>
<tr>
<th>Buffer (size = 4)</th>
<th>Longest match</th>
<th>Dictionary (size = 6)</th>
<th>Next character</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>(1, a)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>(1, a)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>(1, c)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>(0, 3, 4)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
  ```

**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

... 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 ... 
... a | a | c | a | a | c | a | b | c | a | b | a | a | a | c ... 

Theory behind LZ77


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

Source entropy for a substring of length $n$ is given by:

$$H_n = \sum_{X \in A^n} p(X) \log \frac{1}{p(X)}$$

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

Comparison to Lempel-Ziv 78

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 (LZ78 is a trie)
- How it is extended (LZ78 only extends an existing entry by one character)
- How it is indexed (LZ78 indexes the nodes of the trie)
- How elements are removed

Lempel-Ziv Algorithms Summary

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 near 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: decode

Context "wraps" around. Last char is most significant.

<table>
<thead>
<tr>
<th>Context</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>edode</td>
<td>d</td>
</tr>
<tr>
<td>coded</td>
<td>e</td>
</tr>
<tr>
<td>odede</td>
<td>c</td>
</tr>
<tr>
<td>dedec</td>
<td>o</td>
</tr>
<tr>
<td>edeco</td>
<td>d</td>
</tr>
<tr>
<td>deco</td>
<td>e</td>
</tr>
<tr>
<td>decode</td>
<td>d</td>
</tr>
</tbody>
</table>

Context "wraps" around. Last char is most significant.

Sort Context

Output

c o
d e
d e
e c
e d

All rotations of input

Burrows Wheeler Decoding

Key Idea: Can construct entire sorted table from sorted column alone! First: sorting the output gives last column of context:

<table>
<thead>
<tr>
<th>Context</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>o</td>
</tr>
<tr>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td>e</td>
<td>c</td>
</tr>
<tr>
<td>e</td>
<td>d</td>
</tr>
<tr>
<td>o</td>
<td>d</td>
</tr>
</tbody>
</table>
Burrows Wheeler Decoding

Now sort pairs in last column of context and output column to form last two columns of context:

<table>
<thead>
<tr>
<th>Context</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>c o</td>
<td>ec o</td>
</tr>
<tr>
<td>d e</td>
<td>ed e</td>
</tr>
<tr>
<td>d e</td>
<td>od e</td>
</tr>
<tr>
<td>e c</td>
<td>de c</td>
</tr>
<tr>
<td>e d</td>
<td>de d</td>
</tr>
<tr>
<td>o d</td>
<td>co d</td>
</tr>
</tbody>
</table>

Output

<table>
<thead>
<tr>
<th>Context</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>dedec</td>
<td>o</td>
</tr>
<tr>
<td>coded</td>
<td>e</td>
</tr>
<tr>
<td>decod</td>
<td>e</td>
</tr>
<tr>
<td>ode</td>
<td>c</td>
</tr>
<tr>
<td>ecode</td>
<td>d ⇐</td>
</tr>
<tr>
<td>edeco</td>
<td>d</td>
</tr>
</tbody>
</table>

Message was d in first position, preceded in wrapped fashion by ecode: decode.

Optimization: Don’t really have to rebuild the whole context table.

What character comes after the first character, d₁?

Just have to find d₁ in last column of context and see what follows it: e₁.

Observation: instances of same character of output appear in same order in last column of context. (Proof is an exercise.)

The “rank” is the position of a character if it were sorted using a stable sort.
**Burrows-Wheeler Decode**

Function BW_Decode(In, Start, n)

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

Rank gives position of each char in sorted order.

**Decode Example**

<table>
<thead>
<tr>
<th>( S )</th>
<th>Rank(S)</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_5</td>
<td>3'</td>
<td></td>
</tr>
<tr>
<td>c_3</td>
<td>1'</td>
<td></td>
</tr>
<tr>
<td>e_6</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>e_2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>a_4</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

\( \text{Out} \downarrow = \{ a_4, e_2, c_3, o_4, d_5, e_6 \} \)

**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