Data Compression
LZ77

Jens Müller
Universität Stuttgart
2008-11-25
Outline

- Introduction – Principle of dictionary methods
- LZ77 Sliding window
- Examples
- Optimization
- Performance comparison
- Applications/Patents
Principle of dictionary methods

• Compressing multiple strings can be more efficient than compressing single symbols only (e.g. Huffman encoding).

• Strings of symbols are added to a dictionary. Later occurrences are referenced.

• Static dictionary: Entries are predefined and constant according to the application of the text

• Adaptive dictionary: Entries are taken from the text itself and created on-the-fly
LZ77

- Goes through the text in a **sliding window** consisting of a **search buffer** and a **look ahead buffer**.

  ![Search buffer and Look-ahead buffer illustration]

  ...this is a text that is being read through the window...

- The search buffer is used as dictionary.
- Sizes of these buffers are parameters of the implementation. Assumption: Patterns in text occur within range of the search buffer.
**LZ77 – Example (Encoding)**

Encoding of the string: `abracadabra`  
output tuple: (offset, length, symbol)

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td>d</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td>d</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td>d</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td>d</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td>d</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td>d</td>
<td>a</td>
</tr>
</tbody>
</table>

Output:
- (0,0,a)
- (0,0,b)
- (0,0,r)
- (3,1,c)
- (2,1,d)
- (7,4,d)

12 characters compressed into 6 tuples  
Compression rate: \((12 \times 8)/(6 \times (5+2+3)) = 96/60 = 1.6 = 60\%\).
Size of output

- Size for each output tuple (offset, length, symbol) when using fixed-length storage:
  \[ \lceil \log_2 S \rceil + \lceil \log_2 (S + L) \rceil + \lceil \log_2 A \rceil \]

  where \( S \) is the length of the search buffer, \( L \) the length of the look ahead window, \( A \) the size of the alphabet.

- Why \( S+L \) and not only \( S \)? See next slide.

- Worst case if no symbol repeats in the search buffer:
  Blow up of \( n \lceil \log_2 S \rceil + \lceil \log_2 (S + L) \rceil + \lceil \log_2 A \rceil \) instead of \( n \lceil \log_2 A \rceil \)
## Encoding reaches into look-ahead buffer

### Special case

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>he said:</td>
<td>H</td>
<td>A</td>
<td>H</td>
<td>A</td>
<td>H</td>
<td>A</td>
<td>A!</td>
<td>(0,0,H)</td>
</tr>
<tr>
<td>he said:</td>
<td>H</td>
<td>A</td>
<td>H</td>
<td>A</td>
<td>H</td>
<td>A</td>
<td>H!</td>
<td>(0,0,A)</td>
</tr>
<tr>
<td>...esaid:</td>
<td>H</td>
<td>A</td>
<td>H</td>
<td>A</td>
<td>H</td>
<td>A</td>
<td>H</td>
<td>(2,4,H)</td>
</tr>
<tr>
<td>...d:</td>
<td>H</td>
<td>A</td>
<td>H</td>
<td>A</td>
<td>H</td>
<td>A</td>
<td>!</td>
<td>(2,1,!)</td>
</tr>
<tr>
<td>...HA</td>
<td>H</td>
<td>A</td>
<td>H</td>
<td>A</td>
<td>H</td>
<td>A</td>
<td>!</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Search buffer</th>
<th>Look-ahead buffer</th>
</tr>
</thead>
</table>

Jens Müller- IPVS – Universität Stuttgart
while look-ahead buffer is not empty
    go backwards in search buffer to find longest match of the look-ahead buffer

    if match found
        print: (offset from window boundary, length of match, next symbol in look-ahead buffer);
        shift window by length+1;
    else
        print: (0, 0, first symbol in look-ahead buffer);
        shift window by 1;
    fi
end while
## Example (Decoding)

<table>
<thead>
<tr>
<th>input</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0,a)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>(0,0,b)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>(0,0,r)</td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td>b</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>(3,1,c)</td>
<td></td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>(2,1,d)</td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td>c</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>(7,4,d)</td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td>d</td>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>

Note: The decoding process involves filling in the characters according to the specified rules.
Decoding – Pseudo code algorithm

for each token (offset, length, symbol)
    if offset = 0 then
        print symbol;
    else
        go reverse in previous output by offset characters and copy character wise for length symbols;
        print symbol;
    fi
next

LZ77 is asymmetric, encoding is more difficult than decoding as it needs to find the longest match.
Optimizations

Successors following LZ77 used different optimizations:

• Use variable size offset and length fields in the tuples instead of fixed-length. Better if small offsets and sizes prevail.

• Don‘t output a (0,0,x) token when character is not found but instead differentiate using a flag-bit: 0|x or 1|o,l

• Use better suited data structure (e.g. tree, hash set) for the buffers. This allows faster search and/or larger buffers.

• Additional Huffman coding of tuples/references.

-> LZSS, Lzb, LZH, LZR, LZFG, LZMA, Deflate, …
Performance

(From Bell/Cleary/Witten: Text Compression)
Applications, Patents

Unlike **LZ78**, **LZ77** has not been patented. This may be a reason why its successors basing on LZ77 are so widely used:

**Deflate** is a combination of LZSS together with Huffman encoding and uses a window size of 32kB.

This algorithm is open source and used in what is widely known as ZIP compression (although the ZIP format itself is only a container format, like AVI and can be used with several algorithms), and by the formats PNG, TIFF, PDF and many others.
References


