Joint Source-Channel 
LZ'77 Coding

Stefano Lonardi
University of California, Riverside

Wojciech Szpankowski
Purdue University, West Lafayette

Source vs. Channel coding

• Source coding: represent the source information with the minimum of symbols
• Channel coding: represent the source information in a manner that minimizes the error probability in decoding
Problem definition

• How to achieve joint source and channel coding in LZ’77 (by adding error resiliency)
  – without significantly degrading the compression performance,
  – and keeping backward compatibility with the original LZ’77?

Encoding

“Dear Bob, How are you doing today? …”  →  LZRS’77  →  T.gz
Dear Bob, How are you doing today? ...
Roadmap

• We will show how to obtain extra redundant bits from LZ'77

• We will show how to achieve error resiliency in LZ'77

LZ'77: which of these pointers do we choose?
By choosing one of these pointers we are recovering two extra redundant bits. Note that we are not changing LZ’77

Extra bits recovering

• Definition: a LZ’77 phrase has multiplicity $q$ if has exactly $q$ matches in the history
• Given a phrase with multiplicity $q$, we can recover $\lfloor \log_2 q \rfloor$ bits
Average case analysis

- **Theorem**: Let $Q_n$ be the random variable associated with the multiplicity $q$ of a phrase in a string of length $n$. For a Markov source
  
  $E[Q_n] = O(1)$

  as $n \to \infty$
Recent results

- **Theorem**: For memoryless sources

\[
E[Q_n] = \frac{1}{H} + \text{small fluctuations}
\]

\[
P[Q_n = k] = \frac{p(1-p)^k + (1-p)p^k}{kH}
\]

where \( H \) is the entropy of the source, and \( p \) is the probability of generating a “0”

---

**Number of bits recovered**

Remark: more bits can be recovered by relaxing the greediness
Reed Solomon codes

- RS codes are block-based error correcting codes (BCH family)
- $RS(a,b)$ code
  - $a=2^s-1$, where $s$ is the datum size
  - has $(a-b)$ “parity” bits
  - can correct up to $(a-b)/2$ errors
- We used $RS(255,255-2e)$, which can correct up to $e$ errors

LZRS’77 encoder (off-line)

- **compress** the file with LZ’77
- **break** the compressed file in blocks $B_1,...,B_m$ of size $255-2e$
- **for** $i\leftarrow m$ **downto** 2
  - **encode** with $RS(255,255-2e)$ block $B_i$
  - **embed** the extra $2e$ parity bits in the pointers of block $B_{i-1}$
- **encode** with $RS(255,255-2e)$ block $B_1$
- **store** the extra parity bits at the beginning of the file
LZRS’77 encoding

LZRS’77 decoder (on-line)

- (assume $RS_i$ are the $2e$ parity bits for $B_i$)
- **decode** and **correct** block $B_1+RS_1$
- **decompress** block $B_1$ and **recover** $RS_2$
- for $i=2$ to $m$
  - **decode** and **correct** block $B_i+RS_i$
  - **decompress** block $B_i$ and **recover** $RS_{i+1}$
Experiments: gzip

- gzip issues pointers in a sliding window of 32Kbytes (typically)
- The length of phrases is represented by 8 bits (3-258)
- Strings smaller than 3 symbols are encoded as literals

gzip

- gzip always chooses the most “recent” occurrence of the longest prefix

“…the hash chains are searched starting from the most recent strings, to favor small distances and thus take advantage of the Huffman coding…”
“Hacking” gzip

• We modified gzip-1.2.4 to evaluate the potential degradation of compression performance due to changing the rule of choosing always the most “recent” occurrence

• As a preliminary experiment, we simply chose one pointer at random

### gzip vs. gzipS

<table>
<thead>
<tr>
<th>file size</th>
<th>gzip</th>
<th>gzipS</th>
<th>file</th>
<th>bytes embedded</th>
</tr>
</thead>
<tbody>
<tr>
<td>111,261</td>
<td>39,473</td>
<td>39,511</td>
<td>bib</td>
<td>1,721</td>
</tr>
<tr>
<td>768,771</td>
<td>333,776</td>
<td>336,256</td>
<td>book1</td>
<td>14,524</td>
</tr>
<tr>
<td>610,856</td>
<td>228,321</td>
<td>228,242</td>
<td>book2</td>
<td>10,361</td>
</tr>
<tr>
<td>102,400</td>
<td>69,478</td>
<td>71,168</td>
<td>geo</td>
<td>4,101</td>
</tr>
<tr>
<td>377,100</td>
<td>155,290</td>
<td>156,150</td>
<td>news</td>
<td>5,950</td>
</tr>
<tr>
<td>21,504</td>
<td>10,584</td>
<td>10,783</td>
<td>obj1</td>
<td>353</td>
</tr>
<tr>
<td>246,814</td>
<td>89,467</td>
<td>89,757</td>
<td>obj2</td>
<td>3,628</td>
</tr>
<tr>
<td>53,161</td>
<td>20,110</td>
<td>20,204</td>
<td>paper1</td>
<td>937</td>
</tr>
<tr>
<td>82,199</td>
<td>32,529</td>
<td>32,507</td>
<td>paper2</td>
<td>1,551</td>
</tr>
<tr>
<td>46,526</td>
<td>19,450</td>
<td>19,567</td>
<td>paper3</td>
<td>893</td>
</tr>
<tr>
<td>13,286</td>
<td>5,853</td>
<td>5,898</td>
<td>paper4</td>
<td>249</td>
</tr>
<tr>
<td>11,954</td>
<td>5,252</td>
<td>5,294</td>
<td>paper5</td>
<td>210</td>
</tr>
<tr>
<td>38,105</td>
<td>14,433</td>
<td>14,506</td>
<td>paper6</td>
<td>738</td>
</tr>
<tr>
<td>513,216</td>
<td>62,357</td>
<td>61,259</td>
<td>pic</td>
<td>3,025</td>
</tr>
<tr>
<td>39,611</td>
<td>14,510</td>
<td>14,660</td>
<td>progc</td>
<td>730</td>
</tr>
<tr>
<td>71,646</td>
<td>18,310</td>
<td>18,407</td>
<td>progl</td>
<td>1,106</td>
</tr>
<tr>
<td>49,379</td>
<td>12,532</td>
<td>12,572</td>
<td>progp</td>
<td>741</td>
</tr>
<tr>
<td>93,695</td>
<td>22,178</td>
<td>22,098</td>
<td>trans</td>
<td>1,201</td>
</tr>
</tbody>
</table>
Error correction (simulation)

- We chose $e=1$, $e=2$ and $b=10$, $b=100$
- For $b$ blocks, we injected $1, \ldots, b$
  uniformly distributed errors
- We measured the number of times that
  the file was decoded correctly (out of a
  few hundreds simulations)

Error-correction

![Graph showing probability of the file incorrectly decoded vs number of injected errors]
Findings

• Method to recover extra redundant bits from LZ’77
• Extra bits allow to incorporate error resiliency in LZ’77
  – backward-compatible (deployment without disrupting service)
  – compression degradation due to the extra bits is almost negligible