Efficient Sequential Algorithms, Comp309

University of Liverpool

2010–2011
Module Organiser, Igor Potapov
Supplementary Material: String Algorithms.
We will now look at the Lempel-Ziv Welch compression algorithm, which is a lossless compression algorithm that does particularly well on data with repetitions.

A useful feature of LZW compression is that the dictionary is built adaptively during encoding. The dictionary does not need to be passed with the compressed text — the decoding algorithm produces the same dictionary from the compressed text.
We will now look at the Lempel-Ziv Welch compression algorithm, which is a lossless compression algorithm that does particularly well on data with repetitions.

A useful feature of LZW compression is that the dictionary is built adaptively during encoding. The dictionary does not need to be passed with the compressed text — the decoding algorithm produces the same dictionary from the compressed text.
The original paper that describes the LZW algorithm is:


This paper describes an improvement to a compression method introduced by Ziv and Lempel in 1977 and 1978.

LZW and variants have been used in popular software such as Unix compress and GIF compression.
The original paper that describes the LZW algorithm is:


This paper describes an improvement to a compression method introduced by Ziv and Lempel in 1977 and 1978.

LZW and variants have been used in popular software such as Unix compress and GIF compression.
Sources

There is a lot of information about LZW on the web. See, for example, Wikipedia, or the nice animation at


Also, see Dave Marshall’s notes.

http://www.cs.cf.ac.uk/Dave/Multimedia
Compression

The dictionary is initialised so that there is a codeword for every extended ASCII character.

<table>
<thead>
<tr>
<th>character</th>
<th>code word</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>A</td>
<td>65</td>
</tr>
<tr>
<td>B</td>
<td>66</td>
</tr>
<tr>
<td>C</td>
<td>67</td>
</tr>
<tr>
<td>D</td>
<td>68</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>□</td>
<td>255</td>
</tr>
</tbody>
</table>
The compression algorithm

Initialise dictionary
\( w \leftarrow \text{NIL} \)

while there is a character to read
\( k \leftarrow \text{next character in text} \)
If \( wk \) is in the dictionary
\( w \leftarrow wk \)
Else
Add \( wk \) to the dictionary
Output the code for \( w \)
\( w \leftarrow k \)
Output the code for \( w \)
Example

\[ T = ABACABA \]

<table>
<thead>
<tr>
<th>w</th>
<th>k</th>
<th>rest of text</th>
<th>dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIL</td>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>BACABA</td>
<td>AB 256 65</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>ACABA</td>
<td>BA 257 66</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>CABA</td>
<td>AC 258 65</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
<td>ABA</td>
<td>CA 259 67</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>BA</td>
<td></td>
</tr>
<tr>
<td>AB</td>
<td>A</td>
<td>ABA</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Initialise dictionary
\[ w \leftarrow \text{NIL} \]
while there is a char to read
\[ k \leftarrow \text{next char in text} \]
If \( wk \) is in the dictionary
\[ w \leftarrow wk \]
Else
Add \( wk \) to dictionary
Output code for \( w \)
\[ w \leftarrow k \]
Output the code for \( w \)
The Decompression Algorithm

The basic decompression algorithm is as follows.

 Initialise dictionary
 \( c \leftarrow \text{first codeword} \)
 output the translation of \( c \)
 \( w \leftarrow c \)
 While there is a codeword to read
 \( c \leftarrow \text{next codeword} \)
 output the translation of \( c \)
 \( s \leftarrow \text{translation of } w \)
 \( k \leftarrow \text{first character of translation of } c \)
 Add \( sk \) to dictionary
 \( w \leftarrow c \)
Example

Code = 65, 66, 65, 67, 256, 65

<table>
<thead>
<tr>
<th>c</th>
<th>w</th>
<th>s</th>
<th>k</th>
<th>dictionary</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>65</td>
<td>A</td>
<td>B</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>66</td>
<td>66</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>65</td>
<td>65</td>
<td>B</td>
<td>A</td>
<td>AB</td>
<td>B</td>
</tr>
<tr>
<td>67</td>
<td>67</td>
<td>A</td>
<td>C</td>
<td>BA</td>
<td>A</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
<td>C</td>
<td>A</td>
<td>AC</td>
<td>C</td>
</tr>
<tr>
<td>65</td>
<td>65</td>
<td>AB</td>
<td>A</td>
<td>CA</td>
<td>AB</td>
</tr>
<tr>
<td>65</td>
<td>65</td>
<td>AB</td>
<td>A</td>
<td>ABA</td>
<td>A</td>
</tr>
</tbody>
</table>

Initialise dictionary  
$c \leftarrow$ first codeword  
output the translation of $c$

While there is a codeword to read

$c \leftarrow$ next codeword
output the translation of $c$

$s \leftarrow$ translation of $w$

$k \leftarrow$ first character of translation of $c$

Add $sk$ to dictionary

$w \leftarrow c$
Refinement

The decoding algorithm as stated does not always work as it fails if $c$ is not in the dictionary.
### Example

<table>
<thead>
<tr>
<th>w</th>
<th>k</th>
<th>dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIL</td>
<td>A</td>
<td>AB</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>AB</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>CA</td>
</tr>
<tr>
<td>AB</td>
<td>A</td>
<td>ABA</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>ABA</td>
</tr>
<tr>
<td>AB</td>
<td>A</td>
<td>ABA</td>
</tr>
<tr>
<td>ABA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Initialise dictionary

\[ w \leftarrow \text{NIL} \]

while there is a char to read

\[ k \leftarrow \text{next char in text} \]

If \( wk \) is in the dictionary

\[ w \leftarrow wk \]

Else

Add \( wk \) to dictionary

Output code for \( w \)

\[ w \leftarrow k \]

Output the code for \( w \)
Example

Code = 65,66,67,256,259

<table>
<thead>
<tr>
<th>c</th>
<th>w</th>
<th>s</th>
<th>k</th>
<th>...</th>
<th>...</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>65</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>65</td>
<td>A</td>
</tr>
<tr>
<td>65</td>
<td>66</td>
<td>A</td>
<td>B</td>
<td>AB</td>
<td>66</td>
<td>B</td>
</tr>
<tr>
<td>66</td>
<td>66</td>
<td>B</td>
<td>C</td>
<td>AB</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>66</td>
<td>67</td>
<td>B</td>
<td>C</td>
<td>BC</td>
<td>257</td>
<td>C</td>
</tr>
<tr>
<td>67</td>
<td>67</td>
<td>C</td>
<td>A</td>
<td>BC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>256</td>
<td>C</td>
<td>A</td>
<td>CA</td>
<td>258</td>
<td>AB</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
<td>C</td>
<td>A</td>
<td>CA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>259</td>
<td>256</td>
<td>C</td>
<td>A</td>
<td>CA</td>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>

Initialise dictionary
\( c \leftarrow \) first codeword
output the translation of \( c \)
\( w \leftarrow c \)
While there is a codeword to read
\( c \leftarrow \) next codeword
output the translation of \( c \)
\( s \leftarrow \) translation of \( w \)
\( k \leftarrow \) first character of translation of \( c \)
Add \( sk \) to dictionary
\( w \leftarrow c \)
The problem arises when the dictionary has $cs$ in the dictionary for a character $c$ and a string $s$ and then the input contains $cscsc$.

The decompression algorithm can be modified to deal with this case.
The problem arises when the dictionary has \textit{cs} in the dictionary for a character \textit{c} and a string \textit{s} and then the input contains \textit{cscsc}.

The decompression algorithm can be modified to deal with this case.
Initialise dictionary
\[ c \leftarrow \text{first codeword} \]
output the translation of \( c \)
\[ w \leftarrow c \]
While there is a codeword to read
\[ c \leftarrow \text{next codeword} \]
\[ s \leftarrow \text{translation of } w \]
If \( c \) is in dictionary
\[ k \leftarrow \text{first character of translation of } c \]
output the translation of \( c \)
Else (* \( c \) is the code for what we add here*)
\[ k \leftarrow \text{first character of } s \]
output \( sk \)
Add \( sk \) to dictionary
\[ w \leftarrow c \]
Example

Code = 65,66,67,256,259

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>w</td>
<td>s</td>
<td>k</td>
</tr>
<tr>
<td>65</td>
<td>65</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>66</td>
<td>66</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>67</td>
<td>67</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
<td>AB</td>
<td>A</td>
</tr>
<tr>
<td>259</td>
<td>259</td>
<td>ABA</td>
<td>259</td>
</tr>
</tbody>
</table>

Initialise dictionary

\( c \leftarrow \) first codeword

output the translation of \( c \)

\( w \leftarrow c \)

While there is a codeword

\( c \leftarrow \) next codeword

\( s \leftarrow \) translation of \( w \)

If \( c \) is in dictionary

\( k \leftarrow 1st \) char of trans \( c \)

output trans \( c \)

Else (\(* c \) is next added*)

\( k \leftarrow \) first character of \( s \)

output \( sk \)

Add \( sk \) to dictionary

\( w \leftarrow c \)
There are lots of interesting implementation issues. For example, what if the dictionary runs out of space?

Also, if we start re-using dictionary space, what data structure do we use to make dictionary access efficient?

GIF compression solves the problem of dictionary overflow by having variable-length codes. We will not cover the details.