Compression

The LZW compression algorithm in its simplest form is shown in Figure 1. A quick examination of the algorithm shows that LZW is always trying to output codes for strings that are already known. And each time a new code is output, a new string is added to the string table.

*Routine LZW\_COMPRESS*

STRING = get input character

WHILE there are still input characters DO

CHARACTER = get input character

IF STRING+CHARACTER is in the string table then

STRING = STRING+character

ELSE

output the code for STRING

add STRING+CHARACTER to the string table

STRING = CHARACTER

END of IF

END of WHILE

output the code for STRING

**Figure 1 — The Compression Algorithm**

A sample string used to demonstrate the algorithm is shown in Figure 2. The input string is a short list of English words separated by the ‘/’ character. Stepping through the start of the algorithm for this string, you can see that the first pass through the loop, a check is performed to see if the string "/W" is in the table. Since it isn’t, the code for ‘/’ is output, and the string "/W" is added to the table. Since we have 256 characters already defined for codes 0-255, the first string definition can be assigned to code 256. After the third letter, ‘E’, has been read in, the second string code, "WE" is added to the table, and the code for letter ‘W’ is output. This continues until in the second word, the characters ‘/’ and ‘W’ are read in, matching string number 256. In this case, the code 256 is output, and a three character string is added to the string table. The process continues until the string is exhausted and all of the codes have been output.

|  |  |  |  |
| --- | --- | --- | --- |
| Input String = /WED/WE/WEE/WEB/WET | | | |
| Character Input | Code Output | New code value | New String |
| /W | / | 256 | /W |
| E | W | 257 | WE |
| D | E | 258 | ED |
| / | D | 259 | D/ |
| WE | 256 | 260 | /WE |
| / | E | 261 | E/ |
| WEE | 260 | 262 | /WEE |
| /W | 261 | 263 | E/W |
| EB | 257 | 264 | WEB |
| / | B | 265 | B/ |
| WET | 260 | 266 | /WET |
| EOF | T |  |  |

**Figure 2 — The Compression Process**

The sample output for the string is shown in Figure 2 along with the resulting string table. As can be seen, the string table fills up rapidly, since a new string is added to the table each time a code is output. In this highly redundant input, 5 code substitutions were output, along with 7 characters. If we were using 9 bit codes for output, the 19 character input string would be reduced to a 13.5 byte output string. Of course, this example was carefully chosen to demonstrate code substitution. In real world examples, compression usually doesn’t begin until a sizable table has been built, usually after at least one hundred or so bytes have been read in.

Decompression

The companion algorithm for compression is the decompression algorithm. It needs to be able to take the stream of codes output from the compression algorithm, and use them to exactly recreate the input stream. One reason for the efficiency of the LZW algorithm is that it does not need to pass the string table to the decompression code. The table can be built exactly as it was during compression, using the input stream as data. This is possible because the compression algorithm always outputs the STRING and CHARACTER components of a code before it uses it in the output stream. This means that the compressed data is not burdened with carrying a large string translation table.

*Routine LZW\_DECOMPRESS*

Read OLD\_CODE

output OLD\_CODE

WHILE there are still input characters DO

Read NEW\_CODE

STRING = get translation of NEW\_CODE

output STRING

CHARACTER = first character in STRING

add OLD\_CODE + CHARACTER to the translation table

OLD\_CODE = NEW\_CODE

END of WHILE

**Figure 3 — The Decompression Algorithm**

The algorithm is shown in Figure 3. Just like the compression algorithm, it adds a new string to the string table each time it reads in a new code. All it needs to do in addition to that is translate each incoming code into a string and send it to the output.

Figure 4 shows the output of the algorithm given the input created by the compression earlier in the article. The important thing to note is that the string table ends up looking exactly like the table built up during compression. The output string is identical to the input string from the compression algorithm. Note that the first 256 codes are already defined to translate to single character strings, just like in the compression code.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input Codes: / W E D 256 E 260 261 257 B 260 T | | | | |
| Input/ NEW\_CODE | OLD\_CODE | STRING/ Output | CHARACTER | New table entry |
| / | / | / |  |  |
| W | / | W | W | 256 = /W |
| E | W | E | E | 257 = WE |
| D | E | D | D | 258 = ED |
| 256 | D | /W | / | 259 = D/ |
| E | 256 | E | E | 260 = /WE |
| 260 | E | /WE | / | 261 = E/ |
| 261 | 260 | E/ | E | 262 = /WEE |
| 257 | 261 | WE | W | 263 = E/W |
| B | 257 | B | B | 264 = WEB |
| 260 | B | /WE | / | 265 = B/ |
| T | 260 | T | T | 266 = /WET |

**Figure 4 — The Decompression Process**