Compression Algorithm

Overview of Lempel-Ziv Compression

Lempel-Ziv compression is a streaming compression algorithm in which the encoding of a particular portion of a data stream S depends in part on the values of an already-encoded portion of the stream. The stream is compressed into a series of tuples whose fields specify the next decoded portion of the stream in terms of a position and length in an already-decoded portion of the stream. This method is described in its purest form in Ziv and Lempel's "A Universal Algorithm for Sequential Data Compression" (IEEE Transactions on Information Theory, vol. IT-23 no. 3, 1977), but it requires some tweaking to be useful in practice. Windows modifies the basic algorithm in two ways:

1. Metadata is inserted into the compressed data stream to describe the format of the compressed data tuples; and
2. Ability to account for situations in which the size of the compressed data (with its associated metadata) actually exceeds the size of the uncompressed data.

Windows's Lempel-Ziv implementation is based on the original algorithm, but it defines a special buffer format to implement these features.

Buffer Format

The buffer produced by Windows's compression algorithm has the following grammatical structure:

```
<Buffer> ::= <Chunk> <Buffer> | <Chunk>
<Chunk> ::= <Compressed_chunk> |
            <Uncompressed_chunk> |
            End_of_buffer

<Uncompressed_chunk> ::= Chunk_header Uncompressed_data
<Compressed_chunk> ::= Chunk_header <Flag_group>
<Flag_group> ::= <Flag_data> <Flag_group> | <Flag_data>

<Flag_data> ::=
    Flag_byte <Data> <Data> <Data> <Data> <Data> <Data> <Data> <Data>
  | Flag_byte <Data> <Data> <Data> <Data> <Data> <Data> <Data>
  | Flag_byte <Data> <Data> <Data> <Data> <Data> <Data>
  | Flag_byte <Data> <Data> <Data> <Data> <Data>
  | Flag_byte <Data> <Data> <Data> <Data>
  | Flag_byte <Data> <Data> <Data>
  | Flag_byte <Data> <Data>
  | Flag_byte <Data>

<Data> ::= Literal | Compressed_word
```

In other words: A compressed data buffer consists of one or more chunks. A chunk may be either compressed or uncompressed, or it may denote the end of the buffer. If the chunk is uncompressed, it contains a chunk header followed by uncompressed data; if it is compressed, it contains a chunk header

followed by a series of one or more pieces of flagged data. Finally, a piece of flagged data consists of a flag byte followed by between zero and eight individual data elements.

The following sections discuss the structure of each of these grammatical elements, including constraints on their usage that are not expressed in the raw grammar.

Buffers and Chunks

A compressed buffer consists of a series of one or more compressed output chunks. Each chunk begins with a 16-bit header that has the following format:

- Bit 15 indicates whether the chunk's data is compressed.
- Bits [14:12] contain a signature indicating the format of the subsequent data.
- Bits [11:0] contain the size of the compressed chunk, minus three bytes.

Bit 15 indicates whether a chunk's data is compressed. If this bit is cleared, the chunk header is followed by uncompressed literal data. If this bit is set, the next byte of the chunk is the beginning of a *Flag_group* nonterminal that describes some compressed data.

Bits 14 down to 12 contain a signature value. This value must always be 3 (unless the header denotes the end of the compressed buffer).

Bits 11 down to 0 contain the size of the compressed chunk *minus* three bytes. This size otherwise includes the size of any metadata in the chunk, including the chunk header. If the chunk is uncompressed, the total amount of uncompressed data therein can be computed by adding 1 to this value (adding 3 bytes to get the total chunk size, then subtracting 2 bytes to account for the chunk header).

Input streams are compressed in units of 4 KB. When, in the process of creating a chunk, If at least 4 KB of data or the remainder of the input buffer is compressed, processing the current chunk is stopped and a new chunk is started. This 4 KB limit is important when determining the format of compressed data elements, discussed below.

Flag Groups

If a chunk is compressed, its chunk header is immediately followed by the first byte of a *Flag_group* nonterminal.

A flag group consists of a flag byte followed by zero or more data elements. Each data element is either a single literal byte or a two-byte compressed word. The individual bits of a flag byte, taken from low-order bits to high-order bits, specify the formats of the subsequent data elements (such that bit 0 corresponds to the first data element, bit 1 to the second, and so on). If the bit corresponding to a data element is set, the element is a two-byte compressed word; otherwise, it is a one-byte literal.

Not all of the bits in a flag byte may be used. To process compressed buffers, the compressed chunk size stored in the chunk header must be used to determine the position of the last valid byte in the chunk and must ignore flag bits that correspond to bytes that fall past the end of the chunk.

Data Elements

A data element may be either an uncompressed literal or a compressed word. An uncompressed literal is simply a byte of data that was not compressed and can therefore be treated as part of the uncompressed data stream. A compressed word is a two-byte value that contains a length and a displacement and whose format varies depending on the portion of the data that is being processed.

Each compressed word consists of a *D*-byte displacement in the high-order bits and an *L*-byte length in the low-order bits, subject to the constraints that 4 <= D <= 12, 4 <= L <= 12, and D + L = 16. The displacement in a compressed word is the difference between the current location in the uncompressed data (either the current read point when compressing or the current write point when decompressing) and the location of the uncompressed data corresponding to the compressed word, *minus one byte*. The length is the amount of uncompressed data that can be found at the appropriate displacement, *minus three bytes*. While using the compressed buffers, the stored displacement must be incremented by 1 and the stored length must be incremented by 3, to get the actual displacement and length.

For example, suppose that the input data for a given compression consists of the following stream:

F F G A A G F E D D E F F E E | F F G A A G F E D D E F E D D

Suppose further that all of the data prior to the vertical bar has already been compressed. The next 12 characters of the input stream match the first 12 characters of already-compressed data. Moreover, the distance from the current input pointer to the start of this matching string is 15 characters. This can be described with a <displacement, length> pair of <15, 12>.

When this data is decompressed later, decompression will produce the first portion of the input stream:

F F G A A G F E D D E F F E E |

The next data element will be a <15, 12> displacement-length pair. The start of the uncompressed data is 15 characters behind the last character in the already-uncompressed data, and the length of the data to read is 12 characters, so the decompression results in the following buffer:

F F G A A G F E D D E F F E E F F G A A G F E D D E F |

This matches the original data stream:

F F G A A G F E D D E F F E E F F G A A G F E D D E F E D D

The sizes of the displacement and length fields of a compressed word vary with the amount of uncompressed data in the current chunk that has already been processed. The format of a given compressed word is determined as follows: Let U be the amount of uncompressed data that has already been processed in the current chunk (either the amount that has been read when compressing data or the amount that has been written when decompressing data). Note that U depends on the offset from

the start of a chunk and not the offset from the beginning of the uncompressed data. Then let M be the largest value in [4…12] such that 2^M < U, or 4 if there is no such value. A compressed word then has format D = M and L = 16 – M, with the displacement occupying D high-order bits and the length occupying L low-order bits.

Lempel-Ziv compression does not require that the entirety of the data to which a compressed word refers actually be in the uncompressed buffer when the word is processed. In other words, it is not required that (U – displacement + length < U). Therefore, when processing a compressed word, data must be copied from the start of the uncompressed target region to the end—that is, the byte at (U – displacement) must be copied first, then (U – displacement + 1), and so on, since the compressed word may refer to data that will be written during decompression. The example below includes an example of this behavior.

Example

Consider the compression of the following null-terminated ANSI string:

```
F# F# G A A G F# E D D E F# F# E E F# F# G A A G F# E D D E F# E D D E
E F# D E F# G F# D E F# G F# E D E A F# F# G A A G F# E D D E F# E D D
```

Including the terminal NULL, this string has a length of 142 bytes. The algorithm, using the standard compression engine, produces the following output with a length of 59 bytes:

```
0x00000000: 38 b0 88 46 23 20 00 20
0x00000008: 47 20 41 00 10 a2 47 01
0x00000010: a0 45 20 44 00 08 45 01
0x00000018: 50 79 00 c0 45 20 05 24
0x00000020: 13 88 05 b4 02 4a 44 ef
0x00000028: 03 58 02 8c 09 16 01 48
0x00000030: 45 00 be 00 9e 00 04 01
0x00000038: 18 90 00
```

The compressed data is contained in a single chunk. The chunk header, interpreted as a 16-bit value, is 0xB038. Bit 15 is 1, so the chunk is compressed; bits 14 through 12 are the correct signature value (3); and bits 11 through 0 are decimal 56, so the size of the chunk is 59 bytes.

The next byte, 0x88, is a flag byte. Bits 0, 1, and 2 of this byte are all cleared, so the next three bytes are not compressed. They are 0x46 (‘F’), 0x23 (‘#’), and 0x20 (a space). The output stream now contains “F# “.

Bit 3 of the flag byte is set, however, so the next two bytes must be part of a compressed word; in this case, that word is 0x2000. Here, the offset from the start of the uncompressed data, U, is 3 bytes; there is no value M such that M >= 4 and 2^M < U, so the compressed word has 4 bits of displacement and 12 bits of length. The stored displacement is thus 2 (0010) and the stored length is 0 (0000 0000 0000); the actual displacement is therefore 3 (2 + 1 = 3) and the length is 3 (0 + 3 = 3). Thus, the next three characters of uncompressed data are “F# “, resulting in an uncompressed string of length 6: “F# F# “.

Bits 4 through 6 of the flag byte are clear, so the next three bytes are literals: 0x47 (‘G’), 0x20 (a space), and 0x41 (‘A’). The string is now “F# F# G A”. Bit 7 is set, so the next two bytes are a compressed word, 0x1000. The offset from the start of the chunk is 9 bytes, so the compressed word once again has 4 bits of displacement and 12 bits of length. The stored displacement is 1 (0001) and the stored length is 0 (0000 0000 0000); thus, the final displacement is 2 (1 + 1 = 2) and the final length is 3 (0 + 3 = 3). This is a case in which the current uncompressed length (9 bytes) minus the displacement plus the length (10 bytes) actually exceeds the amount of uncompressed data, so character-by-character copying from the beginning of the displaced region is important. The first character is a space, so the string is “F# F# G A “; the next is an A, so “F# F# G A A”; and the next character is the space that was just written, resulting in “F# F# G A A “.

The rest of the decompression proceeds similarly. The only other item of interest is the final flag byte, located at offset 0x37. This is the 56th byte of compressed data, so only three bytes remain; the flag byte is 0x01, so the next two bytes are a single compressed word and the final byte is a literal, 0x00. The remainder of the flag byte is then ignored as there is no data left in the buffer.