The Lempel Ziv Algorithm

Christina Zeeh
Seminar "Famous Algorithms"

January 16, 2003

The Lempel Ziv Algorithm is an algorithm for lossless data compression. It is not a single algorithm, but a whole family of algorithms, stemming from the two algorithms proposed by Jacob Ziv and Abraham Lempel in their landmark papers in 1977 and 1978. Lempel Ziv algorithms are widely used in compression utilities such as gzip, GIF image compression and the V.42 modem standard.

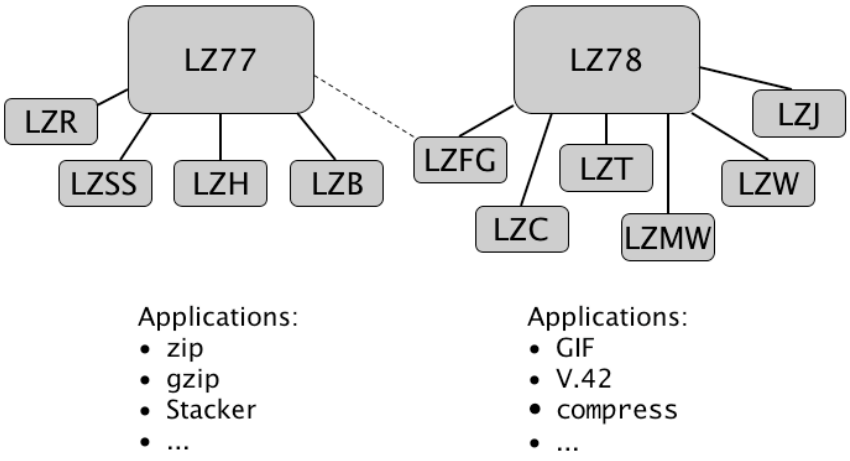


Figure 1: The Lempel Ziv Algorithm Family

This report shows how the two original Lempel Ziv algorithms, LZ77 and LZ78, work and also presents and compares several of the algorithms that have been derived from the original Lempel Ziv algorithms.

Contents

1 Introduction to Data Compression

Most data shows patterns and is subject to certain constraints. This is true for text, as well as for images, sound and video, just to name a few. Consider the following excerpt from "Lord of the Rings":

The eldest of these, and Bilbo's favourite, was young Frodo Baggins. When Bilbo was ninety-nine he adopted Frodo as his heir, and brought him to live at Bag End; and the hopes of the Sackville-Bagginses were finally dashed. Bilbo and Frodo happened to have the same birthday, September 22nd. 'You had better come and live here, Frodo my lad,' said Bilbo one day; 'and then we can celebrate our birthday-parties comfortably together.' At that time Frodo was still in his tweens, as the hobbits called the irresponsible twenties between childhood and coming of age at thirty-three.

Figure 2: Excerpt from Lord of the Rings

This text, as almost any other English text, contains letters that occur more often than other letters. For example, there are significantly more 'e' in this text than there are 'k'. Also notice that many substrings occur repeatedly in this text (all repeated strings of two or more characters have been colored red).

Data compression algorithms exploit such characteristics to make the compressed data smaller than the original data. In this report, only lossless compression algorithms – as opposed to so-called "lossy" compression algorithms – are being presented. Lossless compression ensures that the original information can be exactly reproduced from the compressed data.

Well-known lossless compression techniques include:

- *Run-length coding*: Replace strings of repeated symbols with a count and only one symbol. Example: `aaaaabbbbbbccccc` → `5a6b5c`
- *Statistical techniques*:
  - *Huffman coding*: Replace fixed-length codes (such as ASCII) by variable-length codes, assigning shorter codewords to the more frequently occuring symbols and thus decreasing the overall length of the data. When using variable-length codewords, it is desirable to create a (uniquely decipherable) prefix-code, avoiding the need for a separator to determine codeword boundaries. Huffman coding creates such a code.

- *Arithmetic coding*: Code message as a whole using a floating point number in an interval from zero to one.
- *PPM* (prediction by partial matching): Analyze the data and predict the probability of a character in a given context. Usually, arithmetic coding is used for encoding the data. PPM techniques yield the best results of statistical compression techniques.

The Lempel Ziv algorithms belong to yet another category of lossless compression techniques known as *dictionary coders*.

2 Dictionary Coding

Dictionary coding techniques rely upon the observation that there are correlations between parts of data (recurring patterns). The basic idea is to replace those repetitions by (shorter) references to a "dictionary" containing the original.

2.1 Static Dictionary

The simplest forms of dictionary coding use a static dictionary. Such a dictionary may contain frequently occurring phrases of arbitrary length, digrams (two-letter combinations) or n-grams. This kind of dictionary can easily be built upon an existing coding such as ASCII by using previously unused codewords or extending the length of the codewords to accommodate the dictionary entries.

A static dictionary achieves little compression for most data sources. The dictionary can be completely unsuitable for compressing particular data, thus resulting in an increased message size (caused by the longer codewords needed for the dictionary).

2.2 Semi-Adaptive Dictionary

The aforementioned problems can be avoided by using a semi-adaptive encoder. This class of encoders creates a dictionary custom-tailored for the message to be compressed. Unfortunately, this makes it necessary to transmit/store the dictionary together with the data. Also, this method usually requires two passes over the data, one to build the dictionary and another one to compress the data. A question arising with the use of this technique is how to create an optimal dictionary for a given message. It has been shown that this problem is NP-complete (vertex cover problem). Fortunately, there exist heuristic algorithms for finding near-optimal dictionaries.

2.3 Adaptive Dictionary

The Lempel Ziv algorithms belong to this third category of dictionary coders. The dictionary is being built in a single pass, while at the same time also encoding the data. As we will see, it is not necessary to explicitly transmit/store the dictionary because the decoder can build up the dictionary in the same way as the encoder while decompressing the data.

3 LZ77

3.1 Principle

So far we have always been talking about the dictionary as if it were some kind of data structure that is being filled with entries when the dictionary is being built. It turns out that it is not necessary to use an explicit dictionary.

Example: The data shown in figure 3 is to be encoded. The algorithm is working from left to right and has already encoded the left part (the string $E =$ "`abcbbacde`"). The string $S =$ "`bbadeaa`" is the the data yet to be encoded.

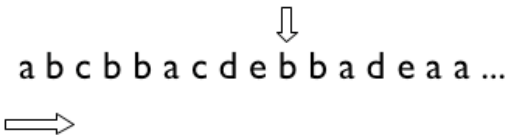


Figure 3: Example

First, the algorithm searches for the longest string in the encoded data E matching a prefix of S. In this particular case, the longest match is the string `"bba"` starting at the third position (counting from zero). Therefore it is possible to code the first two characters of S, `"bba"`, as a reference to the third and fourth character of the whole string. References are encoded as a fixed-length codeword consisting of three elements: position, length and first non-matching symbol. In our case, the codeword would be `33d`. In it, four characters have been coded with just one codeword. When the matches get longer, those coded references will consume significantly fewer space than, for example, coding everything in ASCII.

Probably you have already spotted the weakness of the outlined algorithm. What happens if the input is very long and therefore references

(and lengths) become very large numbers? Well, the previous example was not yet the actual LZ77 algorithm. The LZ77 algorithm employs a principle called *sliding-window*: It looks at the data through a window of fixed-size, anything outside this window can neither be referenced nor encoded. As more data is being encoded, the window slides along, removing the oldest encoded data from the view and adding new unencoded data to it. An example of this window is shown in figure 4.

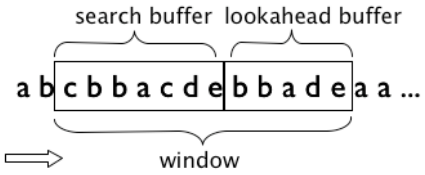


Figure 4: Sliding Window

The window is divided into a search buffer containing the data that has already been processed, and a lookahead buffer containing the data yet to be encoded.

3.2 The Algorithm

This is the actual LZ77 algorithm:

```
while (lookAheadBuffer not empty) {
  get a reference (position, length) to longest match;
  if (length > 0) {
    output (position, length, next symbol);
    shift the window length+1 positions along;
  } else {
    output (0, 0, first symbol in the lookahead buffer);
    shift the window 1 character along;
  }
}
```

If there is no match (length of the match is 0), the algorithm outputs *0 0* and the first symbol of the lookahead buffer.

The algorithms starts out with the lookahead buffer filled with the first symbols of the data to be encoded, and the search buffer filled with a predefined symbol of the input alphabet (zeros, for example).

3.3 Example

The following is the example given in the original LZ77 paper:

$S =$ `00101021021021202102120 0...` *(input string)*
$L_s = 9$ *(length of lookahead buffer)*
$n = 18$ *(window size)*

The search buffer is loaded with zeros and the lookahead buffer is loaded with the first 9 characters of S. The algorithm searches for the longest match in the search buffer. Since it is filled with zeros, any substring of length 2 can be used. In the example, the substring starting at the last position (8, if counting from 0) of the search buffer is being used. Note that the match extends into the lookahead buffer!

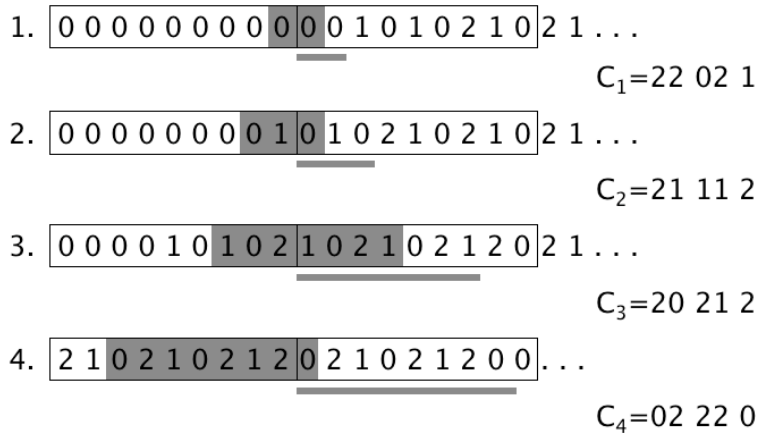


Figure 5: Encoding

Matches are encoded as codewords consisting of the position of the match, the length of the match and the first symbol following the prefix of the lookahead buffer that a match has been found for. Therefore, codewords need to have the length

$$L_c = log_\alpha(n - L_s) + log_\alpha(L_s) + 1$$

In the example the length is $L_s = log_3(9) + log_3(9) + 1 = 5$.

The first match is encoded as the codeword $C_1 = 22021$. 22 is the position of the match in radix-3 representation ($8_{10} = 22_3$). The two following positions represent the length of the match ($2_{10} = 2_3$, 02 because 2 positions are reserved for it according to the formula for L_c). The last element is the first symbol following the match, which in our case is 1.

The algorithm now shifts the window 3 positions to the right, resulting in the situation depicted in 2. This time, the algorithm finds the longest match at the 7th position of the search buffer, the length of the match is 3 because once again it is possible to extend the match into the lookahead buffer. The resulting codeword is $C_2 = 21102$. Steps 3 and 4 result in the codewords $C_3 = 20212$ and $C_4 = 02220$.

The decoder also starts out with a search buffer filled with zeros and reverses the process as shown in figure 6.

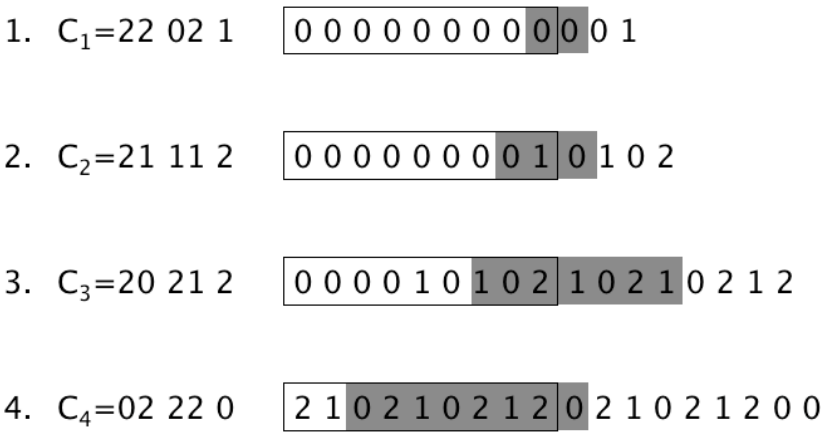


Figure 6: Decoding

If a match extended into the lookahead buffer, the decoder simply starts decoding and is then able to find the remaining part of the match in the already decoded part of the codeword.

3.4 Improvements

3.4.1 LZR

The LZR modification allows pointers to reference anything that has already been encoded without being limited by the length of the search buffer (window size exceeds size of expected input). Since the position and length values can be arbitrarily large, a variable-length representation is being used positions and lengths of the matches.

3.4.2 LZSS

The mandatory inclusion of the next non-matching symbol into each codeword will lead to situations in which the symbol is being explicitly coded

despite the possibility of it being part of the next match. Example: In `"abbca|caabb"`, the first match is a reference to `"ca"` (with the first non-matching symbol being `"a"`) and the next match then is `"bb"` while it could have been `"abb"` if there were no requirement to explicitly code the first non-matching symbol.

The popular modification by Storer and Szymanski (1982) removes this requirement. Their algorithm uses fixed-length codewords consisting of offset (into the search buffer) and length (of the match) to denote references. Only symbols for which no match can be found or where the references would take up more space than the codes for the symbols are still explicitly coded.

```
while( lookAheadBuffer not empty )
{
  get a pointer (position, match) to the longest match;
  if (length > MINIMUM_MATCH_LENGTH){
    output (POINTER_FLAG, position, length);
    shift the window length characters along;
  } else {
    output (SYMBOL_FLAG, first symbol of lookahead buffer);
    shift the window 1 character along;
  }
}
```

3.4.3 LZB

LZB uses an elaborate scheme for encoding the references and lengths with varying sizes.

3.4.4 LZH

The LZH implementation employs Huffman coding to compress the pointers.

3.5 Comparison

The following chart shows a comparison of the compression rates for the different LZ77 variants. Various files are used for testing (all values are taken from [2]):

- **bib**: List of bibliography entries (ASCII)
- **book**: Fiction and non-fiction book (ASCII), mean value

- **geo**: Geophysical data (32 bit numbers)
- **obj**: Executable files for VAX and Mac, mean value
- **paper**: Scientific papers (ASCII), mean value
- **pic**: Black and white bitmap image
- **term**: Terminal session (ASCII)

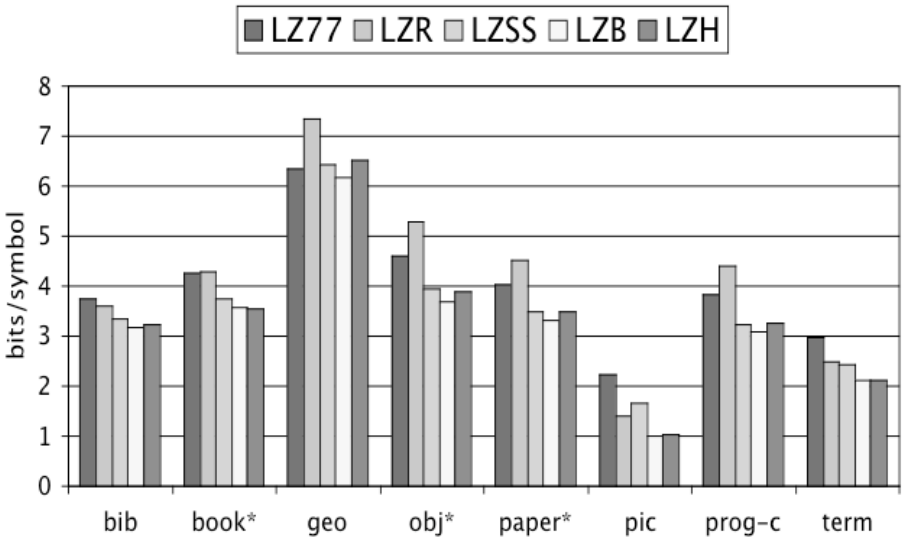


Figure 7: Compression rates for the LZ77 family

The compression rate is measured in bits/symbol, indicating how many bits are needed on average to encode a symbol (for binary files: symbol = byte).

4 LZ78

4.1 Principle

The LZ78 is a dictionary-based compression algorithm that maintains an explicit dictionary. The codewords output by the algorithm consist of two elements: an index referring to the longest matching dictionary entry and the first non-matching symbol.

In addition to outputting the codeword for storage/transmission, the algorithm also adds the index and symbol pair to the dictionary. When a symbol that not yet in the dictionary is encountered, the codeword has the index value 0 and it is added to the dictionary as well. With this method, the algorithm gradually builds up a dictionary.

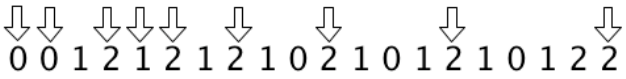
4.2 Algorithm

```
w := NIL;
while (there is input){
  K := next symbol from input;
  if (wK exists in the dictionary) {
    w := wK;
  } else {
    output (index(w), K);
    add wK to the dictionary;
    w := NIL;
  }
}
```

Note that this simplified pseudo-code version of the algorithm does not prevent the dictionary from growing forever. There are various solutions to limit dictionary size, the easiest being to stop adding entries and continue like a static dictionary coder or to throw the dictionary away and start from scratch after a certain number of entries has been reached. Those and more sophisticated approaches will be presented in section 4.4.

4.3 Example

The string $S =$ `00121212102101221011` is to be encoded. Figure 8 shows the encoding process.



#	entry	phrase	Output:	(ternary)
1	0	0	0 0	(0 0)
2	1+1	01	1 1	(1 1)
3	2	2	0 2	(0 2)
4	1	1	0 1	(00 1)
5	3+1	21	3 1	(10 1)
6	5+0	210	5 0	(12 0)
7	6+1	2101	6 1	(20 1)
8	7+2	21012	7 2	(21 2)
9	7+1	21011	7 1	(21 1)

Figure 8: Encoding

In the first step, `0` is encountered and added to the dictionary. The output is `00` because is no match (index `0`) and the first non-matching character is `0`. The encoder then proceeds to the second position, encountering `0`, which is already dictionary. The following `1` is not yet in the dictionary, so the encoder adds the string `01` to the dictionary (a reference to the first entry plus the symbol `1`) and outputs this pair. The next steps follow the same scheme until the end of the input is reached.

The decoding process is shown in figure 9. The decoder receives the reference `0 0`, with the index `0` indicating that a previously unknown symbol (`0`) needs to be added to the dictionary and to the uncompressed data. The next codeword is `1 1` resulting in the entry `01` (a reference to entry `1` plus the symbol `1`) being added to the dictionary and the string `01` appended to the uncompressed data. The decoder continues this way until all codewords have been decoded.

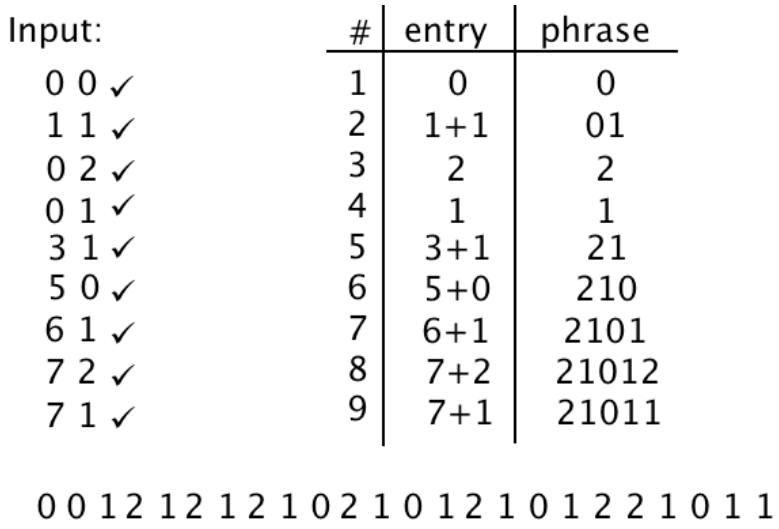


Figure 9: Decoding

4.4 Improvements

LZ78 has several weaknesses. First of all, the dictionary grows without bounds. Various methods have been introduced to prevent this, the easiest being to become either static once the dictionary is full or to throw away the dictionary and start creating a new one from scratch. There are also more sophisticated techniques to prevent the dictionary from growing unreasonably large, some of these will be presented in this chapter.

The dictionary building process of LZ78 yields long phrases only fairly late in the dictionary building process and only includes few substrings of the processed data into the dictionary. The inclusion of an explicitly coded symbol into every match may cause the next match to be worse than it could be if it were allowed to include this symbol.

4.4.1 LZW

This improved version of the original LZ78 algorithm is perhaps the most famous modification and is sometimes even mistakenly referred to as **the** Lempel Ziv algorithm. Published by Terry Welch in 1984, it basically applies the LZSS principle of not explicitly transmitting the next non-matching symbol to the LZ78 algorithm. The only remaining output of this improved algorithm are fixed-length references to the dictionary (indexes). Of course, we can't just remove all symbols from the output and add nothing elsewhere. Therefore the dictionary has to be initialized with all the symbols of the input alphabet and this initial dictionary needs to be made known to the decoder. The actual algorithm then proceeds as follows:

```
w := NIL;
while (there is input){
  K := next symbol from input;
  if (wK exists in the dictionary) {
    w := wK;
  } else {
    output (index(w));
    add wK to the dictionary;
    w := K;
  }
}
```

In the original proposal, the pointer size is chosen to be 12 bit, allowing for up to 4096 dictionary entries. Once this limit has been reached, the dictionary becomes static.

The GIF Controversy GIF image compression is probably the first thing that comes to mind for most people who have ever heard about the Lempel Ziv algorithm. Originally developed by CompuServe in the late 1980s, the GIF file format employs the LZW technique for compression. Apparently, CompuServe designed the GIF format without knowing that the LZW algorithm was patented by Unisys.

For years, the GIF format, as well as other software using the LZW algorithm, existed peacefully and became popular, without ever being subject to licensing fees by the patent holder. Then in 1994, Unisys announced a licensing agreement with CompuServe, and subsequently started demanding royalties from all commercial software developers selling software that incorporated LZW compression. Later the royalty demand was extended to non-commercial software, sparking an even greater outrage among the internet and software development community than the initial announcement. Demands to "Burn all GIFs" and efforts to produce a patent-free alternative to GIF, PNG, received considerable attention, but nevertheless GIF continues to be popular. The patent on the LZW algorithm will expire in June 2003.

Still, several other algorithms of the Lempel Ziv family remain protected by patents. Jean-loup Gailly, the author of the `gzip` compression program has done extensive research into compression patents. Some of his findings are contained in the FAQ of the newsgroup `comp.compression` [1].

4.4.2 LZC

LZC is the variant of the LZW algorithm that is used in the once popular UNIX `compress` program. It compresses according to the LZW principle but returns to variable-length pointers like the original LZ78 algorithm. The maximum index length can be set by the user of the compress program taking into account to the memory available (from 9 to 16 bits). It first starts with 9-bit indexes for the first 512 dictionary entries, then continues with 10-bit codes and so on until the user-specified limit has been reached. Finally, the algorithm becomes a static encoder, regularly checking the compression ratio. When it detects a decrease, it throws away the dictionary and starts building up a new one from scratch.

4.4.3 LZT

This is another variation on the LZW theme, the algorithm is almost like the LZC variant, the only difference being that it makes room for new phrases to be added to the dictionary by removing the least recently used entry (LRU replacement).

4.4.4 LZMS

LZMS creates new dictionary entries not by appending the first non-matching character, but by concatenating the last two phrases that have been encoded. This leads to the quick building of rather long entries, but in turn leaves out many of the prefixes of those long entries.

4.4.5 LZJ

The dictionary used by LZJ contains every unique string of the input up to a certain length, coded by a fixed-length index. Once the dictionary is full, all strings that have only been used once are removed. This is continued until the dictionary becomes static.

4.4.6 LZFG

LZFG uses the dictionary building technique from the original LZ78 algorithm, but stores the elements in a trie data structure. In addition, a sliding window like LZ77's is used to remove the oldest entries from the dictionary.

4.5 Comparison

The following chart shows a comparison of the compression rates for the different LZ78 variants. The same files (all values are taken from [2]) as for the LZ77 comparison have been used.

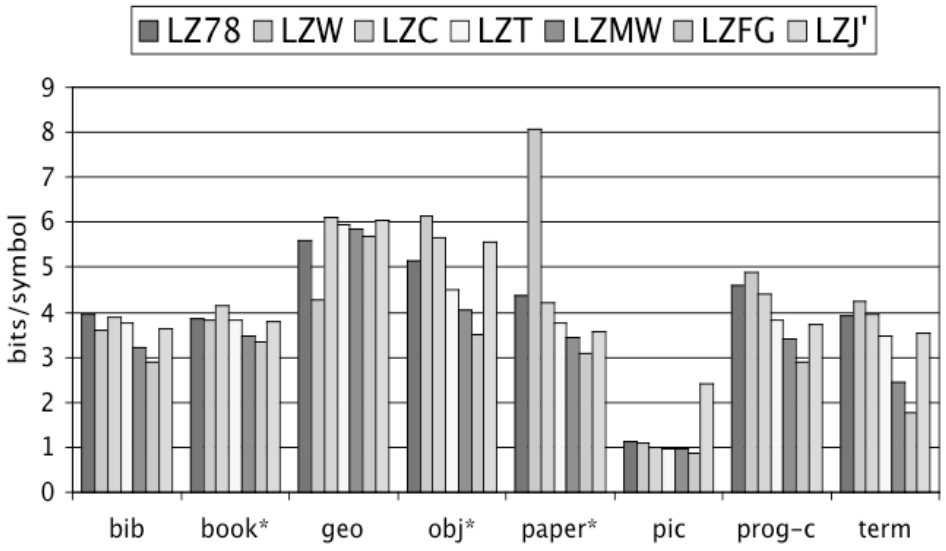


Figure 10: Compression rates for the LZ78 family

The first part of the "obj" files used in the test is very different from the rest of the file. Therefore, the LZW algorithm (static after 4096 dictionary entries) performs particularly bad – the entries created during the adaptive period are not representative of the entire file.

4.6 Comparison LZ77 and LZ78

The following chart shows a direct comparison between the best of the Lempel Ziv family (LZB for LZ77 variants and LZFG for LZ78 variants) and the algorithm which achieved the best results for statistical coding, PPMC, a PPM-based algorithm:

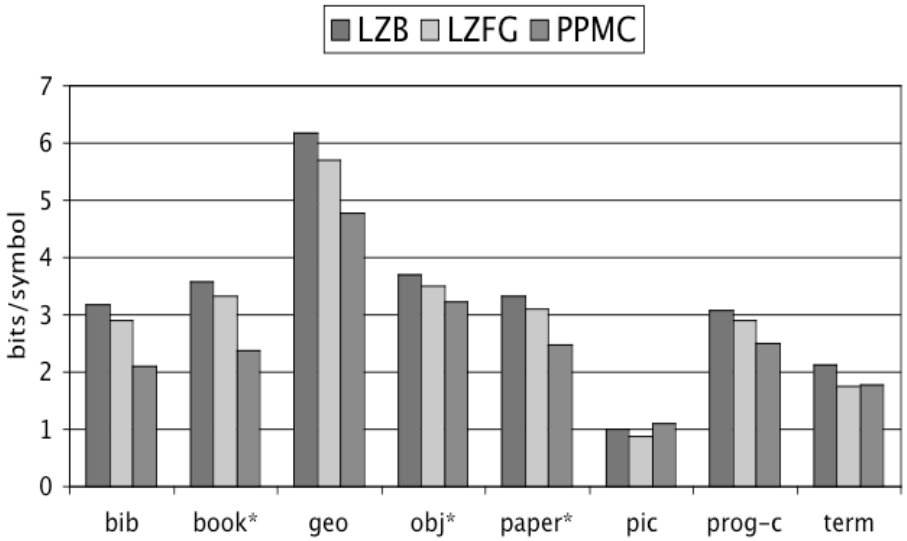


Figure 11: Comparsion LZ77/LZ78

The PPM-based algorithm outperforms Lempel Ziv algorithms in almost all test cases regarding compression-rate. But this algorithm requires significantly more ressources to achieve this compression than the Lempel Ziv algorithms.

References

[1] comp.compression FAQ. http://www.faqs.org/faqs/compression-faq/.

[2] BELL, T. C., CLEARY, J. G., AND WITTEN, I. H. *Text Compression*. Prentice Hall, Upper Sadle River, NJ, 1990.

[3] SAYOOD, K. *Introduction to Data Compression*. Academic Press, San Diego, CA, 1996, 2000.

[4] ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23* (1977), 337–343.

[5] ZIV, J., AND LEMPEL, A. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory 24* (1978), 530–536.