Create account Log in

Article Talk Read Edit View history Search

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Čeština
Deutsch
Eesti
Français
Italiano
עברית
Magyar
日本語
Polski
Português
Русский
Українська
中文
Edit links

LZ77 and LZ78

From Wikipedia, the free encyclopedia
(Redirected from Lempel–Ziv)

LZ77 and **LZ78** are the two lossless data compression algorithms published in papers by Abraham Lempel and Jacob Ziv in 1977[1] and 1978.[2] They are also known as **LZ1** and **LZ2** respectively.[3] These two algorithms form the basis for many variations including LZW, LZSS, LZMA and others. Besides their academic influence, these algorithms formed the basis of several ubiquitous compression schemes, including GIF and the DEFLATE algorithm used in PNG.

IEEE MILESTONE IN ELECTRICAL ENGINEERING
AND COMPUTING
LEMPEL-ZIV DATA COMPRESSION ALGORITHM, 1977
September 2004
INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS

A sign in an Israeli science museum describing the importance of the algorithm

They are both theoretically dictionary coders. LZ77 maintains a *sliding window* during compression. This was later shown to be equivalent to the *explicit dictionary* constructed by LZ78—however, they are only equivalent when the entire data is intended to be decompressed. LZ78 decompression allows random access to the input as long as the entire dictionary is available,[*dubious – discuss*] while LZ77 decompression must always start at the beginning of the input.

The algorithms were named an IEEE Milestone in 2004.[4]

Contents [hide]

Theoretical efficiency [edit]

In the second of the two papers that introduced these algorithms they are analyzed as encoders defined by finite-state machines. A measure analogous to information entropy is developed for individual sequences (as opposed to probabilistic ensembles). This measure gives a bound on the compression ratio that can be achieved. It is then shown that there exist finite lossless encoders for every sequence that achieve this bound as the length of the sequence grows to infinity. In this sense an algorithm based on this scheme produces asymptotically optimal encodings. This result can be proved more directly, as for example in notes by Peter Shor.[5]

LZ77 [edit]

LZ77 algorithms achieve compression by replacing repeated occurrences of data with references to a single copy of that data existing earlier in the uncompressed data stream. A match is encoded by a pair of numbers called a *length-distance pair*, which is equivalent to the statement "each of the next *length* characters is equal to the characters exactly *distance* characters behind it in the uncompressed stream". (The "distance" is sometimes called the "offset" instead.)

To spot matches, the encoder must keep track of some amount of the most recent data, such as the last 2 kB, 4 kB, or 32 kB. The structure in which this data is held is called a *sliding window*, which is why LZ77 is sometimes called *sliding window compression*. The encoder needs to keep this data to look for matches, and the decoder needs to keep this data to interpret the matches the encoder refers to. The larger the sliding window is, the longer back the encoder may search for creating references.

It is not only acceptable but frequently useful to allow length-distance pairs to specify a length that actually exceeds the distance. As a copy command, this is puzzling: "Go back *four* characters and copy *ten* characters from that position into the current position". How can ten characters be copied over when only four of them are actually in the buffer? Tackling one byte at a time, there is no problem serving this request, because as a byte is copied over, it may be fed again as input to the copy command. When the copy-from position makes it to the initial destination position, it is consequently fed data that was pasted from the *beginning* of the copy-from position. The operation is thus equivalent to the statement "copy the data you were given and repetitively paste it until it fits". As this type of pair repeats a single copy of data multiple times, it can be used to incorporate a flexible and easy form of run-length encoding.

Another way to see things is as follows: While encoding, for the search pointer to continue finding matched pairs past the end of the search window, all characters from the first match at offset D and forward to the end of the search window must have matched input, and these are the (previously seen) characters that comprise a single run unit of length L_R, which must equal D. Then as the search pointer proceeds past the search window and forward, as far as the run pattern repeats in the input, the search and input pointers will be in sync and match characters until the run pattern is interrupted. Then L characters have been matched in total, L>D, and the code is [D,L,c].

Upon decoding [D,L,c], again, D=L_R. When the first L_R characters are read to the output, this corresponds to a single run unit appended to the output buffer. At this point, the read pointer could be thought of as only needing to return int(L/L_R) + (1 if L mod L_R does not equal 0) times to the start of that single buffered run unit, read L_R characters (or maybe fewer on the last return), and repeat until a total of L characters are read. But mirroring the encoding process, since the pattern is repetitive, the read pointer need only trail in sync with the write pointer by a fixed distance equal to the run length L_R until L characters have been copied to output in total.

Considering the above, especially if the compression of data runs is expected to predominate, the window search should begin at the end of the window and proceed backwards, since run patterns, if they exist, will be found first and allow the search to terminate, absolutely if the current maximum matching sequence length is met, or judiciously, if a sufficient length is met, and finally for the simple possibility that the data is more recent and may correlate better with the next input.

Pseudocode [edit]

The pseudocode is a reproduction of the LZ77 compression algorithm sliding window.

```
begin
     fill view from input
     while (view is not empty) do
     begin
          find longest prefix p of view starting in coded part
          i := position of p in window
          j := length of p
          X := first char after p in view
          output(i,j,X)
          add j+1 chars
     end
end
```

Example [edit]

The calculation of the LZ77-based factorization of the string `aacaacabcabaaac` illustrated.

The table shows the calculation of the LZ77 factorization using a dictionary buffer of size 12 and a preview buffer of size 9. In the far right column is from top to bottom read the output of the algorithm (0, 0, "a") (1, 1, "c") (3, 4, "b") (3, 3, "a") (12, 3, "$"). The position is relative to the right edge of the dictionary buffer, this must be considered when decoding.

The buffers operate on the principle of a sliding window, i.e. to be compressed data stream is pushed right into the buffer. As noted in the algorithm, the shift is to the length of the match found in the dictionary, and a further position. This means that redundant triples be avoided as new characters are usually always taken individually in the dictionary. In the example, so the third triple (0, 0, "c") should be incorporated, what the compression ratio, however, deteriorated significantly. The matches are green and marked to be moved string in red. It is important to note that more and more a character is shifted, was found to be in accordance to new characters do not have to double encode.

Example of a LZ77 compression sliding window

Example of a LZ77 compression sliding window

Line	12	11	10	9	8	7	6	5	4	3	2	1	0	1	2	3	4	5	6	7	8	9		Output
1	(Empty)												a	a	c	a	a	c	a	b	c	a	⟹	(0,0,"**a**")
2	(Empty)											a	a	c	a	a	c	a	b	c	a	b	⟹	(1,1,"**c**")
3	(Empty)									a	a	c	a	a	c	a	b	c	a	b	a	a	⟹	(3,4,"**b**")
4	(Empty)				a	a	c	a	a	c	a	b	c	a	b	a	a	a	c	(Empty)			⟹	(3,3,"**a**")
5	a	a	c	a	a	c	a	b	c	a	b	a	a	a	c	(Empty)							⟹	(12,3,"**$**")
finished																								

The first popular characters is unknown, so that the first "a" is added to (0, 0, "a"). In the 2nd line "a" can already be read from the dictionary buffer (marked in green) so that "c" is accepted as the new character. In the 3rd line is a special case of the LZ77 algorithm can be seen as the matching string extends into the preview window, shown in the example by green text on a red background. Line 4 and 5 are equivalent to deal with the first two. Except that last a triple $ is next inserted character, since the text is fully compressed and there is no next character

Implementations [edit]

Even though all LZ77 algorithms work by definition on the same basic principle, they can vary widely in how they encode their compressed data to vary the numerical ranges of a length-distance pair, alter the number of bits consumed for a length-distance pair, and distinguish their length-distance pairs from *literals* (raw data encoded as itself, rather than as part of a length-distance pair). A few examples:

- The algorithm illustrated in Lempel and Ziv's original 1977 paper outputs all its data three values at a time: the length and distance of the longest match found in the buffer, and the literal which followed that match. If two successive characters in the input stream could only be encoded as literals, the length of the length-distance pair would be 0.
- LZSS improves on LZ77 by using a 1 bit flag to indicate whether the next chunk of data is a literal or a length-distance pair, and using literals if a length-distance pair would be longer.
- In the PalmDoc format, a length-distance pair is always encoded by a two-byte sequence. Of the 16 bits that make up these two bytes, 11 bits go to encoding the distance, 3 go to encoding the length, and the remaining two are used to make sure the decoder can identify the first byte as the beginning of such a two-byte sequence.
- In the implementation used for many games by Electronic Arts,[6] the size in bytes of a length-distance pair can be specified inside the first byte of the length-distance pair itself; depending on if the first byte begins with a 0, 10, 110, or 111 (when read in big-endian bit orientation), the length of the entire length-distance pair can be 1 to 4 bytes large.
- As of 2008, the most popular LZ77 based compression method is DEFLATE; it combines LZ77 with Huffman coding.[7] Literals, lengths, and a symbol to indicate the end of the current block of data are all placed together into one alphabet. Distances can be safely placed into a separate alphabet; since a distance only occurs just after a length, it cannot be mistaken for another kind of symbol or vice versa.

LZ78 [edit]

LZ78 algorithms achieve compression by replacing repeated occurrences of data with references to a dictionary that is built based on the input data stream. Each dictionary entry is of the form dictionary[...] = {index, character}, where *index* is the index to a previous dictionary entry, and character is appended to the string represented by dictionary[index]. For example, "abc" would be stored (in reverse order) as follows: dictionary[k] = {j, 'c'}, dictionary[j] = {i, 'b'}, dictionary[i] = {0, 'a'}, where an index of 0 specifies the first character of a string. The algorithm initializes *last matching index* = 0 and *next available index* = 1. For each character of the input stream, the dictionary is searched for a match: {last matching index, character}. If a match is found, then *last matching index* is set to the index of the matching entry, and nothing is output. If a match is not found, then a new dictionary entry is created: dictionary[next available index] = {last matching index, character}, and the algorithm outputs *last matching index*, followed by *character*, then resets *last matching index* = 0 and increments *next available index*. Once the dictionary is full, no more entries are added. When the end of the input stream is reached, the algorithm outputs *last matching index*. Note that strings are stored in the dictionary in reverse order, which an LZ78 decoder will have to deal with.

LZW is an LZ78-based algorithm that uses a dictionary pre-initialized with all possible characters (symbols), (or emulation of a pre-initialized dictionary). The main improvement of LZW is that when a match is not found, the current input stream character is assumed to be the first character of an existing string in the dictionary (since the dictionary is initialized with all possible characters), so only the *last matching index* is output (which may be the pre-initialized dictionary index corresponding to the previous (or the initial) input character). Refer to the LZW article for implementation details.

See also [edit]

- Lempel-Ziv-Stac (LZS)
- Statistical Lempel–Ziv

References [edit]

1. ^ Ziv, Jacob; Lempel, Abraham (May 1977). "A Universal Algorithm for Sequential Data Compression". *IEEE Transactions on Information Theory* **23** (3): 337–343. doi:10.1109/TIT.1977.1055714. CiteSeerX: 10.1.1.118.8921.
2. ^ Ziv, Jacob; Lempel, Abraham (September 1978). "Compression of Individual Sequences via Variable-Rate Coding". *IEEE Transactions on Information Theory* **24** (5): 530–536. doi:10.1109/TIT.1978.1055934. CiteSeerX: 10.1.1.14.2892.
3. ^ US Patent No. 5532693 Adaptive data compression system with systolic string matching logic
4. ^ "Milestones:Lempel-Ziv Data Compression Algorithm, 1977". *IEEE Global History Network*. Institute of Electrical and Electronics Engineers. 2014-07-22. Retrieved 2014-11-09.
5. ^ Peter Shor (2005-10-14). "Lempel-Ziv notes" (PDF). Retrieved 2014-11-09.
6. ^ "QFS Compression (RefPack)". *Niotso Wiki*. Retrieved 2014-11-09.
7. ^ Feldspar, Antaeus (23 August 1997). "An Explanation of the Deflate Algorithm". *comp.compression newsgroup*. zlib.net. Retrieved 2014-11-09.

External links [edit]

- "The LZ77 algorithm". *Data Compression Reference Center: RASIP working group*. Faculty of Electrical Engineering and Computing, University of Zagreb. 1997.
- "The LZ78 algorithm". *Data Compression Reference Center: RASIP working group*. Faculty of Electrical Engineering and Computing, University of Zagreb. 1997.
- "The LZW algorithm". *Data Compression Reference Center: RASIP working group*. Faculty of Electrical Engineering and Computing, University of Zagreb. 1997.

v · t · e		**Data compression methods** [hide]
Lossless	Entropy type	Unary · Arithmetic · Golomb · Huffman (Adaptive · Canonical · Modified) · Range · Shannon · Shannon–Fano · Shannon–Fano–Elias · Tunstall · Universal (Exp-Golomb · Fibonacci · Gamma · Levenshtein)
	Dictionary type	Byte pair encoding · DEFLATE · **Lempel–Ziv** (**LZ77 / LZ78** (**LZ1 / LZ2**) · LZJB · LZMA · LZO · LZRW · LZS · LZSS · LZW · LZWL · LZX · LZ4 · Statistical)
	Other types	BWT · CTW · Delta · DMC · MTF · PAQ · PPM · RLE
Audio	Concepts	Bit rate (average (ABR) · constant (CBR) · variable (VBR)) · Companding · Convolution · Dynamic range · Latency · Nyquist–Shannon theorem · Sampling · Sound quality · Speech coding · Sub-band coding
	Codec parts	A-law · μ-law · ACELP · ADPCM · CELP · DPCM · Fourier transform · LPC (LAR · LSP) · MDCT · Psychoacoustic model · WLPC
Image	Concepts	Chroma subsampling · Coding tree unit · Color space · Compression artifact · Image resolution · Macroblock · Pixel · PSNR · Quantization · Standard test image
	Methods	Chain code · DCT · EZW · Fractal · KLT · LP · RLE · SPIHT · Wavelet
Video	Concepts	Bit rate (average (ABR) · constant (CBR) · variable (VBR)) · Display resolution · Frame · Frame rate · Frame types · Interlace · Video characteristics · Video quality
	Codec parts	Lapped transform · DCT · Deblocking filter · Motion compensation
Theory	Entropy · Kolmogorov complexity · Lossy · Quantization · Rate–distortion · Redundancy · Timeline of information theory	
		Compression formats · Compression software (codecs)

Categories: Lossless compression algorithms