LZ-UTF8

LZ-UTF8 is a practical text compression algorithm and software designed with the following objectives and properties:

1. Compress UTF-8 and 7-bit ASCII strings **only**. No support for arbitrary binary content or other string encodings (such as ISO 8859-1, UCS-2, UTF-16 or UTF-32).
2. Fully compatible with UTF-8. **Any** valid UTF-8 byte stream is also a valid LZ-UTF8 byte stream (but not vice versa).
3. Individually compressed strings can be freely concatenated and intermixed with each other as well as with **plain UTF-8 strings**, and yeild a byte stream that can be decompressed to the equivalent concatenated source strings.
4. An input string may be incrementally compressed with some arbitrary partitioning and then be decompressed with any other arbitrary partitioning.
5. Decompressing any input block should **always** yield a valid UTF-8 string, trailing incomplete codepoint sequences should be saved and only sent to the output when a valid sequence is available. Decompressor should always output the longest sequence possible. Consequently: **no flushing** is needed when the end of a the stream is reached.
6. Should run efficiently in pure Javascript. Including on mobile. Decompression should be fast in particular.
7. Achieve reasonable compression ratio.
8. Simple algorithm. Relatively small amount of code (decompressor should be less than 100 lines of code).
9. One single permanent, standard scheme. No variations or metadata apart from the raw bytes. Try to adopt the best possible practices and find reasonable parameters that would be uniform across implementations (having uniform compressor output between different implementations would significantly simplify testing and quality assurance).

The Algorithm

The algorithm is the standard LZ77 algorithm with a window size of 32767 bytes. The minimum match length is 4 bytes, maximum is 31 bytes. The compressed stream format is relatively simple as it is completely byte aligned (necessary in order to allow some of the properties mentioned above).

The Compressed Byte Stream

There are two types of byte sequences in the stream. The first is a standard UTF-8 *codepoint sequence* - a leading byte followed by 0 to 3 continuation bytes. The second is a *sized pointer sequence* (or alternatively, a length-distance pair), made of 2 or 3 consecutive bytes, referencing the length and distance to a previous location in the stream.

The 5 least significant bits of the lead byte of a sized pointer sequence contain the length of sequence, which can have a value between 4 and 31. The 3 most significant bits for the lead byte are either 110 for a 2 byte pointer or 111 for a 3 byte pointer. This was intentionally chosen to be similar to the standard UTF-8 lead byte identifier bits*. The decoder will differentiate between a codepoint and a sized pointer sequence by the second byte's top bit - which is always 1 in a codepoint sequence but 0 in a sized pointer sequence.

The second and optionally third bytes of a pointer sequence represent the distance to the start of the pointed sequence. If the distance is smaller than 128, a single byte is used containing the literal 7 bit value. Otherwise two bytes are used and encoded as a big endian 15bit integer.

** The reason has to do with being easily able to detect invalid or truncated streams, and issues with decompression of incomplete parts.*

The following table demonstrates the unambiguity of the two encoding patterns (where “l” bits represent length and “d” bits represent distance) :

	Sized pointer sequence	UTF-8 Codepoint sequence
1 byte	**n/a**	**0**xxxxxxx
2 bytes	**110**lllll **0**ddddddd	**110**xxxxx **10**xxxxxx
3 bytes	**111**llllll **0**ddddddd dddddddd	**1110**xxxx **10**xxxxxx **10**xxxxxx
4 bytes	**n/a**	**11110**xxx **10**xxxxxx **10**xxxxxx **10**xxxxxx

The Compression Process (reference implementation)

Note: this describes the exact algorithm used by the reference implementation. In terms of decompressor compatibility, there’s no actual necessity that every implementation follows it, but having uniform results from other implementations would greatly simplify testing and overall quality assurance.

During compression every consecutive 4 byte input sequence is hashed (including overlapping sequences). The hash function used is a Rabin-Karp polynomial hash with a prime constant of 199 (`b0*199^3 + b1*199^2 + b2*199^1 + b3*199^0`). A rolling hash implementation may be used, although in pratice it has proven to be slower than the simpler recalculation of the hash.

The hash is then looked up in a hash table (having a capacity of exactly 65537 buckets, with a hash index of (hash mod 65537)) to find a match to previous occurrences of the 4 byte sequence it represents. If the bucket representing the hash of the sequence is empty, no match has been found: the literal byte is sent to the output and read position is incremented to the next byte. Otherwise a match may have been found: all sequences (within the 32k window) referenced in the bucket are then tested. For every possible match tested, all bytes (including additional bytes beyond the first 4) are linearly compared up to the longest match possible (or the maximum allowed - 31).

Note: the bucket may contain references to non-matching sequences - that may happen because of hash collisions. This is not a problem as they will not pass the linear comparison.

The longest match found* in the bucket is the one eventually used. A length-distance pair is then encoded to a sized pointer sequence of 2 or 3 bytes (described above) and sent to the output. Read position is then incremented to the next byte(s), sequences within the matched sequence are hashed and added to the hash table, but not compared to previous ones. Matching resumes at the first byte following the matched range.

A reference to the current sequence is then added to the bucket. Every bucket in the hash table may contain up to 64 elements. When filled up to its maximum capacity, the oldest 32 elements are discarded.

** Since distances smaller than 128 are encoded to 2 bytes and larger by 3, the compressor optimizes for the highest length/byte count ratio. It will reject matches of distance >=128 that are not more than 1.5x longer than a previous match of distance <128. This may also slightly improve compression speed.*

The Decompression Process

Decompression is very simple. The compressed stream is iterated one byte at a time. If the current byte's top bits are 111 or 110 and the following one has 0 as top bit, the current byte and the one(s) following, are interpreted and decoded as a sized pointer sequence (described above). Otherwise, the raw input byte value is sent to the output. Decoding a sized pointer is as simple as finding its start position (= current read position - distance) and copying the pointed bytes to the output, numbered by length.

Note: when decompressing an arbitrary, partial block from the input stream, special care is needed to always output valid UTF8 strings. If any of the last 1-3 output bytes are part of a truncated codepoint or pointer sequence, they are saved and only decoded with the next block.

Efficiency and Performance

The Javascript implementation, benchmarked on a single core of an Intel Pentium G3320 (a low end desktop) on a modern browser, for 1MB files, has a typical compression speed of 3-14MB/s. Decompression speed ranges between 20-80MB/s. The C++ implementation is typically around 30-40MB/s for compression and 300-500MB/s for decompression. These figures may further improve with subsequent optimizations of the code.

Compression ratio is usually lower than *lzma* and *lz-string* on relatively long inputs (more than 64KB) and similar on shorter ones. On very long (or some pathologically repetitive) inputs, it may be substantially lower, mainly because of the 32k window and 31 byte sequence limits.

Compression memory requirement is about <input size> * 2 plus a hash table memory size bounded by an approximate worst case of 65537 * 64 * 4 (<BucketCount> * <MaxBucketCapcity> * <SequenceReferenceByteSize>) = ~16MB. The theoretical upper bounding number of different 4 byte permutations for a 32kb window is 4 * 32kb = 128kb (actual number is smaller because of overlap), which is about twice the hash table bucket count, yielding a worst case load factor of about 2x (assuming a uniform hash distribution), which is acceptable. A large/smaller maximum bucket capacity may be used, increasing/decreasing compression ratio slightly (and decreasing/increasing compression speed respectively), though experimentally it hasn't proven to be significant enough, given the small window size.

Some example comparison results in various browsers:

(note this includes binary to string conversion time, which may be a significant fraction of the decompression time)

English Bible Excerpt (978453 bytes)

Program Name	Compressed size	Comp. (FF 32)	Decomp. (FF 32)	Comp. (Ch 37)	Decomp. (Ch 37)	Comp. (IE11)	Decomp. (IE11)
lzma	244983	2604	307	2825	615	3527	1396
lz-string	327206	577	77	455	64	976	71
lz-utf8	367359	109	17	103	17	257	50

Hindi Bible Excerpt (999716 bytes)

Program Name	Compressed size	Comp. (FF 31)	Decomp. (FF 31)	Comp. (Ch 37)	Decomp. (Ch 37)	Comp. (IE11)	Decomp. (IE11)
lzma	140104	3738	228	3230	552	3826	1190
lz-string	157328	213	37	309	45	300	34
lz-utf8	226721	95	16	90	15	280	40

Japanese Alice in Wonderland (246025 bytes)

Program Name	Compressed size	Comp. (FF 31)	Decomp. (FF 31)	Comp. (Ch 37)	Decomp. (Ch 37)	Comp. (IE11)	Decomp. (IE11)
lzma	59711	1405	80	1084	141	1088	272
lz-string	59472	58	14	113	11	63	13
lz-utf8	86302	59	4	52	3	98	8

JQueryUI.js (451466 bytes)

Program Name	Compressed size	Comp. (FF 31)	Decomp. (FF 31)	Comp. (Ch 37)	Decomp. (Ch 37)	Comp. (IE11)	Decomp. (IE11)
lzma	95942	4198	129	1284	262	1518	585
lz-string	154464	212	36	214	19	334	36
lz-utf8	137170	74	8	62	8	110	24