Analysis of LZ3

How many bits does LZ3 use to encode a sequence? For each newly parsed phrase, the encoder needs to specify the index of the best matching (prefix) phrase, and the new (unmatched) symbol. For the i^{th} phrase, the index of the prefix is a number from 0 to $i-1$ and so a code with $\lceil \log i \rceil$ bits is sufficient.[1] For the unmatched symbol, a fixed length code of $\lceil \log N \rceil$ bits is sufficient.

Let $\phi(n)$ be the number of phrases of the entire sequence. Using the above coding scheme, we would use

$$totbits \;=\; \sum_{i=1}^{\phi(n)} (\; \lceil \log i \rceil + \lceil \log N \rceil \;)$$

We assume $N = 2$ to keep the analysis simple, so

$$totbits \;=\; \sum_{i=1}^{\phi(n)} (\; \lceil \log i \rceil + 1)$$

Clearly, we get better compression when the number of phrases $\phi(n)$ is small.

Using Jensen's inequality, one can manipulate the above equation to get an upper bound:

$$totbits \;\leq\; \phi(n)\; (\; \log(\phi(n)+1) + 1 \;) \tag{1}$$

bits, since $i \leq \phi(n)$ by definition.

Let's briefly examine the best and worst case performance of LZ3, in order to give you a flavor of the analysis people have carried out. The basic approach is to estimate how $\phi(n)$ depends on n.

Best case:

For the case we are considering, the LZ3 algorithm compresses best when the number of phrases is minimized. This happens when the best match for each new phrase is to the longest previous phrase, for example.

`a`, `ab`, `aba`, `abaa`, `abaab`, `abaaba`, `abaabab`, `abaababa`, *etc.*

To calculate $\phi(n)$ for the best case, we note that phrase i has i symbols. Thus,

$$\begin{aligned} n &= \sum_{i=1}^{\phi(n)} i \\ &= \frac{\phi(n)(\phi(n)+1)}{2} \end{aligned}$$

Thus, in the best case:

$$\phi(n) \approx \sqrt{2n}\;.$$

[1]Two qualifiers here. First, we ignore the special case of the last phrase in the file where we encode i for the phrase number. Second, we might be more clever and avoid using $\lceil \log i \rceil$ bits, since some of the internal nodes of the encoder's tree might already have filled all their children, in which case both the encoder and decoder know that these nodes cannot be the best match. (Think why.)

How many bits do we use in total in the best case? Noting that $\log\sqrt{2n} = \frac{1}{2}(\log(n)+1)$, we substitute into (1) above and get

$$totbits \quad \approx \quad \sqrt{\frac{n}{2}}\ \log(n)\ .$$

Note *totbits* grows much slower than n.

How many best cases are there? If there are $\phi(n)$ phrases and $N = 2$, then there are $2^{\phi(n)}$ best cases since each phrase can extend the previous longest phrase in two ways ($N = 2$). Note that $2^{\phi(n)} \ll 2^n$ which is the number of possible sequences of length n. Thus, the percentage of sequences that are best cases is quite small.

Worst case:

The worst case occurs when $\phi(n)$ is maximized. This happens when each phrase is as short as possible. For alphabet $\{\mathtt{a}, \mathtt{b}\}$, the worst case for LZ3 would be a sequence like

$$\mathtt{a}, \mathtt{b}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}, \mathtt{aa}, \mathtt{aaa}, \mathtt{aab}, \mathtt{aba}, \mathtt{abb}, \ldots,$$

that is, all the phrases of length l, followed by all the phrases of length $l+1$, etc.

When the encoder's tree is filled up to depth d,

$$\phi(n) = \sum_{l=1}^{d} 2^l = 2^{d+1} - 2.$$

The reason is that a complete binary tree of depth d has $2^{d+1} - 1$ nodes, and we are not counting the root node since it corresponds to an empty phrase.

$$\begin{aligned} totbits &\leq \phi(n)(\log(\phi(n)+1) \\ &= (2^{d+1}-2)(\log(2^{d+1}-2)+1) \\ &\leq 2^{d+1}\,(d+2) \end{aligned}$$

How does this compare to n ? In the worst case, level l of the tree has 2^l nodes (each node representing one phrase), and thus:

$$n = \sum_{l=1}^{d} l\ 2^l$$

In Appendix 1, I show that

$$\sum_{l=1}^{d} l\ 2^l \ = \ (d-1)2^{d+1} + 2 \qquad (2)$$

So, in this worst case we are considering, the average number of code bits *per symbol in the sequence* is bounded above by:

$$\frac{totbits}{n} \leq \frac{d+2}{d-1} = 1 + \frac{3}{d-1}$$

This is greater than 1, which is no surprise since we cannot expect to compress every sequence.

The nice property is that, in the limit as $d \to \infty$, this ratio goes to 1. So even in the worst case, the LZ3 algorithm doesn't systematically blow up. *In Appendix 2 below, I show how how to write this bound in terms of n rather than d.*

Appendix 1 (you are NOT responsible for this)

In the following proof of (2), I use k rather than depth d to avoid the notation confusion with the "d" in the derivative.

You should be familiar with the identity:

$$\sum_{l=0}^{k} x^l \;=\; \frac{x^{k+1}-1}{x-1}$$

which holds as long as $x \neq 1$. Taking the derivative of both sides gives,

$$\sum_{l=0}^{k} \; l \; x^{l-1} \;=\; \frac{d}{dx}(\frac{x^{k+1}-1}{x-1})$$

Multiplying by both sides by x gives:

$$\sum_{l=0}^{k} \; l \; x^{l} \;=\; x \; \frac{d}{dx}(\frac{x^{k+1}-1}{x-1})$$

Crunch through the derivative on the right side, and substitute $x = 2$ gives the identity to be proven, namely

$$\sum_{l=1}^{k} \; l \; 2^{l} \;=\; (k-1)2^{k+1}+2$$

Note that the $l = 0$ term of the sum is 0.

Appendix 2 (you are NOT responsible for this)

How fast does the compression ratio converge to 1 ? The inequality above is written in terms of d. Let's try to write it instead in terms of n

From Eq. (2), we have

$$\frac{n-2}{4} = (d-1)2^{d-1}$$

but this is still a bit too messy. Instead let's just define

$$n' \;\equiv\; \frac{n-2}{4} \qquad \text{and} \qquad d' = d-1$$

and so

$$n' = d' \; 2^{d'}$$

and study how d' depends on n'.

Taking the log of both sides, we rewrite:

$$\log n' = d' + \log d'$$

Since $n' > \; 2^{d'}$, we have $\log n' > \; d'$ and so

$$-\log\log n' < -\log \; d'.$$

Thus,

$$d' = \log n' - \log d' > \log n' - \log\log n'$$

and so

$$\frac{1}{d'} < \frac{1}{\log n' - \log\log n'}$$

and so

$$\frac{totbits}{n} \leq 1 + \frac{3}{\log n' - \log\log n'} \ .$$

They key observation is that, as $n \to \infty$, this ratio goes to 1. Thus, LZ3 does not blow up in the worst case.