11

Dictionary-based compressors

The methods based on a *dictionary* take a totally different approach to compression than the statistical ones: here we are not interested in deriving the characteristics (i.e. probabilities) of the source which generated the input sequence S. Rather, we assume to have a *dictionary of strings*, and we look for those strings in S, replacing them with a *token* which identifies them in the dictionary. The choice of the dictionary is of course crucial in determining how well the file is compressed. An English dictionary will have a hard time to compress an Italian text, for instance; and it would be totally unappropriate to compress an executable file. Thus, while a *static* dictionary can be used to compress very well certain specific and known in advance kinds of files, it cannot be used for a good *general-purpose compressor*. Moreover, we don't want to transmit the full dictionary along with each compressed file – and it's often unreasonable to assume the receiver already has a copy of our dictionary.

So, starting from 1977, Ziv and Lempel introduced a family of compressors which addressed successfully these problems by designing two algorithms, named LZ77 and LZ78 from the initials of the inventors and the years of the proposal, which use the input sequence they are compressing as the dictionary, and substitute each occurrence of an already seen string with either the offset of its previous position or an ID assigned incrementally to new dictionary phrases. The dictionary is *dynamically built* in the sense that it starts empty and then it grows as the input sequence is processed; at the beginning low compression is achieved, but after some kbs good compression is obtained. For typical textual files, those methods achieve about 33% compression ratio. The Lempel-Ziv's compressors are very popular because of their `gzip` instantiation, and constitute the base of more sophisticated compressors in use today, such as `7zip` and `LZMA` and `LZ4`. In the following paragraphs, we will show them in detail, along with some interesting variants.

11.1 LZ77

Ziv and Lempel, in their seminal paper of 1977 [11], described their contribution as follows *"[...] universal coding scheme which can be applied to any discrete source and whose performance is comparable to certain optimal fixed code book scheme designed for completely specified sources [...]"*. They key expression is "comparable to [...] fixed code book scheme designed for completely specified sources", because the authors compare to previously designed statistical compressors, such as Huffman and Arithmetic, for which a statistical characterization of the source was necessary.

Conversely, dictionary-based compressors waive this characterization which is derived *implicitly* by observing *substring repetitiveness* via a fully syntactic approach.

We will not dig into the observations which provide a mathematical ground to these comments [11, 3], rather we will concentrate only on the algorithmic issues. LZ77's compressor is based on a *sliding window* $W[1, w]$ which contains a portion of the input sequence that has been processed so far, typically consisting of the last w characters, and a *look-ahead buffer* B which contains the suffix of the text still to be processed. In the following picture the window $W = aabbbabab$ is of size 9, and the rest of the input sequence is $B = baababaabbaa\$$.

$$\longleftarrow \cdots \boxed{\text{aabbbabab}}\ baababaabbaa\$ \longrightarrow$$

The algorithm works inductively by assuming that everything occurring before B has been processed and compressed by LZ77; where W is initially set to the empty string. The compressor operates in two main stages: *parsing* and *encoding*. Parsing consists of transforming the input sequence S into a sequence of triples of integers (called *phrases*). Encoding turns these triples into a (compressed) bit stream by applying either a statistical compressor (i.e. Huffman or Arithmetic) to each triplet-component individually, or an integer encoding scheme.

So the interesting algorithmic stage is the parsing stage, which works as follows. LZ77 searches for the longest prefix α of B which occurs as a substring of WB. We write the concatenation WB rather than the single string B because the previous occurrence we are searching for may start in W and extend up to within B. Say α occurs at distance d from the current position (namely the beginning of B), and it is followed by character c in B, then the triple generated by LZ77 is $\langle d, |\alpha|, c\rangle$ where $|\alpha|$ is the length of the *copied* string. If a match is not found the output triple becomes $\langle 0, 0, B[1]\rangle$. We notice that any occurrence of α in W must be followed by a character different of c, otherwise α would be *not* the longest prefix of B which repeats in W.

After that this triple is emitted, LZ77 advances in B by $|\alpha| + 1$ positions, and slides W correspondingly. We talk about LZ77 as a dictionary-based compressor because "the dictionary" is not explicitly stored, rather it is implicitly formed by all substrings of S which start in W and extend rightward, possibly ending in B. Each of those substrings is represented by the triple indicated above. The dictionary is *dynamic* because at every shift it has to be updated by removing the substrings starting in $W[1, |\alpha| + 1]$, and adding the substrings starting in $B[1, |\alpha| + 1]$.

The role of the sliding window is easy to explain, it delimits the size of the dictionary which is quadratic in W's length, so it impacts significantly onto the time cost for the search of α. As a running example, let us consider the following sequence of LZ77-parsing steps:

$$\begin{array}{rcl}
\boxed{|\text{aabbabab}} & \Longrightarrow & \langle 0, 0, a\rangle \\
\boxed{\text{a}|\text{abbabab}} & \Longrightarrow & \langle 1, 1, b\rangle \\
\boxed{\text{aab}|\text{babab}} & \Longrightarrow & \langle 1, 1, a\rangle \\
\boxed{\text{aabba}|\text{bab}} & \Longrightarrow & \langle 2, 3, EOF\rangle \\
\cdots\cdots \quad \cdots\cdots & &
\end{array}$$

It is interesting to note that the last phrase $\langle 2, 3, EOF\rangle$ presents a copy-length which is larger than the copy-distance; this actually indicates the special situation mentioned above in which α starts in W and ends in B. Even if this *overlapping* occurs, the copy-step that must be executed by LZ77 in decompression is not affected, provided that it is executed sequentially according to the following piece of code:

```
for i = 0 to L-1 do { S[s+i] = S[s-d+i]; }
s = s+L;
```

where the triple to be decoded in $\langle d, L, c\rangle$ and $S[1, s-1]$ is the prefix of the input sequence which has been already decompressed. Since $d \leq |W|$ and the window size is up to few Megabytes, the copy operation does not elicit any cache miss, thus making the decompression process very fast indeed. The longer is the window W, the longer are possibly the phrases, the fewer is their number and thus possibly the shorter is the compressed output; but of course, in terms of compression time, the longer is the time to search for the longest copied α. Vice versa, the shorter is W, the worse is the compression ratio but the faster is the compression time. This trade-off is evident and its magnitude depends on the input sequence.

To slightly improve compression we make the following observation which is due to Storer and Szymanski [8] and dates back to 1982. In the parsing process two situations may occur: a longest match has been found, or it has not. In the former case it is not reasonable to add the character following α (third component in the triple), given that we anyway advance in the input sequence. In the latter case it is not reasonable to emit two 0s (first two components in the triple) and thus waste one integer encoding. The simplest solution to these two inefficiencies is to always output a pair, rather than a triple, with the form: $\langle d, |\alpha|\rangle$ or $\langle 0, B[1]\rangle$. This variant of LZ77 is named LZss, and it is often confused with LZ77, so we will use it from this point on.

By referring to the previous running example, LZss would obtain the parsing:

$$
\begin{array}{rcl}
|\text{aabbabab} & \Longrightarrow & \langle 0, a\rangle \\
\text{a}|\text{abbabab} & \Longrightarrow & \langle 1, 1\rangle \\
\text{aa}|\text{bbabab} & \Longrightarrow & \langle 0, b\rangle \\
\text{aab}|\text{babab} & \Longrightarrow & \langle 1, 1\rangle \\
\text{aabb}|\text{abab} & \Longrightarrow & \langle 3, 2\rangle \\
\text{aabbab}|\text{ab} & \Longrightarrow & \langle 2, 2\rangle
\end{array}
$$

Gzip: a smart and fast implementation of LZ77. The key programming problem when implementing LZ77 is the *fast search* for the longest prefix α of B which repeats in W. A brute-force algorithm that checks the occurrence of every prefix of B in W, via a linear backward scan, would be very time-consuming and thus unacceptable for compressing Gbs files.

Fortunately, this process can be accelerated by using a suitable data structure. `Gzip`, the most popular implementation of LZ77, uses a hash table to determine α and find its previous occurrence in W. The idea is to store in the hash table all 3-grams occurring in W, namely all triplets of contiguous characters, by using as `key` the 3-gram and as its `satellite` data the position in B where that 3-gram occurs. Since a 3-gram may repeat multiple times in W, the hash table saves for a given 3-gram all of its multiple occurrences, sorted by increasing position in S. This way, when W shifts to the right because of the emission of the pair $\langle d, \ell\rangle$, the hash table can be updated by deleting the ℓ 3-grams starting at $W[1, \ell]$, and inserting the ℓ 3-grams starting at $B[1, \ell]$.

The search for α is implemented as follows:

- first, the 3-gram $B[1, 3]$ is searched in the hash table. If it does not occur, then `Gzip` emits the phrase $\langle 0, B[1]\rangle$, and the parsing advances of one single character. Otherwise, it determines the list $\mathcal{L}$ of occurrences of $B[1, 3]$ in W.
- second, for each position i in $\mathcal{L}$ (which is expressed as absolute position in S), the algorithm compares character-by-character $S[i, n]$ against B in order to compute their longest common prefix. At the end, the position $i^* \in \mathcal{L}$ sharing this longest common prefix is determined, as well as it is found α.

- finally, let p be the current position of B in S, the algorithm emits the pair $\langle p - i^*, |\alpha| \rangle$, and advances the parsing of $|\alpha|$ positions.

`Gzip` implements the encoding of the phrases by using Huffman over two alphabets: the one formed by the lengths of the copies plus the literals, and the one formed by the distances of the copies. This trick is sufficiently smart to save one extra bit to distinguish between the two types of pairs. In fact, $\langle 0, c \rangle$ is represented as the Huffman encoding of c, and $\langle d, \ell \rangle$ is represented reversed by anticipating the Huffman encoding of ℓ. Given that literals and copy-lengths are encoded within the same alphabet, the decoder fetches the next codeword and decompresses it, so being able to distinguishing whether the next item is a character c or a length ℓ. According to the result, it can either restart the decoding of the next pair (c has been decoded), or it can decode d (ℓ has been decoded) by using the other Huffman code.

`Gzip` deploys an additional programming trick that further speed ups the compression process. It consists of sorting the list of occurrences of the 3-grams from recent to oldest matches, and possibly stop the search for α when a sufficient number of candidates has been checked. This trades the length of the longest match against the speed of the search. As far as the size of the window W is concerned, `Gzip` allows to specify $-1, \ldots, -9$ which actually means that the size may vary from 100Kb to 900Kb, with a consequent improvement of the compression ratio, at the cost of slowing down the compression speed. Not surprisingly, the longer is W, the faster is the decompression because the smaller is the number of encoded phrases, and thus the smaller is the number of cache misses induced by the Huffman decoding process.

For other implementations of LZ77, the reader can look at Chapter 8 where we discussed the use of the Suffix Tree and unbounded window; as well as we refer to [4] for details about implementations which take into account the size of the compressed output (in bits) which clearly depends on the number of phrases but also from the values of their integer components, in a way that cannot be underestimated. Briefly, it is not necessarily the case that a longer α induces a shorter compressed output, because it might need to copy α from a far distance d, thus taking many bits for its encoding; rather, it might be better to divide α into two substrings which can be copied closer enough that the total number of bits required for their encoding is less than the ones needed for d.

11.2 LZ78

The sliding window used by LZ77 from the one hand speeds up the search for the longest phrase to encode, but from the other hand limits the search space, and thus the ultimate compression ratio. In order to avoid this problem and still keep a fast compression stage, Ziv and Lempel devised in 1978 another algorithm, which has been consequently called LZ78 [12]. The key idea is to build *incrementally* an *explicit dictionary* that contains only a subset of the substrings of the input sequence S, selected according to a simple rule that is detailed below. Concurrently, S is decomposed into phrases which are taken from the current dictionary.

Phrase detection and dictionary update are deeply intermingled. Let S' be the sequence to be parsed yet, and let $\mathcal{D}$ be the current dictionary in which every phrase f is identified via the integer $\mathtt{id}(f)$. The parsing of S' consists of determining its longest prefix f' which is also a phrase of $\mathcal{D}$, and substituting it with the pair $\langle \mathtt{id}(f'), c \rangle$ where c is the character following f' in S'. Next, $\mathcal{D}$ is updated by adding the new phrase $f'c$, which is just one character longer than the phrase $f' \in \mathcal{D}$. Therefore the dictionary is *prefix-complete* because it will contain all the prefixes of every phrase in $\mathcal{D}$, moreover its size grows with the length of the input sequence.

It goes without saying that, as it occurred for LZ77, the stream of pairs generated by the LZ78-parsing will be encoded via a statistical compressor (such as Huffman or Arithmetic) or via any variable-length integer encoder. This will produce the compressed bit stream, eventual output of LZ78.

Input	Output	Dictionary
-	-	0: empty
a	<0, a>	1: a
ab	<1, b>	2: ab
b	<0, b>	3: b
aba	<2, a>	4: aba
bb	<3, b>	5: bb
ba	<3, a>	6: ba
abab	<4, b>	7: abab
aa	<1, a>	8: aa

TABLE 11.1 LZ78-parsing of the string $S = aabbababbbaababaa$.

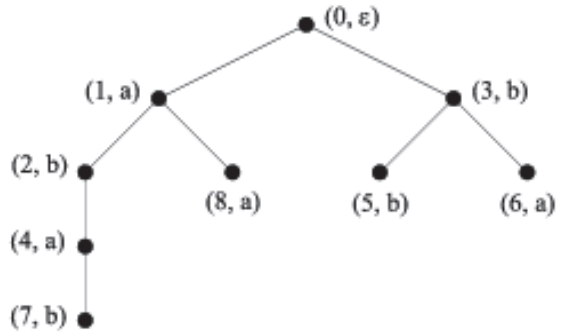

FIGURE 11.1: The *trie* for the dictionary in table 11.1

As an illustrative example, let us consider the sequence $S = abcdefg\ldots$ and the dictionary $\mathcal{D}$ containing the phrase *abcd* with `id` = 43, but not containing the phrase *abcde*. Given this scenario, LZ78 outputs the pair $\langle 43, e\rangle$ and adds the new phrase *abcde* to the current dictionary. The parsing will then continue over the suffix $S = fg\ldots$. Table 11.1 reports a full running example.

The decompressor works in a very similar way: it reads a pair $\langle$`id`$, x\rangle$, it determines the phrase f corresponding to the integer `id` in the current dictionary $\mathcal{D}$, emits the substring fc and updates the current dictionary by adding that substring as a new phrase.

The LZ78 algorithm needs an efficient data structure to manage the dictionary, which can be easily found in the *trie* (see Chapter 7), given that it supports fast insertion and prefix-search of strings. The prefix-complete property satisfied by $\mathcal{D}$ ensures that the trie is *uncompacted*, namely every edge is labeled with a single character. The encoding algorithm fits nicely on this structure (see Figure 11.1). Searching for the longest prefix of S' which is a dictionary phrase, can be implemented by traversing the trie according to S''s characters until a trie leaf is reached. That is the detected phrase f'. In addition, the new phrase $f'c$ is inserted in the trie by just appending a new node to the leaf for f' and labeling it with the single character c. That new node will be actually a leaf of the trie.

The final question is how do we manage large files and, thus, large dictionaries which host longer and longer phrases. There are a few possibilities to cope with this problem:

1. Freeze the dictionary, disallowing the entry of new strings. This is the simplest option.
2. Discard the dictionary, starting with a new empty one. This can also be an advantage if the file can be seen as structured in blocks, each one with a different set of recurring strings.

Input	Output	Dictionary
a	-	0-255:'\0'-'\255'
a	97 (a)	256: aa
b	97 (a)	257: ab
b	98 (b)	258: bb
ab	98 (b)	259: ba
aba	257 (ab)	260: aba
eof	97 (aba)	261: aba EOF

TABLE 11.2 LZW-encoding of $S = aabbababa$; 97 and 98 are the ASCII codes for a and b.

3. Before inserting a new string, delete the least recently used one. This solution recalls a sort of LRU model in the dictionary access and management.

11.3 LZW

LZW is a very popular variant of LZ78, developed by Welch in 1984 [10]. Its main objective is to avoid the need for the second component of the pair $\langle \texttt{id}(f'), c\rangle$, and thus for the byte representing a character. To accomplish this goal, before the start of the algorithm, all possible one-character strings are written into the dictionary. This means that the phrase-ids from 0 to 255 have been allocated to these characters. Next, the parsing of S starts searching for the longest prefix f' which matches a phrase in $\mathcal{D}$. Since the next prefix $f'c$ of S' does not occur in $\mathcal{D}$, then $f'c$ is added to the dictionary, taking the next available `id`, and the next phrase to detect starts from c *rather than* from the following character, as instead done in LZ78 and LZ77. So parsing and dictionary updating are misaligned.

Decoding is a bit tricky because of this misalignment. Assume that decoding has to manage two ids i' and i'', and call their corresponding dictionary phrases f' and f''. The decoder, in order to re-align the dictionary, has to create the phrase n' from the reading of i' and i'', setting $n' = f'f''[1]$, where $f''[1]$ is the first character of the phrase f''. This seems easy but, indeed, it is not!

If f' and f'' are already available in $\mathcal{D}$, the construction of n' is a trivial task, just set $n' = f'f''[1]$. But if f'' is not available the situation gets complicate! Let us look how this can happen. Take the compression stage when the compressor emitted i' for f' and inserted $n' = f'f''[1]$ in $\mathcal{D}$. Clearly the compressor knows f'' and so can do this insertion.

But if the next phrase f'' starts with n', then the decompressor is in trouble. In fact the decompressor sees i', decodes it and obtains f', but then it needs to construct n' and insert it in $\mathcal{D}$. This seems not possible because n' needs $f''[1]$ and we have that f'' is prefixed by n' which is exactly the string we are willing to reconstruct. Thus we falled in a sort of circular definition.

To circumvent this circularity it is enough to observe that f'' consists of at least one character (recall that we initialized $\mathcal{D}$ with all single characters) and it is $f''[1] = n'[1] = (f'f''[1])[1] = f'[1]$ which is indeed available! So the reconstruction of n' is possible even in this special case, by setting $n' = f'f'[1]$. Hence the decompressor can correctly construct n', insert it in $\mathcal{D}$, and thus re-align the dictionaries available at LZW's compressor and decompressor after the reading of i'.

Table 11.3 shows the actions taken by the LZW-decoder over the stream of `id`s obtained from Table 11.2. The dictionary starts with all the possible characters in the ASCII code, each one with its value, so the new phrases take `id`s from 256. Notice that the special case occurs when 260 is read from the compressed sequence but the phrase 260 is not in the dictionary because it has yet to be constructed. Nevertheless, by using the observations above, we can conclude that $n' = f'f'[1] = aba$.

Input	Dictionary	Output
-	0-255:'\0'-'\255'	
97	256: a?	a
97	256: aa	a
	257: a?	
98	257: ab	b
	258: b?	
98	258: bb	b
	259: b?	
257	259: ba	ab
	260: ab?	
260	261: aba	aba

TABLE 11.3 LZW-decoding of the id-stream: 97,97,98,98,257,257,258,256,259,259,97.

As the LZ77 algorithm, LZW has many common-use implementations among which we point out the popular GIF image format [9]. It assumes that the original (uncompressed) image is rectangular and uses 8 bits per pixel, so the alphabet has size 256 and the input sequence S comes as a normal stream of bytes, obtained by reading line-by-line the pixels of the image[1]. Since 8 bits are very few to represent all possible colors of an image, each value actually is an index in a *palette*, whose entries are 24-bits descriptions of the actual color (the typical RGB format). This restricts the maximum number of different colors present in an image to 256.

Some researchers [5] explored the possibility to introduce a lossy variant of GIF compression without changing the way the output is represented: this would give the possibility to have a shorter format, using however standard decoders. The basic idea is in fact quite simple: instead of looking for the longest *exact* match in the dictionary while parsing, we perform some kind of *approximate* matching. In this way we can find longer matches, thus reducing the output size, but at the cost of representing a slightly different image. Approximate matching of two strings of colors is done with a measure of difference based on their actual RGB values, which must be guaranteed to not exceed a threshold value.

11.4 On the optimality of compressors$^{\infty}$

The literature shows many results regarding the optimality of LZ-inspired algorithms. Ziv and Lempel themselves demonstrated that LZ77 is optimal for a certain family of sources (see [11]), and LZ78 asymptotically reaches the best compression ratio among finite-state compressors (see [12]). Optimality here means that, assuming the string to compress is infinite and is produced by a *stationary ergodic source with a finite alphabet*, then the compression ratio asymptotically tends to the entropy of the underlying source. More recent results made it possible to have a quantitative estimate of algorithms' *redundancy*, which is a measure of the distance between the source's entropy and the compression ratio, and can thereby be seen as a measure of "how fast" the algorithm reaches the source's entropy.

All these measures are very interesting but unrealistic because it is actually quite unusual, if not impossible, to know the entropy of the source which generated the string we're going to compress. In order to circumvent this problem a different empirical approach has been taken by introducing

[1] Actually the GIF format can also present the lines in an *interleaved* format, the details of which are out of the scope of this brief discussion; the compression algorithm is however the same.

the notion of *k-th order empirical entropy* of a string S, denoted by $\mathcal{H}_k(S)$. In Chapter 10 we talked about the case $k = 0$, which depends on the frequencies of the individual symbols occurring in S. With $\mathcal{H}_k(S)$ we wish to empower the entropy definition by considering the frequencies of k-grams in S, thus taking into account sequences of symbols, hence the *compositional structure* of S.

More precisely, let S be a string over an alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_h\}$, and let us denote by n_ω the number of occurrences of the substring ω in S. We use the notation $\omega \in \Sigma^k$ to specify that the length of ω is k. Given this notation, we can define

$$\mathcal{H}_k(S) = \frac{1}{|S|} \sum_{\omega \in \Sigma^k} \left(\sum_{i=1}^{h} n_{\omega\sigma_i} \log \left(\frac{n_\omega}{n_{\omega\sigma_i}} \right) \right) \tag{11.1}$$

A compression algorithm is then defined to be *coarsely optimal* iff, for all k there exists a function $f_k(n)$ tending to 0 as $n \to \infty$ and such that, for all sequences S of increasing length, it holds that the compression ratio of the evaluated algorithm is at most $\mathcal{H}_k(S) + f_k(|S|)$.

Plotnik *et al.* [6] proved the coarse optimality of LZ78; Kosaraju and Manzini [3] noticed that the notion of coarse optimality does not necessarily imply a good algorithm because, if the entropy of the string S approaches zero, the algorithm can compress badly. This observation makes the par with the one we made for Huffman, related to the extra-bit needed for each encoded symbol. That extra-bit was ok for large entropies, but it was considered bad for entropies approaching 0.

LEMMA 11.1 There exist strings for which the compression ratio achieved by LZ78 is at least $g(|S|)\mathcal{H}_0(S)$, with $g(n)$ such that $\lim_{n\to\infty} g(n) = \infty$.

Proof Consider the string $S = 01^{n-1}$, which has entropy $\mathcal{H}_0(S) \in \Theta(\frac{\log n}{n})$. It is easy to see that LZ78 parses S with $\Theta(\sqrt{n})$ phrases. Thus we get $g(n) = \frac{\sqrt{n}}{\log n}$.

To circumvent these inefficiencies, Kosaraju and Manzini introduced a stricter version of optimality, called λ-optimality: it applies to an algorithm whose compression ratio can be bounded by $\lambda\mathcal{H}_k(S) + o(\mathcal{H}_k(S))$. As the previous lemma clearly demonstrates, LZ78 is not λ-optimal, however there exists a modified version of LZ78 combined with run-length compression (RLE) that is 3-optimal with respect to $\mathcal{H}_0$, but cannot be λ-optimal for any $k \geq 1$.

Let us now turn our attention to LZ77, which seems more powerful than LZ78, given that its dictionary is larger. The practical variant of LZ77 that uses a fixed-size compression window is not much good, and actually worse than LZ78:

LEMMA 11.2 The LZ77 algorithm, with a bounded sliding window, is not coarsely optimal.

Proof We will show that, for each size L of the sliding window, we can find a string S for which the compression ratio exceeds the k-th order entropy. Consider in fact the string $(0^k1^k)^n1$ of length $2kn + 1$, where we choose $k = L - 1$. Due to the sliding window, LZ77 parses S in the following way:

$$\underline{0}\ \underline{0^{k-1}1}\ \underline{1^{k-1}0}\ \underline{0^{k-1}1} \ldots \underline{1^{k-1}0}\ \underline{0^{k-1}1}\ \underline{1^k}$$

Every phrase has then length up to k, splitting the input in $\Theta(n)$ phrases. In order to compute $\mathcal{H}_k(S)$ we need to work on all different k-length substrings of S, which are $2k$: $\{0^i1^{k-i}\}_{i=1\ldots k} \cup \{1^i0^{k-i}\}_{i=1\ldots k}$. Now, all strings in the form 0^i1^{k-i} are always followed by a 1. Similarly, for $i < k$ all strings 1^i0^{k-i}

are always followed by a 0. Only the string 1^k is followed $n-1$ times by a 0, and once by a 1. So we can split the sum over the k-grams ω within the definition of $\mathcal{H}_k(S)$ into 4 parts:

$$\begin{array}{lll} \omega \in \{0^i 1^{k-i}\}_{i=1\ldots k} & \rightarrow n_{\omega 0} = 0 & n_{\omega 1} = n \\ \omega \in \{1^i 0^{k-i}\}_{i=1\ldots k-1} & \rightarrow n_{\omega 0} = n & n_{\omega 1} = 0 \\ \omega = 1^k & \rightarrow n_{\omega 0} = n-1 & n_{\omega 1} = 1 \\ \text{else} & \rightarrow n_{\omega 0} = 0 & n_{\omega 1} = 0 \end{array}$$

It is now easy to calculate that

$$|S|\,\mathcal{H}_k(S) = \log n + (n-1)\log\frac{n}{n-1} \in \Theta(\log n)$$

and the lemma follows.

Nevertheless it does exist a modified LZ77, with no sliding window, which is coarsely optimal and also 8-optimal with respect to $\mathcal{H}_0$. However it is not λ-optimal for any $k \geq 1$:

LEMMA 11.3 There exist strings for which the compression ratio of LZ77, with no sliding window, which is at least $g(|S|)\,\mathcal{H}_1(S)$, with $g(n)$ such that $\lim_{n\to\infty} g(n) = \infty$.

Proof Consider the string $10^k\, 2^{2^k}\, 1\, 101\, 10^2 1\, 10^3 1 \ldots 10^k 1$ of length $2^k + O(k^2)$, and compression lower bound $|S|\,\mathcal{H}_k(S) = k\log k + O(k)$. The string is parsed with $k+4$ words:

$$\underline{1}\;\underline{0}\;\underline{0^{k-1}2}\;\underline{2^{2^k-1}1}\;\underline{101}\;\underline{10^2 1}\ldots\underline{10^k 1}$$

The problem is that the last k phrases refer back to the beginning of S, which is 2^k characters away. This generates $\Omega(k)$ long phrases, thus an overall output size of $\Omega(k^2)$.

So this variant of LZ77 is better than LZ78, as expected, but not yet good as we would like to for $k \geq 1$. The next chapter will introduce the Burrows-Wheeler Transform, proposed in 1994, which allows to surpass the inefficiencies of LZ-based methods by devising a novel approach to data compression which achieves λ-optimality, for very small λ and simultaneously for all $k \geq 0$. It is therefore not surprising that the BWT-based compressor `bzip2`, available in most Linux distributions, produces a more succinct output than `gzip`.

References

[1] Jon Bentley and Doug Mc Ilroy Data compression using long commons strings. *Proc. IEEE Data Compression Conference (DCC)*, 287-295, 1999.

[2] Richard Karp and Michael Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):319-327, 1987.

[3] S. Rao Kosaraju and Giovanni Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *Siam Journal on Computing*, 29(3):893-911, 1999.

[4] Paolo Ferragina, Igor Nitto, and Rossano Venturini. On the bit-complexity of lempel-ziv compression. In *Procs of the ACM-SIAM Symposium on Algorithms (SODA)*, pages 768–777, 2009.

[5] Steven Pigeon. An optimizing lossy generalization of LZW. *Procs of IEEE Data Compression Conference*, 509, 2001.