VARIABLE TO FIXED LENGTH ADAPTIVE SOURCE CODING - LEMPEL-ZIV CODING

6.441 Supplementary Notes 2a, 2/17/94

1) THE LEMPEL-ZIV ALGORITHM

An adaptive source coder, or a universal encoder, is designed to compress data from any source. The Lempel-Ziv [LZ76, ZL77,ZL78] coding strategy is essentially a variable to fixed length code containing a parsing dictionary of source strings (as with the Tunstall codes), but this dictionary changes dynamically. The algorithm as described here and in [ZL78] is primarily appropriate for asymptotic analysis; more practical versions are discussed in [Sto88]. More recent Lempel Ziv encoders use the strategy of [ZL77], which is a sliding window version as discussed in class.

Let $u_1, u_2, ..., u_n$ be the source sequence to be encoded. Initially we view n as ∞, and later we can view n either as the total number of letters to be encoded or as a parameter, where the encoder encodes n characters and then starts over with the next n characters, etc. The general idea is to build a Tunstall code adaptively, starting with a dictionary that contains just the single letters of the K letter source alphabet. The dictionary is then enlarged as the encoding proceeds. The rule for enlarging the dictionary is essentially an adaptive version of the rule used to construct the Tunstall code. That is, in constructing a Tunstall code, one successively takes the most probable word in the current dictionary and replaces it with all K single letter extensions of that word. In the Lempel-Ziv code, each time a word in the current dictionary is encoded, that word is replaced in the dictionary with all of its single letter extensions[1].

EXAMPLE 1: Consider the sequence a a a a b ... from the alphabet {a,b,c}. Fig. 1 illustrates how this is parsed into a | a a | a b | ... and how the dictionary tree grows as this parsing takes place. Each time the encoder parses another segment, it generates a binary

[1]In [ZL78], and in most of the literature on this topic, the dictionary is regarded as the set of intermediate nodes above, and the encoding is viewed as sending a dictionary entry followed by a single character (i.e., in our terms sending the leaf node of the tree). The distinction does not affect the algorithm, which is exactly the same in both cases, but allows us to bring out clearly the relationship between Tunstall codes and Lempel-Ziv codes.

code word for that segment. We postpone the question of how the mapping from dictionary entries to code words takes place until later.

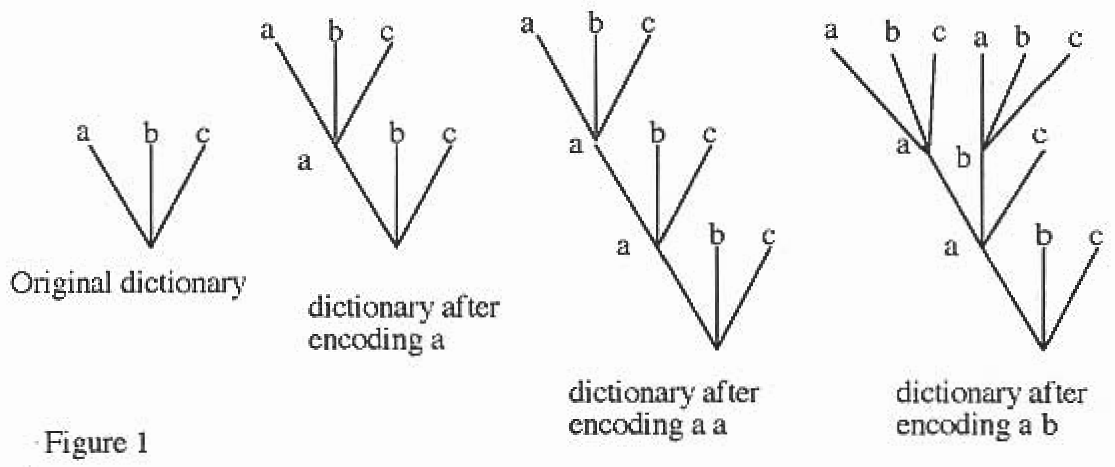
a b c
Original dictionary
dictionary after encoding a
dictionary after encoding a a
dictionary after encoding a b

Figure 1

EXAMPLE 2: Consider an unending sequence of 0's from the binary alphabet {0, 1}. This gets parsed into 0 | 0 0 | 0 0 0 | 0 0 0 0 | 0 0 0 0 0 | .... The corresponding code tree is shown in Figure 2. Note that the number of source letters involved in the first c parsed strings is 1+2+3+...+c, or c(c+1)/2. Thus as the length n of the source sequence increases, the number of parsed strings grows roughly as $\sqrt{2n}$ (see Figure 2). It is not difficult to imagine that this example yields the fastest possible increase in the size of the individual parsed strings with n and the slowest possible growth in the total number of parsed strings with n (see Exercise 1 at the end of these notes).

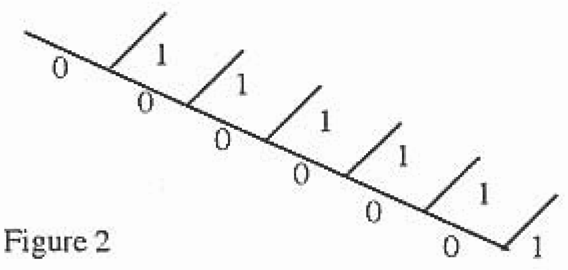

Figure 2

EXAMPLE 3: Consider a sequence made up of the concatenation of all binary strings, starting with strings of length 1, then strings of length 2, etc. Thus the sequence, using spaces to make the structure more apparent visually, is 0 1 00 01 10 11 000 001 010 ... . Note that the Lempel-Ziv algorithm parses this sequence according to the spaces above (i.e., into the binary strings being concatenated in the construction). The dictionary tree, after encoding the part of the sequence shown above, is illustrated in Figure 3. Note that the size of the parsed strings grows as $\log_2 n$ where n is the length of the input sequence already of the individual parsed strings with n and the fastest possible growth in the total number of parsed strings with n (see Lemma 1 in section 3).

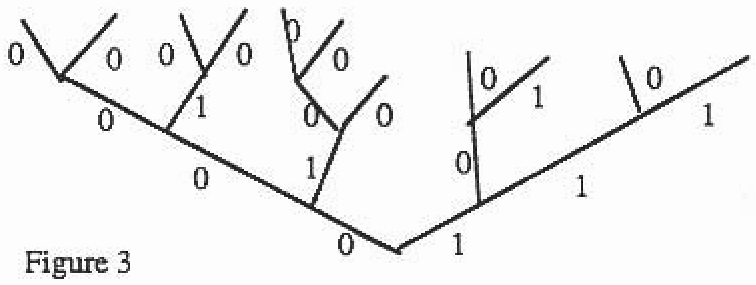

Figure 3

Now consider the encoding of dictionary entries into binary strings. In practice, Lempel-Ziv encoding is usually done with a maximum dictionary size, with some rule to replace old dictionary entries with new entries. To understand the asymptotic operation of the algorithm, however, we want to view the dictionary as simply growing with time. After c-1 parses, the size of the dictionary is K+(c-1)(K-1), where K is the source alphabet size. Thus $\lceil \log_2[K+(c-1)(K-1)] \rceil$ bits are required for a fixed length encoding of the cth parsed string. This means that all code words at a given time have the same length, but the code word length gradually increases with time (i.e., with successive parses). The particular mapping of dictionary entries into binary code words is simply a matter of implementation convenience, but must of course follow some given algorithm.

Note that when the decoder attempts to decode the encoded message stream, it can duplicate the actions of the encoder. That is, for example 3 above, the encoder might map the initial string 0 into the code word 0 (since the size of the initial dictionary is 2). The next string, 1, must be mapped into a code word of length two since the size of the dictionary is now 3. The decoder, on seeing the initial 0 in the encoded sequence, decodes the source letter 0 and also knows that the dictionary now contains {00, 01, 1}. The decoder also knows the algorithm for mapping dictionary entries into code words, and knows that the next code word will have length 2. Thus the decoder can decode the second and third encoded binary digits into the source string 1 and again enlarge the dictionary. In this way, the decoder always knows what the current dictionary and what the current code word length is at the beginning of trying to decode a new code word.

In general, we can view the dictionary at any time as a tree. The terminal nodes are strings that have never been used up to the given time and each intermediate node has occurred at least once in the parsing of the currently entered input sequence. In fact the number of times that an intermediate node has been used as a prefix of one of the parsed segments of the input is precisely equal to the number of intermediate nodes that stem from the given intermediate node (including the node itself).

Thus if one visualizes the Lempel-Ziv technique as attempting to choose equi-probable dictionary entries (i.e., doing the same thing as the Tunstall code, except without prior knowledge of the probabilities), one sees that there is very little statistical evidence for the equi-probability of the leaves. On the other hand, for an intermediate node not too far from the root, the size of the tree emanating from that node, relative to the size of the entire tree, should be very close to the probability of the corresponding string. Thus, one gets the intuition that the tree generated by the Lempel-Ziv algorithm is internally similar to the tree generated by a Tunstall code with known probabilities; the difference is in the "uneven growth" of the Lempel-Ziv tree close to the leaves.

We now turn to analyzing the algorithm and making the above intuitive notions more precise. Let u_1^∞ denote the input string $u_1, u_2, \ldots, u_n, \ldots$ and suppose that the Lempel-Ziv algorithm parses u_1^∞ into the strings $u_1^{m_1}, u_{m_1+1}^{m_2}, \ldots, u_{m_{k-1}+1}^{m_k}$ .... With this notation, we can state the algorithm precisely.

LEMPEL-ZIV ENCODING ALGORITHM:

> The intial set of strings in the dictionary is the set of all single letter strings of the alphabet; m=1
>
> 1) Find n such that u_m^n is a string in the dictionary; generate the code word for that entry.
>
> 2) Remove u_m^n from the dictionary and add the concatenation $(u_m^n a_i)$ to the dictionary for each letter a_i in the source alphabet. Set $m = n+1$ and goto step 1.

In the algorithm as stated, the input sequence is infinite, whereas with a finite input sequence u_1^n, there is a problem with the last string. That is, after parsing the next to last string, say the c-1st, the encoder reads in the final source letters $u_{m_{c-1}+1}, \ldots, u_n$. This might be just a prefix in the dictionary, leading to the need for a special rule for encoding the last segment generated by the parsing rule. One possibility is to use any dictionary entry for which $u_{m_{c-1}+1}, \ldots, u_n$ is a prefix. The decoder, if it knows that only n letters are being encoded, simply strips off the final letters. Another possibility is to provide code words for all the intermediate nodes in the dictionary tree as well as for the leaves (i.e., the real dictionary entries according to the algorithm above).

We have already observed that $\lceil \log_2[K+(c-1)(K-1)] \rceil$ bits are required to represent a dictionary entry after c-1 parses have occurred. If code words are also provided for all intermediate nodes, the size of the dictionary plus intermediate nodes is cK, so that $\lceil \log_2[Kc] \rceil$ bits are required. Since the latter expression is simpler, we use it for an upper bound on the length of the code words which is valid whether or not code words are provided for intermediate nodes. Since at most this many binary letters are used for each segment in the encoding, we see that at most $c\lceil \log_2[Kc] \rceil$ bits are required by the Lempel-Ziv code to represent u_1^n. Upper bounding the ceiling function above by adding 1, we have proven the following theorem:

THEOREM 1: Let $c_{LZ}(u_1^n)$ be the number of strings into which the Lempel-Ziv algorithm parses a given sequence u_1^n. The length $L_{LZ}(u_1^n)$ of the encoded output then satisfies

$$L_{LZ}(u_1^n) \le c_{LZ}(u_1^n) \log\{2Kc_{LZ}(u_1^n)\} \qquad (1)$$

The upper bound here seems very loose since the dictionary starts out small, and short code words could be used to encode the first few strings. Since the number of bits required to encode a segment is a logarithmic function of the dictionary size, however, this saving is rather unimportant.

Theorem 1 relates the length of the encoded sequence (and thus the efficiency of the code) to the number of strings that the parser generates, but it gives us very little insight into the relation between the input sequence and the number of strings. Example 2 above illustrated a particular input (all 0's) in which $c_{LZ}(u_1^n)$ grows as the square root of n, and thus $L_{LZ}(u_1^n)/n$ goes to 0 with increasing n. It is not surprising that very few code letters are required per source letter for this source sequence, and we see that successive inputs are being compressed more and more as the encoder "learns" just how boring the input is. Example 3 illustrated an input where $c_{LZ}(u_1^n)$ grows with n as n/log n, and in this case no compression at all occurs.

We now go on to show that the compression achieved with the Lempel-Ziv encoder is asymptotically as good as can be achieved with any finite state encoder that knows the source statistics (or even knows the exact source sequence). Because of this result, we see that $c_{LZ}(u_1^n)$ must be in some sense a fundamental characterization of the "complexity" of u_1^n, and that this characterization is independent of any assumed stochastic model from which u_1^n might have arisen. Note that we can only hope for such a result asymptotically as $n\to\infty$. For example, if we knew that the source would produce u_1^n with probability 1, then we could build an encoder that would represent u_1^n with just one binary digit (or even no binary digits), and the decoder would simply remember u_1^n. The problem here is that such an encoder/decoder could be constructed for any particular value of n, but the decoder could not be constructed to generate a continuing binary encoded output that would produce u_1^∞ in the limit. The mathematical artifice that Ziv and Lempel use to approach these limiting questions is that of looking at the best possible finite state encoders for encoding u_1^n.

2) FINITE STATE ENCODERS

The class of finite state encoders is a generalization of both the fixed length to variable length and the variable to fixed length encoders we have considered up to now (although we restrict ourselves to binary encoded sequences; the generalization to arbitrary code alphabets should be obvious). As each successive letter from the source sequence enters the encoder, the encoder responds to the incoming source letter and its current state by going into a new state and by emitting a string (perhaps empty) of binary output letters. To put this more mathematically, a finite state encoder with $S\geq 1$ states is defined by two functions, g(u,s) and h(u,s); g(u,s) maps an input letter u and a state s, $1\leq s\leq S$ into a new state s', $1\leq s'\leq S$, and h(u,s) maps an input letter u and a state s, $1\leq s\leq S$ into a binary output string (perhaps the empty string). Thus, letting $u_1, u_2, \ldots$ be the input sequence and letting $s_0, s_1, s_2, \ldots$ denote the state sequence for the encoder, the state sequence is determined (for a given an initial state s_0), by $s_i = g(u_i, s_{i-1})$, $i\geq 1$. Similarly, the ith output string, $\vec{y}_i$, is given by $\vec{y}_i = h(u_i, s_{i-1})$ for $i\geq 1$. The encoded output is the concatenation of $\vec{y}_1, \vec{y}_2, \ldots, \vec{y}_i, \ldots$. In what follows, we extend these functions to denote the concatenated output $\vec{y}_m^n$ from a string of letters u_m^n by $\vec{y}_m^n = h(u_m^n, s_{m-1})$ and we denote the final state by $s_n = g(u_m^n, s_{m-1})$.

Since these output strings can be null, such a finite state encoder (with enough states) can be used to implement an arbitrary variable length to variable length encoder. Viewing a variable length to variable length encoder as a parsing dictionary followed by a mapping from dictionary entries to encoded strings, the states can be used to represent the intermediate nodes in the dictionary tree (with the root node used as the starting state). If the intermediate node extended by the current input letter is another intermediate node, the new state corresponds to the new intermediate node and the output is the null string. If the intermediate node extended by the current input is a leaf node, then the new state is the root node and the output is the code word corresponding to the given leaf node (this assumes that the dictionary satisfies the prefix condition; implementing a non-prefix condition dictionary by a finite state encoder is left as an exercise). The Lempel-Ziv encoder, however, is not a finite state encoder since the dictionary grows without bound. We return to this distinction later.

By unique decodability for a finite state encoder, we mean that for any two distinct input strings, say u_m^n and $\tilde{u}_m^n$, and any starting state s_{m-1}, either the two input strings lead to different final states, $g(u_m^n,s_{m-1}) \neq g(\tilde{u}_m^n,s_{m-1})$, or lead to different output strings, $h(u_m^n,s_{m-1}) \neq h(\tilde{u}_m^n,s_{m-1})$. Note that it is doesn't necessarily cause a problem for distinct input strings to lead to the same output if they lead to different states; for example, in a variable to variable length encoder, all strings corresponding to intermediate nodes in the parsing dictionary would lead to null outputs. If distinct inputs lead to both the same state and the same output, however, it would be impossible for a decoder to ever distinguish those inputs, no matter what the subsequent input and output. Another way to express the condition of unique decodability is to insist that a decoder, given s_{m-1}, $\vec{y}_m^n$, and s_n, must be able to uniquely reconstruct u_m^n.

In what follows, we implicitly include unique decodability as part of the definition of a finite state encoder. The class of "uniquely decodable finite state encoders" as defined here is a little larger than desirable. In particular, the definition guarantees that a decoder, given s_0, $\vec{y}_1^n$, and s_n, could uniquely decode u_1^n, but it does not necessarily mean that u_1^n can be decoded for any n in the absence of s_n, which is not normally known to the decoder. This does not matter for us since the properties established for this class are also valid for the included class of encoders for which these "end effects" can be resolved.

3) DISTINCT PARSING STRINGS

Define $c(u_1^n)$ as the maximum number of distinct strings that u_1^n can be parsed into, counting the null string. In what follows, we develop bounds, in terms of $c(u_1^n)$, on the compression achieved both by Lempel-Ziv encoding and by arbitrary finite state encoders. We will show, following [ZL78], that $c(u_1^n)$ provides a fundamental measure of the compressibility of u_1^n. We shall not be concerned with calculating $c(u_1^n)$, but only with bounds and asymptotic results. First note that the Lempel-Ziv algorithm parses u_1^n into $c_{LZ}(u_1^n)$ strings, all of which are distinct except perhaps the last. By combining the last two strings, we obtain $c_{LZ}(u_1^n) - 1$ distinct strings, and by adding the null string, which is not included in $c_{LZ}(u_1^n)$, we get $c_{LZ}(u_1^n)$ distinct strings. Thus

$$c_{LZ}(u_1^n) \le c(u_1^n) \tag{2}$$

Combining this with (1),

$$L_{LZ}(u_1^n) \le c(u_1^n) \log \{2Kc(u_1^n)\} \tag{3}$$

We next find a lower bound on the length of code word for u_1^n that applies for any finite state encoder with S states.

THEOREM 2: For any uniquely decodable finite state encoder of at most S states, and for any initial state, the length $L_S(u_1^n)$ of the encoding of u_1^n satisfies

$$L_S(u_1^n) \ge c(u_1^n) \log\left[\frac{c(u_1^n)}{4S^2}\right] \tag{4}$$

Before proving this, we first discuss what the theorem means and then present a lemma needed in the proof. The finite state encoder in the theorem is arbitrary (subject to the constraint of S states), and thus could be designed under the assumption that u_1^n will occur. On the other hand, because of the unique decodability requirement, the encoder must also provide for the remote possibility of all possible sequences from the source. What this means is that if the encoder attempts to encode u_1^n into a single binary letter, then the encoder must at least recognize u_1^n in the input; this requires at least n states. Thus, for fixed S and sufficiently large n, one cannot encode u_1^n into a single binary digit.

Note that the theorem only asserts that $L_S(u_1^n)$ is positive when $c(u_1^n) > 4S^2$, and the above example helps to explain why such a theorem can only be meaningful when n is large, and therefore when $c(u_1^n)$ is large.One should also note that the number of states in an encoder is typically very large. The theorem is not intended as a practical bound for small values of n, but rather to see what happens asymptotically as $n \to \infty$.

LEMMA 1: Let n(c) be the minimum possible length of the concatenation of c distinct strings from an alphabet of $K \geq 2$ symbols. Then

$$n(c) \log K \;\geq\; c \log \frac{c}{4} \tag{5}$$

Proof of lemma 1: We claim that the length of the concatenation is minimized if the set of strings contains all strings of length less than some integer j and no strings of length greater than j. To see this, suppose the contrary; then there must be some string not in the set of c that is shorter than some string in the set; removing the longer string and replacing by the shorter string reduces the length of the concatenation, exhibiting a contradiction. Letting r be the number of strings of length j in the set, we then have

$$c = r + \sum_{i=0}^{j-1} K^i \quad ; \quad 0 \leq r \leq 2^j \tag{6}$$

$$n(c) = jr + \sum_{i=0}^{j-1} iK^i \tag{7}$$

Consider the special case in which r = 0. We can sum (6) and (7) to get

$$c = \frac{K^j - 1}{K-1} \tag{8}$$

$$n(c) = \frac{(jK-K-j)K^j + K}{(K-1)^2} \tag{9}$$

Solving (8) for K^j and for j, we get

$$K^j = c(K-1)+1 \quad ; \quad j = \frac{\log[c(K-1)+1]}{\log K}$$

Substituting this into (9) and simplifying, we get

$$n(c) = \left[\frac{c(K-1)+1}{(K-1)\log K}\right]\log[c(K-1)+1] - \frac{Kc}{K-1}$$

Multiplying both sides by log K and recognizing that x log x is increasing in x, we can lower bound this as

$$n(c)\log K \ge c\log[c(K-1)] - \frac{Kc\log K}{K-1} = c\log c + c\left[\log(K-1) - \frac{K\log K}{K-1}\right]$$

The final term in brackets above is equal to -2 for K=2 and is increasing in K for $K\geq2$. Thus,

$$n(c)\log K \ge c\log(c) - 2c = c\log(c/4)$$

Thus (5) is satisfied in the special case r=0. We also have equality for the special case $r = 2^j$, since this case is identical to choosing j one larger and choosing r=0. Finally, from (6) and (7), we see that n(c) is linear in c for fixed j and $0\leq r\leq 2^j$. Since the right hand side of (5) is convex $\cup$ in c, it follows that (5) is satisfied for all r.

Proof of Theorem 2: Let u_1^n be an arbitrary input string with a parsing into $c(u_1^n)$ distinct substrings. Denote the states as $\{1, 2,...,S\}$ and let c_{ij} be the number of substrings in the above parsing for which the original state is i and the final state is j. Thus $\Sigma_{ij}\, c_{ij} = c(u_1^n)$. The binary output from each of these strings for a given i,j must be distinct (because of the unique decodability), and thus, using (5) with K=2, the aggregate length L_{ij} of the output strings for a given starting state i and final state j satisfies

$$L_{ij} \ge c_{ij}\ \log(c_{ij}/4) \tag{10}$$

The total length $L_S(u_1^n)$ of encoded output is $\Sigma_{ij}\, L_{ij}$, so

$$L_S(u_1^n) \ge \sum_{i=1,\ j=1}^{S,S}\left[c_{ij}\log\frac{c_{ij}}{4}\right] \tag{11}$$

The right hand side of (11) is minimized over the choice of $\{c_{ij}\}$, subject to $\Sigma_{ij}\, c_{ij} = c(u_1^n)$, if each c_{ij} is equal to $c(u_1^n)/S^2$. Substituting this in (11) yields (4).

Note that the lower bound on $L_S(u_1^n)$ given in the theorem is zero at the point where $c(u_1^n) = 4S^2$. For $c(u_1^n)$ very large, however, the term S^2 becomes less significant. We note, similarly, that the term 2K in the bound on $L_{LZ}(u_1^n)$ appears to be insignificant for large $c(u_1^n)$ This suggests defining the compressibility $\lambda(u_1^n)$ of the sequence u_1^n as

$$\lambda(u_1^n) = \frac{c(u_1^n)\log[c(u_1^n)]}{n} \tag{12}$$

THEOREM 3:

$$\frac{L_{LZ}(u_1^n)}{n} - \lambda(u_1^n) \le \frac{(\log K)(\log 2K)}{\log[c(u_1^n)/4]} \tag{13}$$

$$\lambda(u_1^n) - \frac{L_S(u_1^n)}{n} \le \frac{(\log K)(\log 4S^2)}{\log[c(u_1^n)/4]} \tag{14}$$

The first part of the theorem says that the number of code letters per source letter with the Lempel-Ziv algorithm is arbitrarily little more than $\lambda(u_1^n)$ and that this excess goes to 0 with increasing $c(u_1^n)$. The rate of approach in (13) is only logarithmic in $c(u_1^n)$, but it depends on u_1^∞ only through $c(u_1^n)$. The second part of the theorem says that the minimum number of code letters per source letter achievable by a uniquely decodable finite state encoder, even one selected for the particular sequence u_1^n, is arbitrarily little less than $\lambda(u_1^n)$. The rate of approach is again logarithmic in $c(u_1^n)$, but is highly dependent on S (as we have already seen that it must be).

Proof of Theorem 3: From (3) and (12), we see that

$$\frac{L_{LZ}(u_1^n)}{n} - \lambda(u_1^n) \le \frac{c(u_1^n)}{n}\log(2K) \tag{15}$$

Similarly, from (4) and (12), we have

$$\lambda(u_1^n) - \frac{L_S(u_1^n)}{n} \le \frac{c(u_1^n)}{n}\log(4S^2) \tag{16}$$

To obtain a bound on $c(u_1^n)$, we rewrite (5) with c equal to $c(u_1^n)$,

$$n \log K \geq c(u_1^n) \log \frac{c(u_1^n)}{4} \;; \qquad \frac{c(u_1^n)}{n} \leq \frac{\log K}{\log[c(u_1^n)/4]} \tag{17}$$

Substituting this in (15) and (16) completes the proof.

From Exercise 1 at the end of this note, $c(u_1^n) \geq \sqrt{n}$. Applying this inequality to (13) and (14), we have the following corollary:

COROLLARY 3.1:

$$\frac{L_{LZ}(u_1^n)}{n} - \lambda(u_1^n) \leq \frac{2(\log K)(\log 2K)}{\log[n/16]} \tag{18}$$

$$\lambda(u_1^n) - \frac{L_S(u_1^n)}{n} \leq \frac{2(\log K)(\log 4S^2)}{\log[n/16]} \tag{19}$$

This shows that the compressibility achievable with the Lempel-Ziv algorithm, i.e., $L_{LZ}(u_1^n)/n$, is almost as small as $\lambda(u_1^n)$, with the upper bound on the difference going to 0 with n. Similarly $\lambda(u_1^n)$ is almost as small as $L_S(u_1^n)/n$. It is very surprising that the right hand side of (18) and (19) depends on u_1^n only through n. One should note[2] that these quantities approach 0 very slowly with increasing n. One can combine (18) and (19) to obtain

$$\frac{L_{LZ}(u_1^n)}{n} - \frac{L_S(u_1^n)}{n} \leq \frac{2(\log K)(\log 8KS^2)}{\log[n/16]} \tag{20}$$

This says that given any finite state encoder of S states (perhaps designed with knowledge of u_1^∞), the compressibility achieved by the Lempel-Ziv algorithm on u_1^n is arbitrarily little more than that achieved by the finite state encoder, with the difference approaching 0 as n approaches infinity.

[2]The term log[n/16] in the denominator of (18) and (19) can be replaced with $2 \log[n/4]\,(1-\varepsilon(n))$ where $\varepsilon(n) \to 0$ as $n \to \infty$. To see this, use the result in Exercise 4 in place of (17) in the proof of theorem 3.

The compressibility of an infinite length sequence u_1^∞ can now be defined by

$$\lambda(u_1^\infty) = \limsup_{n \to \infty} \lambda(u_1^n) \tag{21}$$

This yields the following corollary to theorem 3:

COROLLARY 3.2:

$$\limsup_{n \to \infty} \frac{L_{LZ}(u_1^n)}{n} \le \lambda(u_1^\infty) \tag{22}$$

$$\limsup_{n \to \infty} \frac{L_S(u_1^n)}{n} \ge \lambda(u_1^\infty) \quad \text{for every finite S} \tag{23}$$

One should not read more into this corollary than it says. In particular, $\lambda(u_1^n)$ need not approach a limit with n, and one can construct sequences u_1^∞ for which $\lambda(u_1^n)$ oscillates forever. Whether u_1^∞ is meaningful in these situations is questionable, but because of corollary 3.1, the oscillations in $L_{LZ}(u_1^n)/n$ and $L_S(u_1^n)/n$ are bounded by those of $\lambda(u_1^n)$. This corollary precisely defines the sense in which the number of encoded bits per source letter achieved with the Lempel-Ziv algorithm is, asymptotically, at least as small as that with any finite state encoder. We also note that the Lempel-Ziv algorithm (for operation on u_1^n for any fixed n) can be implemented by a finite state encoder.

The interpretation of this is as follows: (23) says that for any given S, no matter how large, any sequence u_1^∞, and any $\varepsilon \ge 0$, there is a large enough n so that the best finite state uniquely decodable encoder (for u_1^n) requires at least $\lambda(u_1^\infty) - \varepsilon$ bits per source letter for encoding. From (20), the Lempel-Ziv encoder, requiring some finite number S' states, will require at most $\lambda(u_1^\infty) + \varepsilon$ bits per source letter to encode u_1^n. The rub is that S' might be very much larger than S. In other words, the price paid for the universally good performance of the Lempel-Ziv encoder is that it requires many more states than the best finite state encoder designed for the particular source sequence.

There is now a subtlety that we must deal with: the Lempel-Ziv encoder requires a finite number of states for each finite n, but the number of states becomes unbounded as $n \to \infty$. Ziv and Lempel's way of dealing with this issue is to use the Lempel-Ziv strategy for n input letters, for some large block length n, and then to start all over again for each successive set of n input letters. From a practical standpoint, it would clearly be preferable to build the dictionary out to some fixed size, and then to develop some algorithm for replacing dictionary entries that are no longer useful with entries that occur more often; from the standpoint of establishing theoretical properties, however, starting over with each block of n has great advantages. To analyze what happens when an input sequence of say mn letters is encoded n letters at a time, we use theorem 3 on successive n-tuples. Applying Eq. (20) to the ith n-tuple of inputs,

$$\frac{L_{LZ}\left(u_{in+1}^{in+n}\right)}{n} - \frac{L_S\left(u_{in+1}^{in+n}\right)}{n} \le \frac{2(\log K)(\log 8KS^2)}{\log[n/16]} \tag{24}$$

Now if we consider any particular finite state encoder of S states, we see that

$$L_S\left(u_1^{mn}\right) = \sum_{i=1}^{m} L_S\left(u_{in+1}^{in+n}\right) \tag{25}$$

Note that each of the terms on the right of (25) correspond to some particular starting state, which we have heretofore suppressed. Since Theorem 3 applies to any starting state, we can continue to suppress it. Summing (24) over i, and using (25), we have

$$\frac{\sum_{i=1}^{m} L_{LZ}\left(u_{in+1}^{in+n}\right)}{mn} - \frac{L_S\left(u_1^{mn}\right)}{mn} \le \frac{2(\log K)(\log 8KS^2)}{\log[n/16]} \tag{26}$$

The first term is the compressibility of a Lempel-Ziv encoder, operating on m blocks of n source letters at a time, and thus requiring only a finite number of states (which depends on n but not m). The second term is the compressibility of an arbitrary uniquely decodable finite state encoder operating on the same sequence of mn letters. The final term depends neither on m nor on u_1^{∞}. For any S and any $\varepsilon > 0$, we can choose n large enough to make the right side of (25) less than ε, and therefore, for any S, the compressibility achieved by the Lempel-Ziv encoding algorithm, using blocks of sufficiently large length n, is at most ε

more than the compressibility of the best finite state encoder of S states. What is remarkable is that the required value of n is independent of u_1^∞.

In a sense, the lower bound $L_S(u_1^n)/n$ on bits per letter for a finite state encoder is very artificial - there is no point to building an encoder for a particular sequence u_1^n. What we have really shown there is that the Lempel-Ziv algorithm asymptotically does as well, in the absence of knowledge of source or source statistics, as the best finite state encoder. The price that is paid for the lack of knowledge, however, is a larger number of states.

4) STATIONARY SOURCES

Next consider the performance of a Lempel-Ziv encoder on an arbitrary (stochastic) stationary source of entropy $H_\infty(U)$. From Eq. 3.5.12 in theorem 3.5.2 of the text, we know that for any $\delta>0$ we can choose an m large enough so that the expected number of encoded binary digits per source letter in a Huffman code on super letters of length m is at most $H_\infty(U)+\delta$. Now consider encoding i successive m-tuples of inputs. Let $L_{Huff}(u_1^{mi})$ be the length of the encoder output for the input u_1^{mi}. As we have just seen,

$$\frac{E\left[L_{Huff}\left(u_1^{mi}\right)\right]}{mi} \le H_\infty(U) + \delta \tag{27}$$

The Huffman encoder on super letters of length m can be implemented as a finite state encoder with $S = (K^m-1)/(K-1)$ states (see Exercise 5). For an arbitrary input sequence u_1^n, let $n = im+j$, where $0 \le j < m$. The Huffman encoder, with input u_1^n, will generate i code words with expected length satisfying (27) and save the final j inputs waiting for additional input letters to make up the next super letter input. The length, $L_{Huff}(u_1^n)$, of output from the Huffman encoder is at least as large as that for the best finite state encoder for u_1^n with S states. Thus, using (19), we have

$$\frac{L_{Huff}\left(u_1^n\right)}{n} \ge \left.\frac{L_S\left(u_1^n\right)}{n}\right|_{S=(K^m-1)/(K-1)} \ge \lambda\left(u_1^n\right) - \frac{4(\log K)\log[2(K^m-1)/(K-1)]}{\log(n/16)}$$

The final term above approaches 0 with n, so that for large enough n,

$$\frac{L_{Huff}\left(u_1^n\right)}{n} \ge \lambda\left(u_1^n\right) - \delta \tag{28}$$

With n = mi+j, 0≤j<m, we have $L_{Huff}(u_1^n) = L_{Huff}(u_1^{mi})$. Thus from (27),

$$\frac{E\left[L_{Huff}\left(u_1^n\right)\right]}{n} = \frac{E\left[L_{Huff}\left(u_1^{mi}\right)\right]}{n} \leq \frac{E\left[L_{Huff}\left(u_1^{mi}\right)\right]}{mi} \leq H_\infty(U) + \delta \tag{29}$$

Taking expected values of both sides of (28) and combining with (29), we have

$$E\left[\lambda\left(u_1^n\right)\right] \leq H_\infty(U) + 2\delta \quad \text{for all sufficiently large n} \tag{30}$$

Next consider a Lempel-Ziv encoder working on blocks of n inputs at a time. Since this can be viewed as a block to variable length encoder, we have

$$H_\infty(U) \leq \frac{E\left[L_{LZ}\left(u_1^n\right)\right]}{n} \tag{31}$$

Taking the expected value of (18) and substituting into (31), we have

$$H_\infty(U) \leq \frac{E\left[L_{LZ}\left(u_1^n\right)\right]}{n} \leq E\left[\lambda\left(u_1^n\right)\right] + \frac{2(\log K)(\log 2K)}{\log(n/16)} \leq \left[\lambda\left(u_1^n\right)\right] + \delta \tag{32}$$

where the final inequality holds for large enough n. Since (30) and (32) hold for arbitrary δ>0, we have proved the following theorem:

THEOREM 4: For any stationary stochastic source,

$$H_\infty(U) = \lim_{n\to\infty} \frac{E\left[L_{LZ}\left(u_1^n\right)\right]}{n} = \lim_{n\to\infty} \left[\lambda\left(u_1^n\right)\right] \tag{33}$$

There is one practical problem with Lempel-Ziv encoders that is slightly concealed by this result: if one implements a Lempel-Ziv encoder with a particular value of n (or, more practically, with a given dictionary size and given replacement rule), then one can always construct a stationary source with too long a memory for the given encoder size. For example, the source could simply repeat with a repetition cycle of length n+1, and the Lempel-Ziv encoder limited to a particular value of n would be totally stymied. In other words, the rate of convergence in (33) depends on the particular stationary source. Another problem is that if the source is not ergodic, then individual sequences will not behave as the expected value. None-the-less, Theorem 4 is an amazing result. It shows that this very simple algorithm can adapt, eventually, to <u>any</u> stationary source, no matter how complex or long term the memory, and compress it to the entropy rate.

5) KOLMOGOROFF-CHAITIN COMPLEXITY

There is another well known measure of how much a source sequence can be compressed known as the Kolmogoroff-Chaitin complexity measure. The question asked here is the same as the question in Lempel-Ziv coding - how many code letters per source letter are needed, in the limit $n \to \infty$, to represent a source sequence u_1^n. The crucial difference is in the type of encoder and decoder allowed. The Lempel-Ziv algorithm focuses on the encoder, whereas Kolmogoroff-Chaitin complexity focuses on decoding and allows an arbitrary universal Turing machine to be used as the decoder[3]. The input to the universal Turing machine is the encoded version of u_1^∞ and the output is u_1^∞ itself. Thus we can view the input as a program plus data, where the program tells the Turing machine how to process the data in generating the output.

More abstractly, we view the Turing machine input as a sequence of binary digits which the machine interprets according to its rules. There are many Turing machine input sequences that could give rise to the required source sequence u_1^∞. For example, the Turing machine input could be a program for a given Huffman decoder followed by the Huffman encoding of u_1^∞. Similarly the Turing machine input could be a program for Lempel-Ziv decoding, followed by the the Lempel-Ziv encoding of u_1^∞. For the given Turing machine T, we can define $L(u_1^n, T)$ as the minimum length input sequence required to generate the output u_1^n. We can then define the Kolmogoroff-Chaitin complexity of u_1^∞, relative to T, as

[3] For the reader unfamiliar with Turing machine theory, it is sufficient to view a Turing machine as a conventional computer with an unlimited memory, viewed as an infinite tape. A universal Turing machine is one that can be programmed to simulate any other Turing machine. Essentially any general purpose computer with unlimited memory (i.e., where memory is added as needed) can be viewed as universal in this sense.

$$\lambda_{KC}\left(u_1^\infty, T\right) = \limsup_{n\to\infty} \frac{L\left(u_1^n, T\right)}{n} \tag{34}$$

Since the input to the Turing machine could be a program for a Lempel-Ziv decoder followed by the Lempel-Ziv encoding of u_1^∞, we see that $\lambda(u_1^\infty) \geq \lambda_{KC}(u_1^\infty, T)$.

We next observe that $\lambda_{KC}(u_1^\infty, T)$ is independent of the universal Turing machine T. The reason is that any universal Turing machine T can simulate any other Turing machine T'. More specifically, there is some input Z(T,T') that instructs T to act like T' in the sense that if input y to T' produces output u_1^n, then input Z concatenated with y will produce output u_1^n from T. Since the fixed input Z is finite and independent of y, the limit in (34) is the same for any T and T', and therefore we drop the T in what follows (note however that before going to the limit, $L(u_1^n, T)$ is very much a function of T; one can always find a universal Turing machine T that produces u_1^n after looking at just one bit of the input.

Example 3, (first given in [LZ78]) clarifies the difference between $\lambda_{KC}(u_1^\infty)$ and $\lambda(u_1^\infty)$. Recall that u_1^∞ is a listing first of all binary strings of length 1, then of length 2, etc., all ordered within their length, i.e., (0)(1)(00)(01)(10)(11)(000)(001)(010)... A Turing machine can be programmed with a finite program to generate this infinite output stream, so that $\lambda_{KC}(u_1^\infty) = 0$. On the other hand, the parsing above shows that $\lambda(u_1^\infty) = 1$.

From an intuitive viewpoint, it is reasonable to view the above sequence as having zero complexity or zero compressibility - after knowing how it is generated, there is simply no additional information in watching the entire boring sequence unfold. There are infinitely many variations on this theme; for example, one could invert the i^{th} digit in each string of length greater than i, which is one variation for each integer i. There are clearly many other types of examples; in fact any finite input to a Turing machine that gives rise to an infinite output is such an example. Unfortunately, there is a famous theorem in Turing machine theory that states that there is no algorithm that will determine which inputs give rise to an infinite output sequence.

A more troubling aspect of Kolmogoroff-Chaitin complexity is the lack of any approach to encoding a given sequence u_1^∞. One must in principle see the entire sequence before determining the minimum length program to generate it, for whenever one looks at a finite sequence u_1^n, it is important to know which universal Turing machine is to generate u_1^n. Even if one is willing to adopt some given Turing machine, there is no way to find the minimum length program to generate u_1^n.

Kolmogoroff-Chaitin complexity is usually viewed as a fundamental topic for theoretical computer science, but it seems less appropriate (since it ignores encoding) to the study of data compression. This is a personal viewpoint, however, that is not universally shared by all information theorists.

References:

Chaitin, G.J., *Information-Theoretic Computational Complexity*, IEEE Trans. IT, Jan. 1974.

Kolmogoroff, A.N., *Logical Basis for Information Theory and Probability Theory*, IEEE Trans. IT, Sept. 1968, pp. 662-664.

Lempel, Abraham & Jacob Ziv, "On the Complexity of Finite Sequences," IEEE Trans. IT, Jan. 1976, pp. 75-81.

Miller, V. S. & M.N. Wegman, "Variations on a theme by Ziv and Lempel," in *Combinatorial Algorithms on Words*, Springer Verlag, (Apostolico and Galil, ed.), 1985, pp 131-140.

Storer, James A., *Data Compression, methods and theory,* Computer Science Press 1988.

Ziv, Jacob & Abraham Lempel, "A Universal Algorithm for sequential data compression," IEEE Trans. IT, May 1977, pp. 337-343.

Ziv, Jacob & Abraham Lempel, "Compression of Individual Sequences via Variable-Rate Coding," IEEE Trans. IT, Sept. 1978, PP. 530-536.

EXERCISES:

1) The object of this problem is to show that $c_{LZ}(u_1^n)$ might be considerably smaller than $c(u_1^n)$ for particular choices of u_1^n. This indicates that Lempel-Ziv coding might compress some sequences considerably more than the bound in theorem 3 indicates.

a) Use the Lempel-Ziv algorithm to parse the binary sequence of length 55 below (i.e., parse from left to right, always selecting the shortest string that has not appeared earlier).

0010110110011010110100110100011010010110100110110100111

Show that $c_{LZ}(u_1^n) = 10$ for $n = 55$.

b) Now take 00 as the leftmost string in a parsing of the sequence above and then parse the rest of the sequence from left to right by always selecting the shortest string that has not appeared before (including 00 as a string that has appeared before). How many strings does u_1^n parse into by this procedure? Explain why this is an lower bound to $c(u_1^n)$.

c) Show that your answer in (b) is equal to $c(u_1^n)$ for n=55.

d) EXTRA CREDIT (I don't know how to solve this either) Extend u_1^n to arbitrarily large n of the form $n=k(k+1)/2$, k integer, in such a way that $c_{LZ}(u_1^n) = k$. Either show that $c_{LZ}(u_1^n)/c(u_1^n) \to 1$ as $n \to \infty$ or show that $\lim\sup_n c_{LZ}(u_1^n)/c(u_1^n) < 1$.

2) Show that $c(u_1^n) \geq \frac{-1 + \sqrt{1+8(n+1)}}{2} \geq \sqrt{n}$.

Solution: Note that any string can be parsed into unique strings by starting with the null string, then a string of length 1, then of length 2, and so forth. After selecting the i^{th} string (which has length i-1), if the remaining string has length less than 2i+1, it is possible that choosing another string of length i will cause the remainder to be one of the selected strings (recall Example 2), and thus this final string is not parsed. We see that if this process gives us c strings, the i^{th}, $i \leq c-1$ will have length i-1, and the c^{th} will have some length between c-1 and 2c-2. Adding up these lengths, we have $n \leq (c-1)(c+2)/2$, thus yielding

$$c(u_1^n) \geq \frac{-1 + \sqrt{1+8(n+1)}}{2} \geq \sqrt{n} \tag{35}$$

3) Verify Eq. (9).