ECOLE POLYTECHNIQUE FEDERALE DE LAUSANNE

School of Computer and Communication Sciences

Handout 11 Information Theory and Coding

Notes on the Lempel-Ziv Algorithm October 17, 2017

Universal Source Coding — Lempel-Ziv Algorithm

Our experience with data compression so far has been of the following type: We are given the statistical description of an information source, we then try to design a system which will represent the data produced by this source efficiently.

In this note we depart from this model, and consider a method which will represent a sequence efficiently without knowing by which means the sequence was produced. For this purpose, rather than assuming a statistical model for the sequence, it makes more sense to imagine that there is only a single sequence: an infinite string u which we wish to represent.

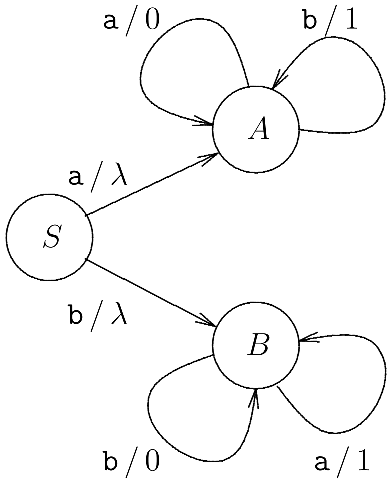
We will first consider the compressibility of an infinite string with a finite state machine. For our purposes, a finite state machine is a device that reads the input sequence one symbol at a time. Each symbol of the input sequence belongs to a finite alphabet $\mathcal{U}$ with $|\mathcal{U}|$ symbols ($|\mathcal{U}| \geq 2$). The machine is in one of a finite number s of states before it reads a symbol, and goes to a new state determined by the old state and the symbol read. We will assume that the machine is in a fixed, known state z_1 before it reads the first input symbol. The machine also produces a finite string of binary digits (possibly the null string) after each input. This output string is again a function of the old state and the input symbol. That is, when the infinite sequence $u = u_1 u_2 \cdots$ is given as the input, the encoder produces $y = y_1 y_2 \cdots$, while visiting an infinite sequence of states $z = z_1 z_2 \cdots$, given by

$$\begin{aligned} y_k &= f(z_k, u_k), \quad k \geq 1 \\ z_{k+1} &= g(z_k, u_k), \quad k \geq 1 \end{aligned}$$

where the function f takes values on the set $\{0,1\}^*$ of finite binary strings, so that each y_k is a (perhaps null) binary string. A finite segment $x_k x_{k+1} \cdots x_j$ of a sequence $x = x_1 x_2 \cdots$ will be denoted by x_k^j, and by an abuse of the notation, the functions f and g will be extended to indicate the output sequence and the final state. Thus, $f(z_k, u_k^j)$ will denote y_k^j and $g(z_k, u_k^j)$ will denote z_{j+1}. Without loss of generality we will assume that any state z is reachable from the initial state z_1 — i.e., that some input sequence will take the machine from state z_1 to z.

To make the question of compressibility meaningful one has to require some sort of an 'invertibility' condition on the finite state encoders. Given the description of the finite state machine that encoded the string, and the starting state z_1, but (of course) without the knowledge of the input string, it should be possible to reconstruct the input string u from the output of the encoder y. A weaker requirement than this is the following: for any state z and two distinct input sequences v_1^m and $\tilde{v}_1^n$, either $f(z, v_1^m) \neq f(z, \tilde{v}_1^n)$ or $g(z, v_1^m) \neq g(z, \tilde{v}_1^n)$. An encoder satisfying this condition will be called *information lossless* (IL). It is clear that if an encoder is not IL, then there is no hope to recover the input from the output, and thus every 'invertible' encoder is IL. [1]

[1] However, as illustrated in Figure 1, an IL encoder is not necessarily uniquely decodable. Starting from state S, two distinct input sequences will leave the encoder in distinct states if they have different first symbols, otherwise they will lead to different output sequences. Thus, the above encoder is IL. Nevertheless, no decoder can distinguish between the input sequences `aaaa`$\cdots$ and `bbbb`$\cdots$ by observing the output $000\cdots$.



A finite state machine with three states S, A and B. The notation `i` /*output* means that the machine produces *output* in response to the input `i`. λ denotes the null output.

Figure 1: An IL encoder which is not uniquely decodable.

We will first derive a lower bound to the the number of bits per input symbol *any* IL encoder will produce when encoding a string u. This lower bound will even apply to IL encoders which may have been designed with the advance knowledge about u. We will then show that a particular algorithm (the Lempel-Ziv algorithm) the design of which *does not* depend on u, does as well as this lower bound. That is to say, a machine that implements the LZ algorithm will compete well against any IL machine in compressing any u. (However, note that a machine that implements LZ will *not* be a finite state machine. The point is to show that LZ performs well — we are not interested in the fairness of the competition.)

We can now define the compressibility of an infinite string u. Given an IL encoder E, the compression ratio for the initial n symbols u_1^n of u with respect to this encoder is defined by

$$\rho_E(u_1^n) = \frac{1}{n} \operatorname{length}(y_1^n),$$

where $\operatorname{length}(y_1^n)$ is the length of the binary sequence y_1^n. (Note that since each y_i is a possibly null binary string $\operatorname{length}(y_1^n)$ may be more or less than n.) The minimum of $\rho_E(u_1^n)$ over the set of all IL encoders E with s or less states is denoted by $\rho_s(u_1^n)$. Observe that $\rho_s(u_1^n) \le \lceil \log_2 |\mathcal{U}| \rceil$. The compressibility of u with respect to the class of IL encoders with s or less states is then defined as

$$\rho_s(u) = \limsup_{n\to\infty} \rho_s(u_1^n).$$

Finally the compressibility of u with respect to IL encoders (or simply the *compressibility*) is defined as

$$\rho(u) = \lim_{s\to\infty} \rho_s(u).$$

Note that since $\rho_s(u)$ is non-increasing in s, the limit indeed exists.

Let us define $m^*(u_1^n)$ as the maximum number of distinct strings that u_1^n can be parsed into, including the null string. (Note that $m^*(u_1^n) \ge 1$.) It turns out that $m^*(u_1^n)$ plays a fundamental role in the compressibility of u.

Before proceeding further, let us note that $m^*(u_1^n)$ is sublinear in n:

Lemma 1. *For any sequence u, $\lim_{n\to\infty} m^*(u_1^n)/n = 0$.*

Proof. Let $v_1, \dots, v_{m^*}$ be a distinct parsing of u_1^n with $m^* = m^*(u_1^n)$. For any $k \geq 1$, the number of strings with length $k-1$ or fewer is $1+|\mathcal{U}|+\dots+|\mathcal{U}|^{k-1} < |\mathcal{U}|^k$. Thus, among the v_i's at least $m^* - |\mathcal{U}|^k$ of them are of length k or more, and consequently $n > k[m^* - |\mathcal{U}|^k]$. Thus, for any k,

$$0 \leq \lim_{n\to\infty} \frac{1}{n} m^*(u_1^n) \leq \lim_{n\to\infty} \frac{1}{n}[n/k + |\mathcal{U}|^k] = 1/k.$$

As k is arbitrary the lemma follows. □

We will also need to the following fact:

Lemma 2. *Suppose $v_1, \dots, v_m$ are binary strings, with no string occurring more than k times. Then, writing $m = \sum_{i=0}^{j-1} k2^i + r$ with $0 \leq r < k2^j$, we have $\sum_{i=1}^m \text{length}(v_i) \geq k\sum_{i=0}^{j-1} i2^i + rj$.*

Proof. The set of binary strings ordered in increasing length consists of: 1 string of length 0, 2 strings of length 1, ..., 2^i strings of length i, ... The shortest total length for the v_i's will be attained if the v_i's are chosen by traversing the set of all binary strings in increasing length, each string repeated k times, until all strings of length $j-1$ or less are repeated k times, and we are left to find $0 \leq r < k2^j$ strings, which are chosen from the set of strings of length j. The lower bound in the lemma is precisely the total length of this optimal collection. □

Lemma 3. *Suppose $v_1, \dots, v_m$ are binary strings, with no string occurring more than k times. Then,*

$$\sum_{i=1}^m \text{length}(v_i) \geq m \log_2 \frac{m}{8k}. \tag{1}$$

Proof. Noting that $\sum_{i=0}^{j-1} 2^i = 2^j - 1$ and $\sum_{i=0}^{j-1} i2^i = (j-2)2^j + 2$, the previous lemma states: writing $m = k2^j - k + r$ with $0 \leq r < k2^j$, the total length of the v_i's is lower bounded by

$$k((j-2)2^j + 2) + rj = (j-2)m + kj + 2r \geq (j-2)m.$$

As $r < k2^j$, we have $m < k(2^{j+1} - 1)$. Rearranging, we get $2^{j+1} > 1 + m/k > m/k$, and thus $j - 2 > \log \frac{m}{8k}$. □

Now we can state the following

Theorem 1. *For any IL-encoder with s states,*

$$\text{length}(y_1^n) \geq m^*(u_1^n) \log_2 \frac{m^*(u_1^n)}{8s^2}. \tag{2}$$

Proof. Let u_1^n be parsed into $m^* = m^*(u_1^n)$ distinct words, $u = v_1 \dots v_{m^*}$. Let s_i denote the state the machine is in just before reading v_i, and s_{i+1} the state the machine is in after reading v_i. Let t_i be the binary string the machine outputs while digesting v_i, so that $y_1^n = t_1 t_2 \dots t_{m*}$

Note that no binary string t can occur more than s^2 times among the t_i's. Indeed, suppose to the contrary the binary string t occurs more than s^2 times among the t_i's. Consider the v_i's for which $t_i = t$. Since there are only s^2 distinct values of $(s_i \to s_{i+1})$, we will find v_i and v_j for $t_i = t_j = t$, $s_i = s_j = z$ and $s_{i+1} = s_{j+1} = z'$. As $v_i \neq v_j$, this contradicts the machine being IL.

As the collection $t_1, \ldots, t_{m^*}$ satisfies the conditions of Lemma 3 with $k = s^2$, we get

$$\text{length}(y_1^n) = \sum_{i=1}^{m^*} \text{length}(t_i) \geq m^* \log \frac{m^*}{8s^2}. \qquad \square$$

Using Lemma 1 and (2), we see that

$$\begin{aligned}
\rho_s(u) &\geq \limsup_{n\to\infty} \frac{1}{n} m^*(u_1^n) \log_2(m^*(u_1^n)/(8s^2)) \\
&= \limsup_{n\to\infty} \frac{1}{n} m^*(u_1^n) \log_2 m^*(u_1^n) - \lim_{n\to\infty} \frac{1}{n} m^*(u_1^n) \log_2(8s^2) \\
&= \limsup_{n\to\infty} \frac{1}{n} m^*(u_1^n) \log_2 m^*(u_1^n)
\end{aligned}$$

and since the right hand side is independent of s,

$$\rho(u) \geq \limsup_{n\to\infty} \frac{1}{n} m^*(u_1^n) \log_2 m^*(u_1^n). \tag{3}$$

Now, let us describe the Lempel-Ziv algorithm. The algorithm proceeds by generating a dictionary for the source and constantly updating it. It starts up with a dictionary just consisting of the words of length 1, and operates in the following manner: When the dictionary has D words, each of its words is assigned a binary codeword of length $\lceil \log_2 D \rceil$ in lexicographic order. When a word in the dictionary is recognized in the input sequence, the encoder generates the binary codeword of that word on its output, and enlarges the dictionary by replacing the just recognized word with all its single letter extensions. The dictionary can be represented as a tree, whose leaves are the current dictionary entries. Figure 2 shows an example of the operation of the algorithm. Since the recognized words are encoded *before* the dictionary is modified, the decoder can keep track of the encoder's operation. Suppose that the algorithm parses the sequence u_1^n into $m(u_1^n)$ words $v_1, \ldots, v_m$. Then we can write:

$$u_1^n = \lambda v_1 v_2 \cdots v_m,$$

where λ denotes the null sequence. By construction, the first $m-1$ of the parses are distinct. (The last word v_m may not be distinct from the others.) If we concatenate the last two parses, and count in λ we get a parsing of u_1^n into $m(u_1^n)$ distinct words. Thus $m(u_1^n) \leq m^*(u_1^n)$. Since each parse extends the dictionary by $|\mathcal{U}| - 1$ entries, the size of the dictionary at the end of parsing u_1^n is

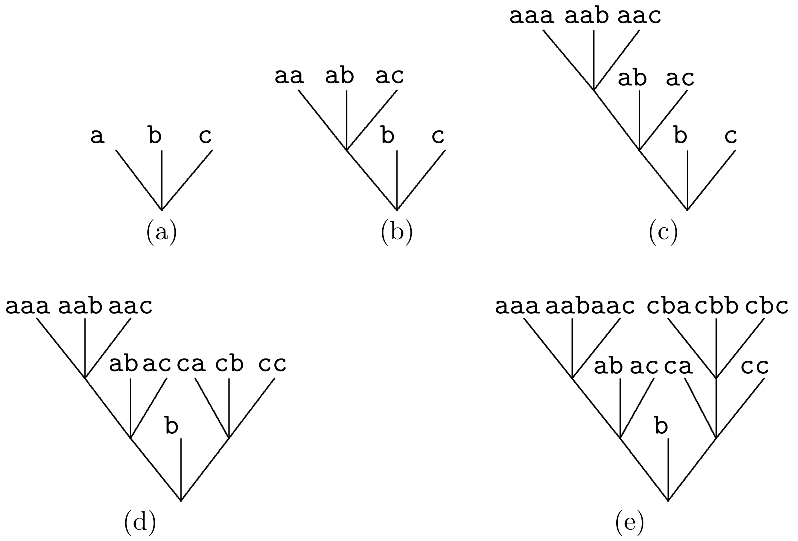
$$1 + (|\mathcal{U}| - 1)m(u_1^n) \leq 1 + (|\mathcal{U}| - 1)m^*(u_1^n) \leq |\mathcal{U}| m^*(u_1^n).$$

Thus, the number of bits $L(u_1^n)$ the LZ algorithm emits after seeing u_1^n is upper bounded by

$$m(u_1^n)\lceil \log_2(|\mathcal{U}| m^*(u_1^n)) \rceil \leq m(u_1^n) \log_2(2|\mathcal{U}| m^*(u_1^n)) \leq m^*(u_1^n) \log_2(2|\mathcal{U}| m^*(u_1^n)).$$

Dividing by n, and taking the $\limsup$ as n gets large we see

$$\begin{aligned}
\limsup_{n\to\infty} \frac{1}{n} L(u_1^n) &\leq \limsup_{n\to\infty} \frac{1}{n} m^*(u_1^n)[\log_2 m^*(u_1^n) + \log_2(2|\mathcal{U}|)] \\
&= \limsup_{n\to\infty} \frac{1}{n} m^*(u_1^n) \log_2 m^*(u_1^n)
\end{aligned}$$



The parsing of the sequence aaaccb with the Lempel-Ziv algorithm. The figure shows the evolution of the dictionary. The sequence is parsed into the phrases a, aa, c and cb. Figure 2(a) shows the initial dictionary. In 2(b) we see the dictionary after reading a, 2(c) shows after aaa has been read, etc. At each stage one might assign each dictionary entry a fixed length binary codeword. If the assignment is done in lexicographic order, at stage (a) it will be {a → 00, b → 01, c → 10}, at stage (b) {aa → 000, ab → 001, ..., c → 100}, at stage (c) {aaa → 000, aab → 001, ..., cc → 110}, and at stage (d) {aaa → 0000, aab → 0001, aac → 0010, ..., cc → 1000}, and the output sequence will be 00,000,110,0111. (Commas are put in to aid the reader, they will not appear at the output.)

Figure 2: Operation of the Lempel-Ziv algorithm

so that the LZ algorithm will achieve the lower bound previously derived (3) in the limit of $n \to \infty$. (However, the algorithm uses up infinite memory, since it keeps track of an ever growing tree.)

One can perhaps express the tradeoff we have seen as follows: suppose we want to compress an infinite string u, and we were given the choice of using the 'off the shelf" Lempel-Ziv, versus designing a machine tuned to u with a finite (but arbitrary) number of states. Then, we might as well pick the Lempel-Ziv: In the long run (i.e., for long strings) the Lempel-Ziv algorithm will do as well as the best finite state machine.

In particular, if one knew that the string u is the output of an information source which is stationary and ergodic, one could have designed a finite state machine that implements, for example, the Huffman algorithm designed for this source for a large enough block length that will compress the source output with high probability, arbitrary close to its entropy rate. Combined with the above paragraph we see that for such sources, the Lempel Ziv algorithm will compress them to their entropy rate too.