Lempel-Ziv Notes
18.310C, Spring 2010, Prof. Peter Shor

We did Huffman coding last time. Huffman coding works fairly well, in that it comes within one bit per letter (or block of letters) of the bound that Shannon gives for encoding sequences of letters with a given set of frequencies. There are some disadvantages to it. For one thing, it requires two passes through the data you wish to encode. The first pass is used for computing the frequencies of all the letters, and the second pass for actually encoding the data. If you don't want to look at the data twice; for instance, if you're getting the data to be encoded from some kind of program, and you don't have the memory to store it without encoding it first, this can be a problem. The Lempel Ziv algorithm constructs its dictionary on the fly, only going through the data once.

There are many variations of Lempel Ziv around, but they all follow the same basic idea. We now explain the algorithm that Lempel and Ziv gave in a 1978 paper, generally called LZ78. This is as opposed to LZ77, an earlier algorithm which is based on the same general idea, but which differs significantly in the implementation details. The idea is that if some text is not random, a substring that you see once is more likely to appear again than substrings you haven't seen. This is certainly true for any natural language, where words get used repeatedly, whereas strings of letters which don't appear in words will hardly ever get used.

The LZ78 algorithm works by constructing a dictionary of substrings, which we will call "phrases," that have appeared in the text. The LZ78 algorithm constructs its dictionary on the fly, only going through the data once. This is a great advantage in that you don't have to receive the entire document before starting to encode it. The idea is to parse the sequence into distinct phrases. The version we analyze does this greedily. Suppose, for example, we have the string

$$AABABBBABAABABBBABBABB$$

We start with the shortest phrase on the left that we haven't seen before. This will always be a single letter, in this case A:

$$A|ABABBBABAABABBBABBABB$$

We now take the next phrase we haven't seen. We've already seen A, so we take AB:

$$A|AB|ABBBABAABABBBABBABB$$

The next phrase we haven't seen is ABB, as we've already seen AB. Continuing, we get B after that:

$$A|AB|ABB|B|ABAABABBBABBABB$$

and you can check that the rest of the string parses into

$$A|AB|ABB|B|ABA|ABAB|BB|ABBA|BB$$

Because we've run out of letters, the last phrase on the end is a repeated one. That's O.K.

Now, how do we encode this? For each phrase we see, we stick it in the dictionary. The next time we want to send it, we don't send the entire phrase, but just the number of this phrase. Consider the following table

1	2	3	4	5	6	7	8	9
A	AB	ABB	B	ABA	$ABAB$	BB	$ABBA$	BB
A	$1\,B$	$2\,B$	$0\,B$	$2\,A$	$5\,B$	$4\,B$	$3\,A$	7

The first row gives the numbers of the phrases (which you should think of as being in binary), the second row gives the phrases, and the third row their encodings. That is, when we're encoding the ABAB from the sixth phrase, we encode it as 5B (We will later encode both 5 and B in binary to complete the encoding.) This maps to ABAB since the fifth phrase was ABA, and we add B to it. Here, the empty set $\emptyset$ should be considered as the 0'th phrase and encoded by 0. The last piece is encoding this string into binary This gives

$$011101001010010111001011001111$$

To see how it works, let's insert dividers and commas, to make it more comprehensible.

$$,0|1,1|10,1|00,1|010,0|101,1|100,1|011,0|0111$$

We have taken the third row of the previous array, expressed all the numbers in binary (before the comma) and the letters in binary (after the comma) Note that I've mapped A to 0 and B to 1. If you had a larger alphabet, you would encode the letters by more than one bit. Note also that we've mapped 2 (referencing the second phrase) to both 10 and 010. As soon as a reference to a phrase might conceivably involve k bits (starting with the $2^{k-1}+1$ dictionary element), we've used k bits to encode the phrases, so the number of bits used before the comma keeps increasing. This ensures

that the decoding algorithm knows where to put the commas and dividers; otherwise the receiver wouldn't know whether a '10' stood for the second phrase or was the first two bits of '100', standing for the fourth phrase.

You might notice that in this case, the compression algorithm actually made the sequence longer. This is the case for one of two reasons. Either this original sequence was too random to be compressed much, or it was too short for the asymptotic efficiency of Lempel-Ziv to start being noticeable.

To decode, the decoder needs to construct the same dictionary. To do this, he first takes the binary string he receives, and inserts dividers and commas. This is straightforward. The first divider comes after one bit, the next comes after 2 bits. The next two each come after 3 bits. We then get 2^2 of length 4 bits, 2^3 of length 5 bits, 2^4 of length 6 bits, and in general 2^k of length $k+2$ bits. The phrases just give the dictionary. For example, our dictionary for the above string is

$\emptyset$	0
A	1
AB	2
ABB	3
B	4
ABA	5
ABAB	6
BB	7
ABBA	8

The empty set $\emptyset$ is always encoded by 0 in the dictionary.

Recall that when we encoded our phrases, if we had r phrases in our dictionary, we used $\lceil \log_2 r \rceil$ bits to encode the number of the phrase. (Recall $\lceil x \rceil$ is the smallest integer greater than x.) This ensures that the decoder knows exactly how many bits are in each phrase. You can see that in the example above, the first time we encoded AB (phrase 2) we encoded it as 10, and the second time we encoded it as 010. This is because the first time, we had three phrases in our dictionary, and the second time we had five.

The decoder uses the same algorithm to construct the dictionary that the encoder did; this ensures that the decoder and the encoder construct the same dictionary. The decoder knows phrases 0 through $r-1$ when he is trying to figure out what the rth phrase is, and this is exactly the information he needs to reconstruct the dictionary.

How well have we encoded the string? Suppose we have broken it up into $c(n)$ phrases, where n is the length of the string. Each phrase is broken

up into a reference to a previous phrase and a letter of our alphabet. The previous phrase is always represented by at most $\lceil \log_2 c(n) \rceil$ bits, since there are $c(n)$ phrases, and each letter can be represented by at most $\lceil \log_2 \alpha \rceil$ bits, where α is the size of the alphabet (in the above example, it is 2). We have thus used at most

$$c(n)(\log_2 c(n) + \log_2 \alpha + 2)$$

bits total in our encoding.

(In practice, you don't want to use too much memory for your dictionary. Thus, most implementation of Lempel-Ziv type algorithms have some maximum size for the dictionary. When it gets full, they will drop a little-used word from the dictionary and replace it by the current word. This also helps the algorithm adapt to encode messages with changing characteristics. You merely need to use some deterministic algorithm for choosing which word to drop, so that both the sender and the receiver will drop the same word.)

So how well does the Lempel-Ziv algorithm work? In these notes, we'll calculate two quantities. First, how well it works in the worst case, and second, how well it works in the random case where each letter of the message is chosen uniformly and independently from a probability distribution, where the ith letter appears with probability p_i. We'll skip the worst case calculation in class, because it isn't as important, and there's not enough time.

In both cases, the compression is asymptotically optimal. That is, in the worst case, the length of the encoded string of bits is $n + o(n)$. Since there is no way to compress all length-n strings to fewer than n bits, this can be counted as asymptotically optimal. In the second case, the source is compressed to length

$$H(p_1, p_2, \ldots, p_\alpha)n + n \log \alpha + o(n) = n \sum_{i=1}^{\alpha} (-p_i \log_2 p_i) + n \log \alpha + o(n),$$

where $o(n)$ means a term that grows more slowly than n asymptotically. This is, to leading order, the Shannon bound. The Lempel-Ziv algorithm actually works asymptotically optimally for more general cases, including cases where the letters are produced by certain classes of probabilistic processes.

1 Worst-Case Analysis

Let's do the worst case analysis first. Suppose we are compressing a binary alphabet. We ask the question: what is the maximum number of distinct

phrases that a string of length n can be parsed into. There are some strings which are clearly worst case strings. These are the ones in which the phrases are all possible strings of length at most k. For example, for $k = 1$, one of these strings is

$$0|1$$

with length 2. For $k = 2$, one of them is

$$0|1|00|01|10|11$$

with length 10; and for $k = 3$, one of them is

$$0|1|00|01|10|11|000|001|010|011|100|101|110|111$$

with length 34. In general, the length of such a string is

$$n_k = \sum_{j=1}^{k} j2^j$$

since it contains 2^j phrases of length j. It is easy to check that

$$n_k = (k-1)2^{k+1} + 2$$

by induction. [This could be an exercise. We saw the same sum appear in our analysis of heapsort.] If we let $c(n_k)$ be the number of distinct phrases in this string of length n_k, we get that

$$c(n_k) = \sum_{i=1}^{k} 2^i = 2^{k+1} - 2$$

For n_k, we thus have

$$c(n_k) = 2^{k+1} - 2 \leq \frac{(k-1)2^{k+1}}{k-1} \leq \frac{n_k}{k-1}$$

2 Average-Case Analysis

Now, for an arbitrary length n, we can write $n = n_k + \Delta$. To get the case where $c(n)$ is largest, the first n_k bits can be parsed into $c(n_k)$ distinct phrases, containing all phrases of length at most k, and the remaining Δ bits can be parsed into into phrases of length $k+1$. This is clearly the most

distinct phrases a string of length n can be parsed into, so we have that for a general string of length n, the number of phrases is at most total is

$$c(n) \leq \frac{n_k}{k-1} + \frac{\Delta}{k+1} \leq \frac{n_k + \Delta}{k-1} = \frac{n}{k-1} \leq \frac{n}{\log_2 c(n) - 3}$$

Now, we have that a general bit string is compressed to around $c(n) \log_2 c(n) + c(n)$ bits, and if we subsitute

$$c(n) \leq \frac{n}{\log_2 c(n) - 3}$$

we get

$$c(n) \log_2 c(n) + c(n) \leq n + 4c(n) = n + O(\frac{n}{\log_2 n})$$

So asymptotically, we don't use much more than n bits for compressing any string of length n. This is good: it means that the Lempel-Ziv algorithm doesn't expand any string very much. We can't hope for anything more from a general compression algorithm, as it is impossible to compress all strings of length n into fewer than n bits. If a lossless compression algorithm compresses some strings to fewer than n bits, it will have to expand other strings to more than n bits. [Lossless here means the uncompressed string is exactly the original message.]

We now need to show that in the case of random strings, the Lempel Ziv algorithm's compression rate asymptotically approaches the entropy. Let us imagine that we construct our sequence by, at each position, independently choosing letter α_i with probability p_i. This gives us a sequence

$$x = \alpha_{x(1)} \alpha_{x(2)} \alpha_{x(3)} \cdots \alpha_{x(n)}$$

We define $P(x)$ to be the probability of seeing this sequence. That is,

$$P(x) = \prod_{i=1}^{n} p_{x(i)}.$$

Now, note that $-\log P(x)$ is the entropy of the sting x. That is, if the ith letter α_{x_i} occurs np_i times, then

$$P(x) = \prod_i p_i^{p_i n},$$

and

$$-\log P(x) = -n \sum_i p_i \log p_i$$

In the proof of Shannon's noiseless coding theorem, we used the weak law of large numbers to say that with high probability, the entropy of a string emitted from a source is very close to nH, where n is the entropy of a source.

Now, suppose the string x is broken into distinct phrases

$$x = y_1 y_2 y_3 \dots y_{c(x)},$$

where $c(x)$ is the number of distinct phrases that x parses into. It is not hard to see that

$$P(x) = \prod_{i=1}^{c(x)} P(y_i) \tag{1}$$

Now, let's let c_l be the number of phrases y_i of length l. These are (because of the way Lempel-Ziv works) all distinct. We can now prove a version of what is known as Ziv's inequality. This inequality says

$$-\log_2 P(x) \geq \sum_l c_l \log_2 c_l$$

We start by rewriting Eq. 1 above

$$P(x) = \prod_l \prod_{|y_i|=l} P(y_i)$$

Now, let's look at the inner product. We know that, since the y_i with length l are all distinct, they are mutually independent events, so the probabilities $P(y_i)$ sum to at most 1.

$$\sum_{|y_i|} P(y_i) \leq 1$$

Now, there is an inequality that says that to maximize a product of k terms, given a fixed sum of these terms, the best thing to do is make all of these terms equal. (You can easily prove this using Lagrange multipliers, which you learned in 18.02.) There are c_l terms $P(y_i)$ with $|y_i| = l$. Setting them all equal, we see that

$$\prod_{|y_i|=l} P(y_i) \leq \left(\frac{1}{c_l}\right)^{c(l)}$$

Taking logs, and adding over all phrase lengths l, we get what is known as Ziv's inequality

$$-\log_2 P(x) \geq \sum_l c_l \log_2 c_l$$

Recall that the length of our compressed string from the Lempel-Ziv algorithm was approximately $c(x) \log_2 c(x)$. Thus, all we need to do is show

$$\sum_l c_l \log_2 c_l \approx c(x) \log_2 c(x).$$

We're going to do some algebraic manipulations which lead us to this answer. First, let me say that the intuitive meaning of these manipulations is that the maximum entropy comes from maximizing the entropies on the phrases of length l, and then maximizing the entropies on the distributions of these phrase lengths.

We have $\sum_l c_l = c(x)$. This gives us the equation

$$\begin{aligned} \sum c_l \log_2 c_l &= \sum c_l (\log_2 \frac{c_l}{c(x)} + \log_2 c(x)) \\ &= c(x) \log_2 c(x) + c(x) \sum_l \frac{c_l}{c(x)} \log_2 \frac{c_l}{c(x)} \end{aligned}$$

The first term on the last line is what we want. The last term, which we'd like to show is small, is $-c(x)$ times

$$-\sum_l \frac{c_l}{c(x)} \log_2 \frac{c_l}{c(x)},$$

which looks like an entropy. In fact, it is an entropy: we have a constraint that saying that $\frac{c_l}{c(x)}$ are probabilities:

$$\sum_l \frac{c_l}{c(x)} = 1$$

and a constraint on the expected value of l under this probability distribution:

$$\sum_l \frac{c_l}{c(x)} = \frac{n}{c(x)}$$

Now,

$$-\sum_l \frac{c_l}{c(x)} \log_2 \frac{c_l}{c(x)},$$

is the entropy of the probability distribution $\pi_l = \frac{c_l}{c(x)}$. The l are positive integers, and the expected value of l is

$$\sum_l l \frac{c_l}{c(x)} = \frac{n}{c(x)}$$

Now, the maximum possible entropy for a probability distribution π_l on positive integers whose expected value is $\frac{n}{c(x)}$ is $O(\log \frac{n}{c(x)})$ [Exercise: the distribution maximizing entropy can be computed exactly using Lagrange multipliers; do it.] It isn't hard to see why this should be true intuitively. If the expected value is $n/c(x)$, then most of the weight must be in the first $O(n/c(x))$ integers, and if a distribution is spread out over a sample space of size $O(n/c(x))$, the entropy is at most $O(\log(n/c(x)))$.

So we get that

$$-\log_2 P(x) \geq c(x) \log_2 c(x) - c(x) O(\log_2 \frac{n}{c(x)})$$

and since $n/c(x) = O(\log x)$, this shows that

$$c(x) \log_2 c(x) \leq -\log_2 Q(x) + c(x) O(\log \log n)).$$

Recall that $\log_2 Q(x)$ is approximately $nH(p)$, and $c(x) \log_2 c(x)$ is approximately the length of the compressed sequence. Thus, this shows that the sequence is compressed to length not much more than n times the entropy, and so is asymptotically optimal.

The Lempel-Ziv algorithm also works for any messages that are generated by probabilistic processes with limited memory. This means that the probability of seeing a given letter may depend on the previous letters, but it can only depend on letters that are close to it. The spirit of the proof is the same, although the details get more complicated. This kind of process seems to reflect real-world sequences pretty well, and the Lempel-Ziv family of algorithms works very well on a lot of real-world sequences.