Lempel-Ziv codes

Lecturer: Michel Goemans

We have described Huffman coding in the previous lecture note. 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.

1 The LZ78 Algorithm

There are many variations of Lempel Ziv around. 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 is quite different in the implementation details. The idea behind all the Lempel-Ziv algorithms is that if some text is not uniformly random (i.e., all the letters of the alphabet don't have the same probability), a substring that have already seen is more likely to appear again than substrings you haven't seen. This is certainly true for any natural language, where words will 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 means that you don't have to receive the entire document before starting to encode it. The algorithm parses the sequence into distinct phrases. We do 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 $\emptyset$ is considered to be the 0'th phrase and encoded by 0. The last step of the algorithm is to encode this string into binary This gives

$$011101001010010111100101100111$$

To see how it works, let's insert dividers and commas (which aren't present in the real output of LZ78) 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 (that is, 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 couldn't tell 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 could be 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 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 phrase $\emptyset$ is always encoded by 0 in the dictionary.

Recall that when we encoded our phrases, if we had r phrases in our dictionary (including the empty phrase $\emptyset$), 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? For an input string x, let $c(x)$ denote the number of phrases that x gets split into. 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 $\lceil \log_2 c(x) \rceil$ bits, since there are $c(x)$ phrases, and each letter can be represented by at most $\lceil \log_2 N \rceil$ bits, where N is the size of the alphabet (in the above example, it is 2). We have thus used at most

$$c(x)(\log_2 c(x) + \log_2 N + 2) \tag{1}$$

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 phrase from the dictionary and replace it by the current phrase. This also helps the algorithm adapt to encode messages with changing characteristics. You only 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 independently from a probability distribution with the ith letter of the alphabet having 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 binary strings to fewer than n bits (this could be an exercise), the worst-case behavior is asymptotically

optimal. In the second case, the source is compressed to length

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

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 many classes of probabilistic processes where the distribution of a letter depends on the letters immediately before it.

2 Worst-Case Analysis

(This section is not examinable.)

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

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

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

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

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 encoding 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.]

3 Average-Case Analysis

We now need to show that in the case of random strings, the Lempel Ziv algorithm's compression rate asymptotically approaches the entropy. Let A be the alphabet and p_a be the probability of obtaining letter $a \in A$. As before, we assume that we have a "first order source": that is we have a random sequence of letters

$$x = X_1 X_2 \ldots X_n$$

where the X_i are independent letters from A with $\mathbb{P}(X_j = a_i) = p_i$ for all i, j. Further let us assume that $p_i \leq \frac{1}{2}$ for all i, i.e., that no letter has a large probability. These assumptions are not necessary to show that Lempel-Ziv performs well, but they do make the proof quite a bit easier.

For any particular sequence of letters

$$x = x_1 x_2 \ldots x_n.$$

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

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

The plan for what follows is:

1. Bound $P(x)$ in terms of $c(x)$; we want to show that messages that require many phrases (and hence are long upon encoding by Lempel-Ziv) occur with very low probability.
2. Relate $P(x)$ to the entropy.

Bounding $P(x)$ in terms of $c(x)$

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

$$P(x) = \prod_{i=1}^{c(x)} P(y_i), \tag{2}$$

where $P(y_i)$ is simply the probability that phrase y_i occurs in the correct position of the input string; this is just the product of the letter probabilities in y_i.

Let us define C_j, for j a nonnegative integer, as the set of phrases

$$\{y_i : 2^{-j-1} < P(y_i) \leq 2^{-j}\},$$

that is, the set of phrases in $\{y_1, \dots, y_{c(x)}\}$ with probabilities between 2^{-j-1} and 2^{-j}.

Claim 1. *For any j, $\sum_{y_i \in C_j} P(y_i) \leq 1$.*

Proof. We first observe that if y_i and $y_{i'}$ are both in C_j, then y_i is not a prefix of $y_{i'}$. For if that were the case, then the probability of obtaining $y_{i'}$ would be $P(y_i)$ multiplied by the products of the probabilities of obtaining the correct extra letters in $y_{i'}$. but even one extra letter has probability at most $1/2$ of being correct (by our assumption on the p's). So $P(y_{i'}) \leq P(y_i)/2$, contradicting that both are in C_j.

Obviously $P(y_i)$, the probability of obtaining the phrase y_i at the correct position in the input, is the same as the probability of obtaining phrase y_i at the very beginning of the input. So define E_i as the event that a random message $s = X_1 X_2 \dots X_n$ starts with the phrase y_i. By the above, $\{E_i : y_i \in C_j\}$ is a collection of disjoint events; if E_i holds, then $E_{i'}$ cannot hold for any $i' \neq i$, since y_i is not a prefix of $y_{i'}$, or vice versa. Hence $\sum_{y_i \in C)j} \mathbb{P}(E_i) \leq 1$. But $\mathbb{P}(E_i) = P(y_i)$, so we're done. □

Since each phrase in C_j has probability at least 2^{-j-1}, and these probabilities sum to at most 1, there can be no more than 2^{j+1} phrases in C_j. Using in addition that $P(y_i) \leq 2^{-j}$ for all $y_i \in C_j$, we obtain

$$P(x) \leq \prod_{j=0}^{\infty} (2^{-j})^{|C_j|};$$

taking logs, we obtain

$$-\log_2 P(x) \geq \sum_{j=0}^{\infty} j|C_j|.$$

Note that since our goal was to upper bound $P(x)$ as a function of $c(x)$, we now want to lower bound $-\log_2 P(x)$ in terms of $c(x)$. As such, let's minimize the right hand side of the above equation, subject to the constraints that $|C_j| \leq 2^{j+1}$ and $\sum_{j=0}^{\infty} |C_j| = c(x)$.

We can describe the problem in the following way: we have a bucket for each nonnegative integer, and it costs j dollars to put a ball into bucket j. Also, bucket j can fit only 2^{j+1} balls. We

have $c(x)$ balls; what is the cheapest way of disposing of them in the buckets? It should be clear that we should full up the buckets starting from the cheapest; so fill up bucket 0, 1, 2 etc. in turn, until we've disposed of all the balls. This gives a total cost of

$$\sum_{j=0}^{k-1} j2^{j+1} + k\left(c(x) - \sum_{j=0}^{k-1} 2^{j+1}\right),$$

where k is the index of the last bucket that gets used, i.e., the smallest integer such that $2 + 4 + \cdots + 2^{k+1} \geq c(x)$. You can check that this yields $k = \lceil \log_2(c(x) - 2) \rceil = \log_2 c(x) + O(1)$.

Simplifying this, we obtain a lower bound of

$$\begin{aligned}
-\log_2 P(x) &\geq \sum_{j=0}^{k-1} j2^{j+1} + k(c(x) - 2^{k+1} + 2) \\
&= (k-2)2^{k+1} + 4 + k(c(x) - 2^{k+1} + 1) \\
&= (\log c(x) + O(1))2^{k+1} + (\log c(x) + O(1))(c(x) - 2^{k+1} + 2) \\
&= (\log c(x) + O(1))(c(x) + 1) \\
&= c(x) \log c(x) + O(c(x)).
\end{aligned}$$

This is precisely what we obtained already for the encoding length of x! See Equation 1. So

$$-\log_2 P(x) \geq L_\phi(x) + o(n),$$

where $L_\phi(x)$ is the encoding length of x using Lempel-Ziv. Here, we also used that $c(x) = o(n)$, which was proved in the previous section on the worst-case bound.

Relating $P(x)$ to the entropy

We can rewrite $P(x)$ as

$$P(x) = \prod_{a \in A} p_a^{n_a},$$

where n_a is the number of occurences of the letter a in x. Recall that when we proved Shannon's noiseless coding theorem, we actually showed that in a random message from the source, it's very likely that each n_a is close to its expected value, which is np_a. More precisely, we said that a message x is ϵ-typical if $|n_a - np_a| \leq \epsilon n$ for each $a \in A$, and we showed that

$$\mathbb{P}(s \text{ is } \epsilon\text{-typical}) \geq \frac{1}{\epsilon^2 n}. \tag{3}$$

So suppose that x is ϵ-typical. Then

$$\begin{aligned}
-\log_2 P(x) &= -\sum_{a \in A} n_a \log_2 p_a \\
&= -n \sum_{a \in A} p_a \log_2 p_a + nO(\epsilon \log_2 \epsilon) \\
&= nH(p) + nO(\epsilon \log_2 \epsilon).
\end{aligned}$$

I skipped the details regarding the ϵ term here, but you can find the calculations in the notes on Shannon's theorem.

Now take $\epsilon = 1/\log_2 n$ for what follows, so that both $\epsilon \to 0$ and $\mathbb{P}(s \text{ is } \epsilon\text{-typical}) \to 0$ by (3). So then if x is ϵ-typical, $-\log_2 P(x) = nH(p) + o(n)$, and hence

$$L_\phi(x) = -\log_2 P(x) + o(n) = nH(p) + o(n).$$

Since a random message from the source is very likely to be ϵ-typical, we will just use a weak bound for the encoding length of a message that is not ϵ-typical. Using the result of the previous section, we have $L_\phi(x) \leq n + o(n)$ for any message x. Thus

$$\begin{aligned}
\mathbb{E}(L_\phi) &= \mathbb{E}(L_\phi | s \text{ is } \epsilon\text{-typical})\mathbb{P}(s \text{ is } \epsilon\text{-typical}) + \mathbb{E}(L_\phi | s \text{ is not } \epsilon\text{-typical})\mathbb{P}(s \text{ is not } \epsilon\text{-typical}) \\
&\leq (nH(p) + o(n)) \cdot 1 + (n + o(n)) \cdot \frac{1}{\epsilon^2 n} \\
&= nH(p) + o(n).
\end{aligned}$$

If the highest probability letter has probability larger than $\frac{1}{2}$, a more complicated version of essentially the same argument works to show that Lempel-Ziv gives optimal compression in this case as well.

The Lempel-Ziv algorithm will also works for any source that is generated by a probabilistic process 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. Since we haven't introduced enough probability theory in this class to model these sources, we will not go over the proof. However, it is worth mentioning that the details are quite a bit more complicated than the proof given in these notes. This kind of process does seem to reflect real-world sequences pretty well; the Lempel-Ziv family of algorithms works very well on a lot of real-world sequences.

18.310 Principles of Discrete Applied Mathematics
Fall 2013