Notes 11 for CS 170

1 The Lempel-Ziv algorithm

There is a sense in which the Huffman coding was "optimal", but this is under several assumptions:

1. The compression is lossless, i.e., uncompressing the compressed file yields exactly the original file. When lossy compression is permitted, as for video, other algorithms can achieve much greater compression, and this is a very active area of research because people want to be able to send video and audio over the Web.

2. We know all the frequencies $f(i)$ with which each character appears. How do we get this information? We could make two passes over the data, the first to compute the $f(i)$, and the second to encode the file. But this can be much more expensive than passing over the data once for large files residing on disk or tape. One way to do just one pass over the data is to assume that the fractions $f(i)/n$ of each character in the file are similar to files you've compressed before. For example you could assume all Java programs (or English text, or PowerPoint files, or ...) have about the same fractions of characters appearing. A second cleverer way is to estimate the fractions $f(i)/n$ on the fly as you process the file. One can make Huffman coding adaptive this way.

3. We know the set of characters (the alphabet) appearing in the file. This may seem obvious, but there is a lot of freedom of choice. For example, the alphabet could be the characters on a keyboard, or they could be the key words and variables names appearing in a program. To see what difference this can make, suppose we have a file consisting of n strings $aaaa$ and n strings $bbbb$ concatenated in some order. If we choose the alphabet $\{a, b\}$ then $8n$ bits are needed to encode the file. But if we choose the alphabet $\{aaaa, bbbb\}$ then only $2n$ bits are needed.

Picking the correct alphabet turns out to be crucial in practical compression algorithms. Both the UNIX compress and GNU gzip algorithms use a greedy algorithm due to Lempel and Ziv to compute a good alphabet in one pass while compressing. Here is how it works.

If s and t are two bit strings, we will use the notation $s \circ t$ to mean the bit string gotten by concatenating s and t.

We let f be the file we want to compress, and think of it just as a string of bits, that is 0's and 1's. We will build an alphabet A of common bit strings encountered in f, and use it to compress f. Given A, we will break f into shorter bit strings like

$$f = A(1) \circ 0 \circ A(2) \circ 1 \circ \cdots \circ A(7) \circ 0 \circ \cdots \circ A(5) \circ 1 \circ \cdots \circ A(i) \circ j \circ \cdots$$

and encode this by

$$1 \circ 0 \circ 2 \circ 1 \circ \cdots \circ 7 \circ 0 \circ \cdots \circ 5 \circ 1 \circ \cdots \circ i \circ j \circ \cdots$$

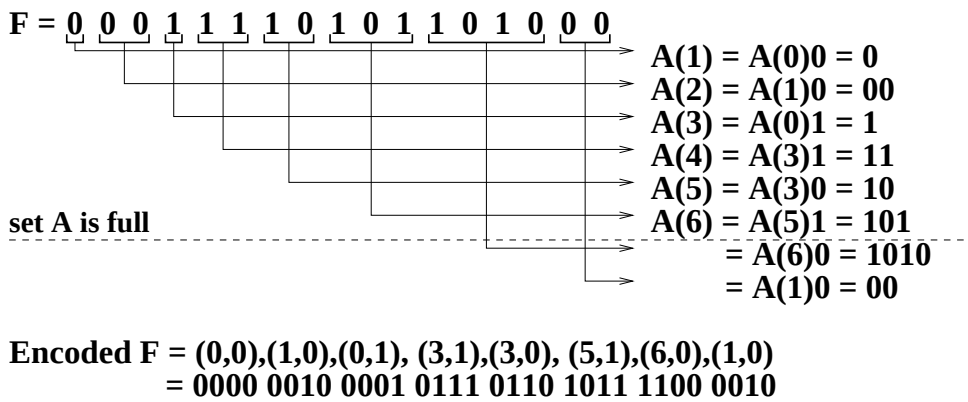

Figure 1: An example of the Lempel-Ziv algorithm.

The indices i of $A(i)$ are in turn encoded as fixed length binary integers, and the bits j are just bits. Given the fixed length (say r) of the binary integers, we decode by taking every group of $r+1$ bits of a compressed file, using the first r bits to look up a string in A, and concatenating the last bit. So when storing (or sending) an encoded file, a header containing A is also stored (or sent).

Notice that while Huffman's algorithm encodes blocks of fixed size into binary sequences of variable length, Lempel-Ziv encodes blocks of varying length into blocks of fixed size.

Here is the algorithm for encoding, including building A. Typically a fixed size is available for A, and once it fills up, the algorithm stops looking for new characters.

$A = \{\emptyset\}$ ... start with an alphabet containing only an empty string
$i = 0$ ... points to next place in file f to start encoding
repeat
  find $A(k)$ in the current alphabet that matches as many leading bits
    of $f_i f_{i+1} f_{i+2} \cdots$ as possible
      ... initially only $A(0)$ = empty string matches
  let b be the number of bits in $A(k)$
  if A is not full, add $A(k) \circ f_{i+b}$ to A
      ... f_{i+b} is the first bit unmatched by $A(k)$
  output $k \circ f_{i+b}$
  $i = i + b + 1$
until $i >$ length(f)

Note that A is built "greedily", based on the beginning of the file. Thus there are no optimality guarantees for this algorithm. It can perform badly if the nature of the file changes substantially after A is filled up, however the algorithm makes only one pass through the file (there are other possible implementations: A may be unbounded, and the index k would be encoded with a variable-length code itself).

In Figure 1 there is an example of the algorithm running, where the alphabet A fills up after 6 characters are inserted. In this small example no compression is obtained, but if A were large, and the same long bit strings appeared frequently, compression would

be substantial. The gzip manpage claims that source code and English text is typically compressed 60%-70%.

To observe an example, we took a latex file of $74,892$ bytes. Running Huffman's algorithm, with bytes used as blocks, we could have compressed the file to $36,757$ bytes, plus the space needed to specify the code. The Unix program `compress` produced an encoding of size $34,385$, while `gzip` produced an encoding of size $22,815$.

2 Lower bounds on data compression

2.1 Simple Results

How much can we compress a file without loss? We present some results that give lower bounds for *any* compression algorithm. Let us start from a "worst case" analysis.

THEOREM 1
Let $C : \{0,1\}^n \to \{0,1\}^*$ be an encoding algorithm that allows lossless decoding (i.e., let C be an injective function mapping n bits into a sequence of bits). Then there is a file $f \in \{0,1\}^n$ such that $|C(f)| \geq n$.

In words, for any lossless compression algorithm there is always a file that the algorithm is unable to compress.

PROOF: Suppose, by contradiction, that there is a compression algorithm C such that, for all $f \in \{0,1\}^n$, $|C(f)| \leq n-1$. Then the set $\{C(f) : f \in \{0,1\}^n\}$ has 2^n elements because C is injective, but it is also a set of strings of length $\leq n-1$, and so it has at most $\sum_{l=1}^{n-1} 2^l = 2^n - 2$ elements, which gives a contradiction. $\square$

While the previous analysis showed the existence of incompressible files, the next theorem shows that random files are hard to compress, thus giving an "average case" analysis.

THEOREM 2
Let $C : \{0,1\}^n \to \{0,1\}^*$ be an encoding algorithm that allows lossless decoding (i.e., let C be an injective function mapping n bits into a sequence of bits). Let $f \in \{0,1\}^n$ be a sequence of n randomly and uniformly selected bits. Then, for every t,

$$\Pr[|C(f)| \leq n-t] \leq \frac{1}{2^{t-1}}$$

For example, there is less than a chance in a million of compression an input file of n bits into an output file of length $n-21$, and less than a chance in eight millions that the output will be 3 bytes shorter than the input or less.

PROOF: We can write

$$\Pr[|C(f)| \leq n-t] = \frac{|\{f : |C(f)| \leq n-t\}|}{2^n}$$

Regarding the numerator, it is the size of a set that contains only strings of length $n-t$ or less, so it is no more than $\sum_{l=1}^{n-t} 2^l$, which is at most $2^{n-t+1} - 2 < 2^{n-t+1} = 2^n/2^{t-1}$. $\square$

The following result is harder to prove, and we will just state it.

THEOREM 3
Let $C : \{0,1\}^n \to \{0,1\}^*$ be a prefix-free encoding, and let f be a random file of n bits. Then $\mathbf{E}[|C(f)|] \geq n$.

This means that, from the average point of view, the optimum prefix-free encoding of a random file is just to leave the file as it is.

In practice, however, files are not completely random. Once we formalize the notion of a not-completely-random file, we can show that some compression is possible, but not below a certain limit.

First, we observe that even if not all n-bits strings are possible files, we still have lower bounds.

THEOREM 4
Let $F \subseteq \{0,1\}^n$ be a set of possible files, and let $C : \{0,1\}^n \to \{0,1\}^*$ be an injective function. Then

1. *There is a file $f \in F$ such that $|C(f)| \geq \log_2 |F|$.*
2. *If we pick a file f uniformly at random from F, then for every t we have*
$$\Pr[|C(f)| \leq (\log_2 |F|) - t] \leq \frac{1}{2^{t-1}}$$
3. *If C is prefix-free, then when we pick a file f uniformly at random from F we have $\mathbf{E}[|C(f)|] \geq \log_2 |F|$.*

PROOF: Part 1 and 2 is proved with the same ideas as in Theorem 1 and Theorem 2. Part 3 has a more complicated proof that we omit. □

2.2 Introduction to Entropy

Suppose now that we are in the following setting:

the file contains n characters
there are c different characters possible
character i has probability $p(i)$ of appearing in the file

What can we say about probable and expected length of the output of an encoding algorithm?

Let us first do a very rough approximate calculation. When we pick a file according to the above distribution, very likely there will be about $p(i) \cdot n$ characters equal to i. The files with these "typical" frequencies have a total probability about $p = \prod_i p(i)^{p(i)\cdot n}$ of being generated. Since files with typical frequencies make up almost all the probability mass, there must be about $1/p = \prod_i (1/p(i))^{p(i)\cdot n}$ files of typical frequencies. Now, we are in a setting which is similar to the one of parts 2 and 3 of Theorem 4, where F is the set of files with typical frequencies. We then expect the encoding to be of length at least $\log_2 \prod_i (1/p(i))^{p(i)\cdot n} = n \cdot \sum_i p(i) \log_2(1/p(i))$. The quantity $\sum_i p(i) \log_2 1/(p(i))$ is the expected number of bits that it takes to encode each character, and is called the *entropy*

of the distribution over the characters. The notion of entropy, the discovery of several of its properties, (a formal version of) the calculation above, as well as a (inefficient) optimal compression algorithm, and much, much more, are due to Shannon, and appeared in the late 40s in one of the most influential research papers ever written.

2.3 A Calculation

Making the above calculation precise would be long, and involve a lot of ϵs. Instead, we will formalize a slightly different setting. Consider the set F of files such that

the file contains n characters
there are c different characters possible
character i occurs $n \cdot p(i)$ times in the file

We will show that F contains roughly $2^{n \sum_i p(i) \log_2 1/p(i)}$ files, and so a random element of F cannot be compressed to less than $n \sum_i p(i) \log_2 1/p(i)$ bits. Picking a random element of F is almost but not quite the setting that we described before, but it is close enough, and interesting in its own. Let us call $f(i) = n \cdot p(i)$ the number of occurrences of character i in the file.

We need two results, both from Math 55. The first gives a formula for $|F|$:

$$|F| = \frac{n!}{f(1)! \cdot f(2)! \cdots f(c)!}$$

Here is a sketch of the proof of this formula. There are $n!$ permutations of n characters, but many are the same because there are only c different characters. In particular, the $f(1)$ appearances of character 1 are the same, so all $f(1)!$ orderings of these locations are identical. Thus we need to divide $n!$ by $f(1)!$. The same argument leads us to divide by all other $f(i)!$.

Now we have an exact formula for $|F|$, but it is hard to interpret, so we replace it by a simpler approximation. We need a second result from Math 55, namely Stirling's formula for approximating $n!$:

$$n! \approx \sqrt{2\pi} n^{n+.5} e^{-n}$$

This is a good approximation in the sense that the ratio $n!/[\sqrt{2\pi} n^{n+.5} e^{-n}]$ approaches 1 quickly as n grows. (In Math 55 we motivated this formula by the approximation $\log n! = \sum_{i=2}^{n} \log i \approx \int_1^n \log x \, dx$.) We will use Stirling's formula in the form

$$\log_2 n! \approx \log_2 \sqrt{2\pi} + (n + .5) \log_2 n - n \log_2 e$$

Stirling's formula is accurate for large arguments, so we will be interested in approximating $\log_2 |F|$ for large n. Furthermore, we will actually estimate $\frac{\log_2 |F|}{n}$, which can be interpreted as the *average number of bits per character* to send a long file. Here goes:

$$\begin{aligned} \frac{\log_2 |F|}{n} &= \frac{\log_2(n!/(f(1)! \cdots f(c)!))}{n} \\ &= \frac{\log_2 n! - \sum_{i=1}^{c} \log_2 f(i)!}{n} \end{aligned}$$

$$
\begin{aligned}
\approx \quad & \frac{1}{n} \cdot [\log_2 \sqrt{2\pi} + (n + .5) \log_2 n - n \log_2 e \\
& - \sum_{i=1}^{c} (\log_2 \sqrt{2\pi} + (f(i) + .5) \log_2 f(i) - f(i) \log_2 e)] \\
= \quad & \frac{1}{n} \cdot [n \log_2 n - \sum_{i=1}^{c} f(i) \log_2 f(i) \\
& + (1 - c) \log_2 \sqrt{2\pi} + .5 \log_2 n - .5 \sum_{i=1}^{c} \log_2 f(i)] \\
= \quad & \log_2 n - \sum_{i=1}^{c} \frac{f(i)}{n} \log_2 f(i) \\
& + \frac{(1 - c) \log_2 \sqrt{2\pi}}{n} + \frac{.5 \log_2 n}{n} - \frac{.5 \sum_{i=1}^{c} \log_2 f(i)}{n}
\end{aligned}
$$

As n gets large, the three fractions on the last line above all go to zero: the first term looks like $O(1/n)$, and the last two terms look like $O(\frac{\log_2 n}{n})$. This lets us simplify to get

$$
\begin{aligned}
\frac{\log_2 |F|}{n} \quad \approx \quad & \log_2 n - \sum_{i=1}^{c} \frac{f(i)}{n} \log_2 f(i) \\
= \quad & \log_2 n - \sum_{i=1}^{c} p(i) \log_2 np(i) \\
= \quad & \sum_{i=1}^{c} (\log_2 n) p(i) - \sum_{i=1}^{c} p(i) \log_2 np(i) \\
= \quad & \sum_{i=1}^{c} p(i) \log_2 n/np(i) \\
= \quad & \sum_{i=1}^{c} p(i) \log_2 1/p(i)
\end{aligned}
$$

Normally, the quantity $\sum_i p(i) \log_2 1/p(i)$ is denoted by H.

How much more space can Huffman coding take to encode a file than Shannon's lower bound Hn? A theorem of Gallagher (1978) shows that at worst Huffman will take $n \cdot (p_{max} + .086)$ bits more than Hn, where p_{max} is the largest of any p_i. But it often does much better.

Furthermore, if we take blocks of k characters, and encode them using Huffman's algorithm, then, for large k and for n tending to infinity, the average length of the encoding tends to the entropy.