Indexes and inverted files

You are all familiar with indexes in books. You wish to look up some term like "entropy" or "ASCII" which you hope is discussed in the book. So you turn to the alphabetically ordered list of terms (index) at the back of the book. For each of these terms there is a list of page numbers. These are pointers to pages in which the term appears in the book.

For such indexes, the book is partitioned into pages that are numbered from 1 to m. When you find a term in the index, you note each of the page numbers turn to the corresponding pages, and do a linear search for the term in the page. Typically, if the term appears multiple times on a page, the page number is still only is listed once in the index. So you need to scan the entire page to see all occurances of the term. Also, any terms that are in the book do not appear in the index. The reason is that the publishers of the book want to keep the index small.

It should be obvious that such real indexes typically do not allow you to reconstruct the original book. They only provide a *subset* of words that appear on each of the pages, and provide no information about where each word appears on each page or how many times each word appears on a page. The question of which words to choose to keep the index small but still useful is very interesting, but not what they course is about. If you would like to learn more about this topic, look at the very nice book **Managing Gigabytes** (See website `www.cs.mu.oz.au/mg/`.) Today, we will keep it simple and consider the problem of compressing an index that *does* provide all information about the original file.

Suppose we have a sequence of n events from some alphabet $\{A_1, A_2, \ldots, A_N\}$. Suppose that A_i occurs n_i times in the sequence. This is the same n_i as we used in our analysis of move-to-front.

For each symbol A_i, we represent a list of positions in the sequence where A_i occured. We have N lists. These lists together define the *inverted file*. The inverted file has the form:

$$\begin{array}{l}(n_1; t_1^1, t_2^1, \ldots, t_{n_1}^1)\\ (n_2; t_1^2, t_2^2, \ldots, t_{n_2}^2)\\ :\\ (n_N; t_1^N, t_2^N, \ldots, t_{n_N}^N)\end{array}$$

Note that we specify how many elements are in each "inverted list". Also note that it is now possible to reconstruct the original file from the inverted file. We can think of the occurance times t_j^i using the bit matrix that we saw in the move-to-front lecture.

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & \ldots \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & \ldots \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & \ldots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & \ldots \end{pmatrix}$$

How could we compress the inverted file?

If we use a prefix code for the integers then there is a straightforward method to encode the inverted file. We just encode each of the numbers in the above lists. This is not very effective, though, since the numbers can be large (up to n) and large numbers would use too many bits.

A more effective method is to transform the lists above into a list of "gaps" which amount to run lengths in the rows of the above matrix:

$$
\begin{aligned}
&(n_1; t_1^1,\ t_2^1 - t_1^1, t_3^1 - t_2^1, \ldots, t_{n_1}^1 - t_{n_1-1}^1)\\
&(n_2; t_1^2,\ t_2^2 - t_1^2,\ \ \ldots, t_{n_2}^2 - t_{n_2-1}^2)\\
&:\\
&(n_i; t_1^i,\ t_2^i - t_1^i,\ \ \ldots, t_{n_i}^i - t_{n_i-1}^i)\\
&:\\
&(n_N; t_1^N,\ t_2^N - t_1^N, \ldots, t_{n_N}^N - t_{n_N-1}^N)
\end{aligned}
$$

A gap size j indicates that there are $j-1$ non-occurances of a symbol followed by an occurance. Symbol A_i appears in the sequence n_i times. So $\frac{n_i}{n}$ can be treated as the probability of symbol A_i occuring at any position.

One way to encode the n_i occurrences of A_i would be to encode the gaps,

$$t_1^i,\ t_2^i - t_1^i,\ \ \ldots, t_{n_i}^i - t_{n_i-1}^i.$$

For example, we could use the Bernouilli trial model so that a runlength j in list i occurs with probability $(\frac{n_i}{n})^{j-1}(1-\frac{n_i}{n})$. For each list i, the encoder could use a Golomb code with parameter b, where b is chosen appropriately for the frequency $\frac{n_i}{n}$. (See Assignment 1.) Intuitively, for symbols A_i that occur more often in the file, n_i is a larger number, and so the gaps (runlengths) are shorter. Shorter gaps imply a smaller b for the Golomb code.

The encoder begins by telling the decoder what is n. Then, for each inverted list i, the encoder tells the decoder what is n_i. The encoder could also the Golomb code constant b for this i. Or, the encoder and decoder could have agreed beforehand on a method for computing b, given n and n_i. For example, since we want

$$\lambda_i \approx \log \frac{1}{(1-\frac{n_i}{n})^{j-1}(\frac{n_i}{n})} = (j-1)\log(\frac{n}{n-n_i}) + \log(\frac{n}{n_i})$$

it follows from the earlier arguments of Golomb coding that a reasonable b to use would be

$$b = round(\frac{1}{\log(\frac{n}{n-n_i})}).$$

One final point: the Bernouilli model assumes independent trials. Clearly this model is incorrect for the inverted file application. For example, once the decoder has decoded one of the inverted lists, the probabilities for the other inverted lists change, since there is exactly one "1" in each column of the above matrix. The extreme case is that, once the decoder knows the first $N-1$ inverted lists, it could easily compute the inverted list for A_N. This implies that the encoding is far from optimal. (Tradeoff: simplicity vs. optimality)

Facsimiles

Let's briefly look an application that uses some of the ideas we have seen up to now: fax compression. Fax machines are typically used to send copies of documents that are printed or written text and

drawings. A fax image takes a rectangular page and scans it, producing a 2D bit grid, where each bit is either black or white. Each bit represents about $\frac{1}{8}mm \times \frac{1}{8}mm$ square. For one page, the grid is roughly of size about 1600 x 2400 bits. This 2D grid thus defines nearly 4 million bits per page. Each bit is called a *pixel* or picture element.

For the past few decades, it was common for people to communicate fax data with modem at a rate of about 9600 bits per second. A very fast modem was about 64,000 bits per second. Thus even a very fast modem would require many seconds to send or receive a fax *if this data were send as raw bits.*

A standard was developed for compressing fax documents. This standard was developed by an organization called the CCITT (based in Switzerland) and so it is known as the CCITT standard. The standard actually consists of several schemes. Let's just briefly discuss a simpified version of one of them.

Each horizontal line in the 2D bit array is encoded as an alternating set of white and black runs. Two different Huffman codes are used, one for black runs and one for white runs. The reason two codes are used is that typical documents have sparse black text on a white background, and so the vast majority of pixels are white. White runlengths tend to be much longer than black runlengths. Let w denote white and b denote black. Here are some of the details (of Group 3 fax):

- For each row, one bit is spent to say whether the first run is black or white.
- Because errors sometimes occur in transmitting bits, each line is terminated with the codeword (000000000001) for a special end of line (EOL) symbol. There is an EOL symbol in both the white and black Huffman codes. The EOL symbol serves to synchronize the encoder and decoder in case a bit in the code has not been received correctly. If the EOL is not found for a given row, then the decoder knows that an error has occured. (It can just skip bits and scan until it finds the next EOL codeword for the next line, and ignore the intervening lines.[1] Or it could abort.)
- Rather than having codewords for all possible runlengths up to the number of pixels in a row), there are codes for each possible runlength from 0 to 63 (called *termination code words*), and codewords for runlengths that are multiples of 64, namely 64,128, 192, ..., up to 2560 (called *makeup codes*). When a runlength is greater than 63, it is encoded by a makeup code + termination code pair. For example, a runlength of 209 is encoded by makeup code for 192 and termination code for 17. A runlength of 128 is encoded using a makeup code for 128 and a termination code for a runlength of 0.

  Thus, the alphabet for which we need codewords is $\{0, 1, 2, \ldots, 64, 128, 192, \ldots, 64j_{max}\}$.

  Note that termination codes for runlength 0 are necessary here, for the rare case that the runlength is an exact multiple of 64. Why? After the codeword for 64 (or 128, etc), the decoder needs to know whether the subsequent bits encode a termination code for the same color (black or white) or the makeup code for the opposite color.
- Where do the codewords come from? The codewords are part of the CCITT standard. They were determined by taking several "typical" documents and examining the probabilities of runlengths for those documents. The standard codes are a Huffman code for runlengths from

[1] Such a search is unlikely to lead to mistakes since no other codeword has more than 7 consecutive 0's.

these test documents. I emphasize that a *different code is needed for the black runlengths vs the white runlengths*, since the probabilities of runs are very different in the two cases.

Note that this runlength coding method seems slightly different from the method we have discussed earlier. Previously, we considered only "runs" of $i-1$ 0's followed by a 1. Now we are doing is slightly differently by explicitly encoding runs of 0's and 1's (whites and blacks).

For typical documents, the method typically gives a compression of about 10:1, meaning that 3 million pixels (each of 1 bit) are compressed to a code of about 300,000 bits.

2D schemes

The scheme described above encodes each row independently. This is intuitively not optimal since neighboring rows tend to be similar. One can improve the scheme by taking advanage of the similarity between rows. Let's assume that the encoder has transmitted some current row and the decoder has decoded it. How should the next row be transmitted? The decoder is expecting that the next row will be very similar. The encoder can therefor compute the differences between rows: consider a string which has 0's in positions where the value in the next row is equal to that of the previous row, and 1's in positions where the value in the next row is the same. That is,

$$XOR(next\ row, previous\ row)$$

This string would be mostly 0's typically, so its runlengths could be compressed. (Note that in this scheme, the alphabet is still binary, but now it is $\{same, different\}$ as directly above, and we could encode runs of $i-1$ "sames" followed by a "different". This is the basic idea. For the details, I refer you to the CCITT standard.