CS 493: Algorithms for Massive Data Sets | **Dictionary-based compression**
February 14, 2002 | **Scribe: Tony Wirth**

This lecture will explore two adaptive dictionary compression schemes: LZ77 and LZ78. We use the word adaptive, because the dictionary is not an a priori model of the text, but changes as the text is processed. A compromise approach is to scan the whole of the text in advance and then construct a semi-static dictionary. Often the cost of sending the dictionary as a preamble is not worth the improvement in start-up compression.

The aim of both LZ77 and LZ78, published by Abraham Ziv and Jacob Lempel, is to encode a new phrase by providing a reference to an already-seen phrase that is very similar. These methods are easy to implement; decoding uses little memory and is particularly quick.

LZ77

We can explain LZ77 most easily by looking the decoding process. The decoder sees a sequence of triples, consisting of a back-pointer, a length and a character. For example, consider the sequence

$$< 0, 0, \mathtt{a} >, < 0, 0, \mathtt{b} >, < 2, 1, \mathtt{a} >, < 3, 2, \mathtt{b} >, < 5, 3, \mathtt{b} >, < 1, 10, \mathtt{a} > .$$

The first two triples simply refer to the symbols a and b. The third triple says to look back two positions and copy a string of length 1 and then add a final a: that is the string aa. So far the decoded string is abaa. The fourth triple decodes to baab, overall abaabaab. The fifth is abaabb. Now we see something that at first glance is contradictory: a pointer one position back, but to a string of length ten. Consider that the decoder is writing the text in an array, so it moves the copy-from pointer one space to the left and moves it to the right in step with the copy-to pointer. Hence $< 1, 10, \mathtt{a} >$ is a kind of run-length encoding for bbbbbbbbbba and the final string is abaabaababaabbbbbbbbbba.

So how are these triples actually encoded? Firstly, the pointer is capped to a maximum offset of 8Kb, so it can be represented with 13 bits: this is a reasonable bound for English text. The phrase length is often set to be no more than sixteen characters, hence a four bit representation. Needless to say, neither the pointer nor length distributions are uniform in practice. There is a bias towards smaller quantities which therefore demand shorter codewords. In some implementations the new symbol is not included in the triple all the time. Sample encoding and decoding algorithms are presented in Figure 1.

In producing a fast encoding algorithm, we need to consider how to accelerate the search for the longest phrase match in the window. Various data structures such as a trie, a hash table or a binary search tree could be useful. For instance, we could use pairs of characters as the search index, with each index entry assigned a linked list of pointers to occurrences of the pair. We might want to limit the length of these linked lists, at some cost to the compression effectiveness.

To encode the text $S[1 \ldots N]$ using the LZ77 method, with a sliding window of W characters,

1. Set $p \leftarrow 1$.
2. While more text remains:
   (a) Search for the longest match for $S[p \ldots]$ in $S[p - W \ldots p - 1]$. Suppose that the match occurs at position m and is of length l.
   (b) Output the triple $< p - m, l, S[p + l] >$.
   (c) Set $p \leftarrow p + l + 1$.

To decode the text $S[1 \ldots N]$ using the LZ77 method, with a sliding window of W characters,

1. Set $p \leftarrow 1$.
2. For each triple $< f, l, c >$ in the input do
   (a) Set $S[p \ldots p + l - 1] \leftarrow S[p - f \ldots p - f + l - 1]$.
   (b) Set $S[p + l] \leftarrow c$.
   (c) Set $p \leftarrow p + l + 1$.

Figure 1: Encoding and decoding using LZ77.

GZIP

One of the best compression programs derived from the LZ77 scheme is GZIP. It uses a hash table with a three-symbol index, where the entries are linked to bounded lists of pointers. The user can select the point of compromise between throughput and compression. Often it is useful to limit the list length when there are long runs of some symbol in which case little is gained by having many entries. The lists can be reordered to favour recent matches.

Instead of encoding triples, gzip sends either a pointer and length pair, or a new symbol. The method of doing this is a little unusual. One set of codewords is used for both match lengths and characters; another is used for the pointer offsets. The "match length" is sent first so the decoder can tell immediately whether it really is a match length, and thus a pointer will follow, or whether it is actually a new character. The user can instruct the GZIP encoder to scan ahead to see whether it might be better to encode a raw character instead of a pointer, so that compression of the following symbols is improved. Once again, this would come at the cost of GZIP's throughput. Finally, the GZIP Huffman codes are not generated dynamically (and therefore slowly); the raw text is read in blocks of 64 kilobytes and codes are generated for each block. Although this means GZIP is not really a one-pass encoder, the blocks are small enough to fit into memory or perhaps even cache.

LZ78

The LZ77 scheme allows us to point back to any substring, but places a limit on the size of the prior window. In contrast, LZ78 restricts the types of substrings that may be referred to, but not their locations. The hope is that we can avoid some redundancy by keeping only one copy of prior phrases.

Again, it is instructive to study the decoding process. Imagine that the text `abaababaa` has already been seen. The LZ78 scheme breaks the text into the phrases `a, b, aa, ba, baa`. The idea is to split the new text into the longest phrase that has been seen already, plus one new character at the end. New phrases are encoded by stating the old phrase number and the new end symbol. The text so far would have been sent as

$$< 0, \mathtt{a} >< 0, \mathtt{b} >< 1, \mathtt{a} >< 2, \mathtt{a} >< 4, \mathtt{a} >,$$

where phrase 0 refers to the empty phrase. If the decoder then saw the input,

$$< 4, \mathtt{b} >< 2, \mathtt{b} >< 7, \mathtt{b} >< 8, \mathtt{b} >,$$

it would print `bab`, then `bb`, then `bbb` and finally `bbbb`. The encoder can parse preceding text quickly by storing phrases in a trie. However, the trie can grow quite large and may need to be destroyed every so often. To avoid poor start-up compression, it is handy to have available a trie based on the last few hundred characters. Although LZ78 encoding is often faster than that of LZ77, decoding is more involved as the LZ78 decoder also needs the parse trie.

Table 1: Throughput (relative to encoding speed of *compress*) and compression effectiveness of a selection of programs on the Canterbury corpus.

METHOD	Encoding speed	Decoding speed	Compression (bpc)
cat	0.2	0.2	8.00
compress	1.0	0.6	3.31
GZIP-f	1.1	0.4	2.91
GZIP-b	7.0	0.3	2.53
bzip2	5.5	2.0	2.23
PPM	5.3	5.9	2.11

LZW

LZW is the basis of the *compress* utility and is a variant of LZ78. Unlike LZ78, LZW does not send an extra character after the phrase number. Instead it has all the symbols already entered as phrases; phrases 0 to 127 are the ASCII symbols, for instance. Each new phrase is constructed from the one just coded plus the first character of the next phrase. If the encoder saw the stream, `abaababbaabaabaa`, it would break it into `a,b,a,ab,ab,ba,aba,abaa`. The corresponding new phrases would be, (`ab`,128), (`ba`,129), (`aa`,130), (`aba`,131), (`abb`,132), (`baa`,133), (`abaa`,134), while the phrase codes sent to the decoder would be 97, 98, 97, 128, 128, 129, 131, 134. You might ask, how does the decoder know what phrase 134 is, when it has not yet seen the last symbol of phrase 134? Well, it knows that phrase 134 is the same as 131, with another character appended. Therefore it knows that the new text will be `aba?`, that is, phrase 131 plus unknown. But that tells it that the last character of phrase 134 was an `a`, as `a` is the first letter of this new phrase. Therefore phrase 134 is `abaa`.

Having seen both symbol-based and dictionary-based compression schemes, it is worth comparing the performance of the best implementations (Table 1). Note that *bzip2* is an implementation of the Burrows-Wheeler transform that uses a sophisticated selection of Huffman codes, as against arithmetic coding, to keep throughput high without compromising significantly on compression. There is little point in using *compress*, given that GZIP-f is just as quick and is significantly more effective. The better-compressing version of GZIP suffers only in encoding speed, but actually decodes faster than GZIP-f, as it uses the same algorithm, but has slightly less information to decode. So it seems that if compression is the ultimate aim, then PPM is still best, but for speed, go for GZIP.

A problem to consider

Let X be a random variable representing the number of tosses of a fair coin until a head appears. In this case the best set of codewords for X, that is those that minimize the expected codeword length, is the set of sequences of tosses themselves represented in binary. For next class, work out what the codewords ought to be if the coin is not fair, but has some known probability of showing heads.

Much of this material was based on:
IH Witten, A Moffat, TC Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*, Second edition, Morgan Kaufmann, San Francisco, 1999, pp74–84.