Dictionary coding

Simple variant of coding with a variable number of symbols and fixlength codewords.

Create a dictionary containing 2^b different symbol sequences and code them with codewords of length b.

Static dictionaries can work ok when coding sources with wellknown statistics, in other cases we want a method for adaptively constructing a dictionary.

Lempel-Ziv coding

Code symbol sequences using references to previous data in the sequence.

There are two main types:

- Use a history buffer, code a partial sequence as a pointer to when that particular sequence last appeared (LZ77).
- Build a dictionary of all unique partial sequences that appear. The codewords are references to earlier words (LZ78).

Lempel-Ziv coding, cont.

The coder and decoder don't need to know the statistics of the source. Performance will asymptotically reach the entropy rate. A coding method that has this property is called *universal*.

Lempel-Ziv coding in all its different variants are the most popular methods for file compression and archiving, eg zip, gzip, ARJ and compress.

The image coding standards GIF and PNG use Lempel-Ziv.

The standard V.42bis for compression of modem traffic uses Lempel-Ziv.

LZ77

Lempel and Ziv 1977.

View the sequence to be coded through a sliding window. The window is split into two parts, one part containing already coded symbols (search buffer) and one part containing the symbols that are about to coded next (look-ahead buffer).

Find the longest sequence in the search buffer that matches the sequence that starts in the look-ahead buffer. The codeword is a triple $< o, l, c >$ where o is a pointer to the position in the search buffer where we found the match (offset), l is the length of the sequence, and c is the next symbol that doesn't match. This triple is coded using a fixlength codeword. The number of bits required is

$$\lceil \log S \rceil + \lceil \log(W + 1) \rceil + \lceil \log N \rceil$$

where S is the size of the search buffer, W is the size of the look-ahead buffer and N is the alphabet size.

Improvements of LZ77

It is unnecessary to send a pointer and a length if we don't find a matching sequence. We only need to send a new symbol if we don't find a matching sequence. Instead we can use an extra flag bit that tells if we found a match or not. We either send $< 1, o, l >$ eller $< 0, c >$. This variant of LZ77 is called LZSS (Storer and Szymanski, 1982).

Depending on buffer sizes and alphabet sizes it can be better to code short sequences as a number of single symbols instead of as a match.

In the beginning, before we have filled up the search buffer, we can use shorter codewords for o and l.

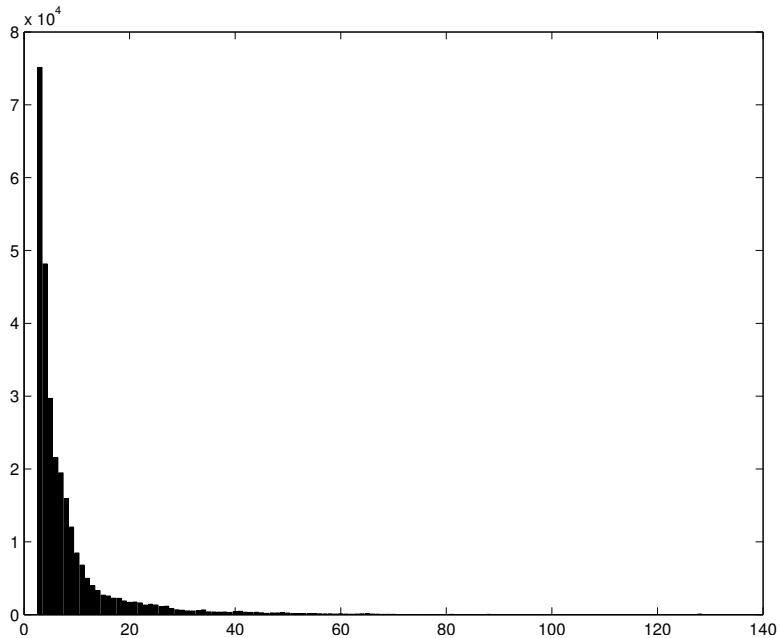
All o, l and c are not equally probable, so we can get even higher compression by coding them using variable length codes (eg Huffman codes).

Buffer sizes

In principle we get higher compression for larger search buffers. For practical reasons, typical search buffer sizes used are around $2^{15} - 2^{16}$.

Very long match lengths are usually not very common, so it is often enough to let the maximum match length (ie the look-ahead buffer size) be a couple of hundred symbols.

Example: LZSS coding of `world192.txt`, buffer size 32768, match lengths 3-130 (128 possible values). Histogram for match lengths:



Optimal LZ77 coding

Earlier we described a greedy algorithm (always choose the longest match) for coding using LZ77 methods. If we want to minimize the average data rate (equal to minimizing the total number of bits used for the whole sequence), this might not be the best way.

Using a shorter match at one point might pay off later in the coding process. In fact, if we want to minimize the rate, we have to solve a shortest path optimization problem. Since this is a rather complex problem, this will lead to slow coding.

Greedy algorithms are often used anyway, since they are fast and easy to implement.

An alternative might be to look ahead just a few steps and choose the match that gives the lowest rate for those few steps.

DEFLATE

Deflate is a variant of LZ77 that uses Huffman coding. It is the method used in zip, gzip and PNG.

Data to be coded are bytes.

The data is coded in blocks of arbitrary size (with the exception of uncompressed blocks that can be no more than 65536 bytes).

The block is either sent uncompressed or coded using LZ and Huffman coding.

The match lengths are between 3 and 258. Offset can be between 1 and 32768.

The Huffman coding is either fixed (predefined codewords) or dynamic (the codewords are sent as side information).

Two Huffman codes are used: one code for single symbols (literals) and match lengths and one code for offsets.

Symbols and lengths

The Huffman code for symbols and lengths use the alphabet $\{0, 1, \ldots, 285\}$ where the values 0-255 are used for symbols, the value 256 marks the end of a block and the values 257-285 are used to code lengths together with extra bits:

	extra bits	length		extra bits	length		extra bits	length
257	0	3	267	1	15,16	277	4	67-82
258	0	4	268	1	17,18	278	4	83-98
259	0	5	269	2	19-22	279	4	99-114
260	0	6	270	2	23-26	280	4	115-130
261	0	7	271	2	27-30	281	5	131-162
262	0	8	272	2	31-34	282	5	163-194
263	0	9	273	3	35-42	283	5	195-226
264	0	10	274	3	43-50	284	5	227-257
265	1	11,12	275	3	51-58	285	0	258
266	1	13,14	276	3	59-66			

Offset

The Huffman code for offset uses the alphabet $\{0, \ldots, 29\}$. Extra bits are used to exactly specify offset 5-32768

	extra bits	offset		extra bits	offset		extra bits	offset
0	0	1	10	4	33-48	20	9	1025-1536
1	0	2	11	4	49-64	21	9	1537-2048
2	0	3	12	5	65-96	22	10	2049-3072
3	0	4	13	5	97-128	23	10	3073-4096
4	1	5,6	14	6	129-192	24	11	4097-6144
5	1	7,8	15	6	193-256	25	11	6145-8192
6	2	9-12	16	7	257-384	26	12	8193-12288
7	2	13-16	17	7	385-512	27	12	12289-16384
8	3	17-24	18	8	513-768	28	13	16385-24576
9	3	25-32	19	8	769-1024	29	13	24577-32768

Fixed Huffman codes

Codewords for the symbol/length alphabet:

value	number of bits	codewords
0 - 143	8	00110000 - 10111111
144 - 255	9	110010000 - 111111111
256 - 279	7	0000000 - 0010111
280 - 287	8	11000000 - 11000111

The reduced offset alphabet is coded with a five bit fixlength code.

For example, a match of length 116 at offset 12 is coded with the codewords 11000000 0001 and 00110 11.

Dynamic Huffman codes

The codeword lengths for the different Huffman codes are sent as extra information.

To get even more compression, the sequence of codeword lengths are runlength coded and then Huffman coded (!). The codeword lengths for this Huffman code are sent using a three bit fixlength code.

The algorithm for constructing codewords from codeword lengths is specified in the standard.

Coding

What is standardized is the syntax of the coded sequence and how it should be decoded, but there is no specification for how the coder should work. There is a recommendation about how to implement a coder:

The search for matching sequences is not done exhaustively through the whole history buffer, but rather with hash tables. A hash value is calculated from the first three symbols next in line to be coded. In the hash table we keep track of the offsets where sequences with the same hash value start (hopefully sequences where the first three symbol correspond, but that can't be guaranteed). The offsets that have the same hash value are searched to find the longest match, starting with the most recent addition. If no match is found the first symbol is coded as a literal. The search depth is also limited to speed up the coding, at the price of a reduction in compressionen. For instance, the compression parameter in gzip controls for how long we search.

Documents

See also:
ftp://ftp.uu.net/pub/archiving/zip/
ftp://ftp.uu.net/graphics/png/
http://www.ietf.org/rfc/rfc1950.txt
http://www.ietf.org/rfc/rfc1951.txt
http://www.ietf.org/rfc/rfc1952.txt
http://www.ietf.org/rfc/rfc2083.txt

LZ78

Lempel and Ziv 1978.

A dictionary of unique sequences is built. The size of the dictionary is S. In the beginning the dictionary is empty, apart from index 0 that means "no match".

Every new sequence that is coded is sent as the tuple $< i, c >$ where i is the index in the dictionary for the longest matching sequence we found and c is the next symbol of the data that didn't match. The matching sequence plus the next symbol is added as a new word to the dictionary.

The number of bits required is

$$\lceil \log S \rceil + \lceil \log N \rceil$$

The decoder can build an identical dictionary.

LZ78, cont.

What to do when the dictionary becomes full? There are a few alternatives:

- Throw away the dictionary and start over.
- Keep coding with the dictionary, but only send index and do not add any more words.
- As above, but only as long as the compressionen is good. If it becomes too bad, throw away the dictionary and start over. In this case we might have to add an extra symbol to the alphabet that informs the decoder to start over.

LZW

LZW is a variant of LZ78 (Welch, 1984).

Instead of sending a tuple $< i, c >$ we only send index i in the dictionary. For this to work, the starting dictionary must contain words of all single symbols in the alphabet.

Find the longest matching sequence in the dictionary and send the index as a new codeword. The matching sequence plus the next symbol is added as a new word to the dictionary.

LZ comparison

A comparison between three different LZ implementations, on three of the test files from the project lab. All sizes are in bytes.

	`world192.txt`	`alice29.txt`	`xargs.1`
original size	2473400	152089	4227
`compress`	987035	62247	2339
`gzip`	721413	54191	1756
`7z`	499353	48553	1860
`pack`	1558720	87788	2821
`ppmd`	374361	38654	1512
`paq6v2`	360985	36662	1478

compress uses LZW, gzip uses deflate, 7z uses LZMA (a variant of LZ77 where the buffer size is 32MB and all values after matching are modeled using a Markov model and coded using an arithmetic coder).

Compression of test data

The same data as the previous slide, but with performance given as bits per symbol. A comparison is also made with some estimated entropies.

	`world192.txt`	`alice29.txt`	`xargs.1`
original size	8	8	8
`compress`	3.19	3.27	4.43
`gzip`	2.33	2.85	3.32
`7z`	1.62	2.55	3.52
`pack`	5.04	4.62	5.34
`ppmd`	1.21	2.03	2.86
`paq6v2`	1.17	1.93	2.80
$H(X_i)$	5.00	4.57	4.90
$H(X_i \mid X_{i-1})$	3.66	3.42	3.20
$H(X_i \mid X_{i-1}, X_{i-2})$	2.77	2.49	1.55

GIF (Graphics Interchange Format)

Two standards: GIF87a and GIF89a.

A virtual screen is specified. On this screen rectangular images are placed. For each little image we send its position and size.

A colour table of maximum 256 colours is used. Each subimage can have its own colour table, but we can also use a global colour table.

The colour table indices for the pixels are coded using LZW. Two extra symbols are added to the alphabet: ClearCode, which marks that we throw away the dictionary and start over, and EndOfInformation, which marks that the code stream is finished.

Interlace: First lines $0, 8, 16, \ldots$ are sent, then lines $4, 12, 20, \ldots$ then lines $2, 6, 10, \ldots$ and finally lines $1, 3, 5, \ldots$

In GIF89a things like animation and transparency are added.

PNG (Portable Network Graphics)

Introduced as a replacement for GIF, partly because of patent issues with LZW (these patents have expired now).

Uses deflate for compression.

Colour depth up to 3×16 bits.

Alpha channel (general transparency).

Can exploit the dependency between pixels (do a prediction from surrounding pixels and code the difference between the predicted value and the real value), which makes it easier to compress natural images.

PNG, cont.

Supports 5 different predictors (called filters):

0. No prediction
1. $\hat{I}_{ij} = I_{i,j-1}$
2. $\hat{I}_{ij} = I_{i-1,j}$
3. $\hat{I}_{ij} = \lfloor (I_{i-1,j} + I_{i,j-1})/2 \rfloor$
4. Paeth (choose the one of $I_{i-1,j}$, $I_{i,j-1}$ and $I_{i-1,j-1}$ which is closest to $I_{i-1,j} + I_{i,j-1} - I_{i-1,j-1}$)

Test image Goldhill

We code the test image Goldhill and compare with our earlier results:

GIF	7.81 bits/pixel
PNG	4.89 bits/pixel
JPEG-LS	4.71 bits/pixel
Lossless JPEG	5.13 bits/pixel

GIF doesn't work particularly well on natural images, since it has trouble exploiting the kind of dependency that is usually found there.