LZP : a new data compression algorithm

by Charles Bloom (cbloom@mail.utexas.edu)

Introduction

A new compression algorithm is presented which may either be considered an improvement to dictionary coding or an improvement to context coding. This algorithm works by reducing the set of available window position, for an LZ77 encoder to match from; thus fewer bits are required to indicate the match-from location, and higher compression is attained because the average match length does not decrease as much as the output code size does. This algorithm is shown to be the best solution currently available in all situations, including archivers, distribution, and on-line compression such as disk compression or network datagram compression. Four variants are presented; LZP1 is faster than LZRW and gets higher compression; LZP2 is fast enough to be used in on-line applications (faster than the commonly used Stac LZS algorithm) and achieves compression on the order of LZSS; LZP3 achieves compression comparable to that of the most advanced hybrid LZ77+Huffman methods (such as the Zip algorithm) while operating at three times the speed of PkZip; LZP4 attains an average on the Calgary Corpus of 2.29 bits per byte, comparable to that of the state-of-the-art PPMD+ algorithm, while operating much faster and using less memory.

Motivation

The basic concept behind LZP is that an LZ77 coder reserves too many bits for the “match-from-position” (henceforth “the source”) (note: the term ‘LZ77’ is used to refer to all variations on the LZ77 compression method [14,18]). For example, using a 64k byte window, an LZ77 coder with a minimum match length of four typically matches to an average string length of seven. Thus if the source is written in 16 bits (neglecting the match-flag) the compression level is 7/2 = 3.5 raw bytes per compressed byte. If we could predict where the long matches will come from, we could write the source in fewer bits - say 8 ; the new match length must be less than or equal to the old one, but if it is greater than half of the old one, then higher compression is attained. For example, if the average match length is now 4, then 4/1 = 4.0 raw bytes per compressed byte is attained.

This can be viewed in the following way : an offset is only worth adding to the set of choices if its “information content” is greater than the increase in size of the source specification.

The method which LZP uses to select offsets is finite context Markov prediction : the current order-n finite context is used to select all previous occurrences of the same context. For example, if part of the raw data is “the cat”, and the ‘c’ is the current position , then all strings which follow the phrase “the ” would be matched against.

It has been discovered through empirical study that no string is worth the extra space required to specify it : i.e. on average, adding a source to the match-from set does not increase the match length as much as it increases the output code size. Thus the ideal size of the set for the LZ77 coder to match from is a one member set. With a one member set the LZ77 coder need not write any bits to specify the source : only a match-flag need be written. Thus only one source pointer may be selected; this is done by using the most recent occurrence of the current order-n context.

The most recent occurrence of a finite context is an excellent predictor for the following character. For example, the PPMCB algorithm [3] uses only the most recent context, and attains a Calgary Corpus average of 2.35 bpb (‘bpb’ is used to mean ‘number of output bits per input byte, averaged on the Calgary Compression Corpus’).

Context-modeling using only the highest order context is also extremely effective. On almost all files, at least half of the characters are written with the highest order model. Statistics follow for the file ‘trans’ from the Calgary Corpus [15], generated by a standard order-5-4-3-2-1-0 PPM compressor :

```
Order-5 :  71831 ->  7529 = 0.838 bpb
Order-4 :   4130 ->   955 = 1.849 bpb
Order-3 :   4922 ->  1678 = 2.727 bpb
Order-2 :   5752 ->  2658 = 3.696 bpb
Order-1 :   5067 ->  2998 = 4.733 bpb
Order-0 :   1994 ->  1483 = 5.949 bpb
Total   :  93696 -> 17301 = 1.477 bpb
```

Figure 1 : Compression by individual context levels of PPM on ’trans’

The algorithm has been named LZP because it is an LZ based algorithm which used prediction to select only a single offset to code with, thus it is LZ+Prediction : LZP.

The LZP Algorithm

The groundwork has now been laid for the LZP algorithm. The compression process is described here in the most general way, so that all the LZP variants are encompassed in it.

For each byte of the input stream, the preceding N bytes are taken to form the order-N finite context. This context, C, is used to perform a lookup in the “index table”. The result of this lookup is a pointer to somewhere in the preceding data stream, P. The current string, S, is matched against P and the length of match, L, between the two is found. S is then inserted into the index table, at the location where P was found. Furthermore, if context-confirmation is used (see below), the value of C is also stored in the index table.

If L is 0, a negative match-flag is written, and then a literal is written.

If L is 1 or more, a match-flag is written and then L is written.

Note that there is no minimum match length - even a match of length one is written. Also note the spare nature of the encoded bit-stream : only match flags, lengths and literals are written.

The LZP algorithm implicitly only updates the index table at phrase boundaries. This was (surprisingly) found to help compression, and may be part of why LZP4 does so well.

Pseudo-code for a generic C implementation of LZP is included in the appendix.

Implementations of the LZP Algorithm

There are four implementations of LZP presented herein : LZP1 - 4, with LZP1 being the fastest with lowest compression, and LZP4 being the slowest with highest compression.

The ways in which the LZP implementations differ are in how the index table is implemented, and how the output fields (the match-flags, the literals, and the match-lengths) are written.

Index Table Implementation

LZP1 and **LZP2** implement the lookup using a fixed order-3 context and hashing. The context C, a 24-bit word, is converted into a hash table index using the function

H = ((C >> 11) ^ C) & 0xFFF

resulting in a 12-bit hash value. This hash value is immediately used for a direct lookup into an array of pointers to the previously encoded stream. Once P is retrieved, the array value at position H is set to the current pointers. Using this hash function instead of H = C was found to hurt compression by less than 0.01 bpb. By contrast, using the LZRW1 hash function [4] (modified to work on the preceding bytes instead of the following bytes) hurt compression by about 0.1 bpb . If the retrieved pointer, P, is not initialized (i.e. this value of H has never been seen before) then a literal is written without a no-match flag. Hash collisions are not avoided. This was found to hurt compression by less than 0.05 bpb, and greatly improved speed and memory usage. The index table has only 4096 members, each of which is a 2-byte word, resulting in 8k bytes of memory usage. LZP1 and 2 use a 16k byte window, resulting in a total memory usage of 24k bytes (for the string-matching portion of the algorithm). Phrases are not removed as they exit the window; the cyclical nature of the window buffer is exploited, for speed.

LZP3 uses a similar scheme to LZP1 and 2, but with an order-4 context and the hash function

H = ((C >> 15) ^ C) & 0xFFFF

Furthermore, in the index-table, the value of C is also stored. If a hash value H has been seen before, but the value of C in the table is different than the current value of C, then P is set to NULL and the algorithm proceeds as if nothing were in the table; this is called “context confirmation”. Context confirmation was found to be critical for order-4 hashing (an 0.2 bpb improvement) but not as important for lower order hashing (less than an 0.05 bpb improvement). Context confirmation doubles the memory usage of the index table. LZP3 also has the further improvement that if P is not valid (i.e. H not seen or context not confirmed) then the order-3 model is examined, then the order-2, instead of writing a literal immediately. Note that this is different than escaping down the orders: it is simply a method of finding the highest order context which has been seen so far. This method of backward traversal from higher to lower is faster than the traditional method of forward traversal (from lower to higher, such as is used in Alistair Moffat’s PPMC compressor [19]) because most of the time (60% or more) a successful match is made to the highest order context.

LZP4 uses an order-5 context and no hashing. To efficiently perform a lookup, two levels of indexing are used: first the 4 lowest order bytes of C are removed to create I, the index value (I is the same as the order-4 context). Then an index hash H is computed from

H = ((I>>15) ^ I) & 0xFFFF

H is used to index into a 65536-member array. This array contains pointers to linked lists of index values which had the same hash. This list is traversed until the current value of I is matched. This list is updated using an MTF scheme, for speed. Each index node contains a pointer to a second list. This list contains the values of the highest-order character (i.e. C>>32) in each context. If the order-5 character for the current context (C) is found, then that node contains a pointer to the most recent occurrence of that context. This elaborate scheme is used because the order-5 context does not fit in a 32-bit machine word. Note that this structure uses much less memory and is much faster than PPM modeling, because only the most recent occurrence of each context need be maintained. Thus a third list of characters seen at each context is totally avoided. If the order-5 context is not found, then lower order contexts are not used - a literal is written immediately.

Implementation of the output fields

LZP1 uses a byte-aligned output method for maximum speed. Literals are written in 8 bits (uncompressed). Match flags and lengths are grouped together into control bytes. The match flag is written using the following scheme :

1 = two literals

01 = a literal then a match

00 = a match

This method achieves 0.1 bpb better compression than the simpler scheme of 0 = literal, 1 = match , because literals tend to occur much more often than matches (this scheme implies P(literal) = 2/3). Match lengths are written using an expanding variable-length-integer code. First, two bits are written, in which the top value (3, or binary 11) is reserved to mean "more bits follow". Then 3 bits are written with the top value (7, or binary 111) reserved to mean "more bits follow". Then, 5 bits are written, and finally an 8 bit repeating code is used, with the hex value 0xFF reserved to mean "another 8 bits follow". A brief table of values follows:

length	code	length	code
1	00	10	11 110
2	01	11	11 111 00000
3	10	12	11 111 00001
4	11 000	41	11 111 11110
5	11 001	42	11 111 11111 00000000
6	11 010	297	11 111 11111 11111111 00000000

Figure 2 : Code used by LZP1 and LZP2 to write the match length

This method has been compared to using a Huffman-based scheme and found to have nearly identical performance; the Huffman scheme only works significantly better on files which vary from the distribution implied by this method (such as 'trans' and 'geo').

LZP2 is identical to LZP1 except that Static Huffman is used to code the literals. Byte-aligned output is abandoned. 'Static Huffman' is meant to refer to the two-pass method originally proposed by D.A. Huffman [5], in which the frequency counts of the symbols are first computed, and then a code set is generated and transmitted for use on the entire data set.

LZP3 is a significant step beyond the prior versions. The match flags and match lengths are both encoded using order-1 Huffman context coding. The context for encoding these output fields is the order (length) of the context which was used to find the match. There are only three possibilities: 2,3, or 4, so the order-1 coding is implemented simply by performing Static Huffman encoding on three arrays of data. The match flags are blocked together into four-bit hunks to facilitate Huffman encoding on a binary alphabet. The literals are encoded using order-1-0 Huffman encoding, which is described in [3]. Both LZP2 and LZP3 use sophisticated methods for compression of the Huffman code lengths (which must be transmitted in the two-pass method); this is critical for order-1-0 Huffman.

Following a match, a literal is immediately written without an escape, as is done in the original LZ77 method. This is motivated by the fact that after a match, the context will probably be the matched string; therefore, if there were more data to be matched, it would have been matched in the previous event. Thus, reserving the code-space for a match is wasteful.

LZP4 uses a full-precision arithmetic encoder and several context-modeling techniques to improve compression.

Each context keeps the number of matches and literals written from that context. These are combined to create the ‘CodingContext’ with which subsequent matches and literals are written. The CodingContext is constructed from ((num-matches)*8 + num-literals). Neither num-matches nor num-literals is allowed to exceed seven.

Literal flags are encoded as a match length of zero, to reduce the number of arithmetic encoding operations required (only one for a match, two for a literal). Match flags are implicitly encoded as a non-zero match length.

Match lengths are encoded as the difference from the last match length in that context. The “last match length” is initialized to one. If the current match length is less than the last match length, a literal is written. Whenever a literal is written, “last match length” in the current context is reset to one.

Match lengths are encoded with an order-1-0 model, in which the CodingContext is used as the order-1 context. The model only includes (adjusted) match lengths from 0 to 255; the value of 256 is reserved to indicate the length was 256 or higher (there is no maximum match length). The rest of the length is then written in an expanding flat arithmetic code (an equal probability is assigned to all values).

This method implicitly contains special handling of deterministic contexts, because they are the ones with num-literals equal to zero. This method solves the deterministic context problem : the actual match probability given a certain context, for a certain file, is modelled.

Literals are encoded with order-4-3-2-1-0 PPMC with full exclusion. The character predicted by LZP (i.e. the character following the last occurrence of the context) is initially excluded. A literal is written immediately after a match, as in LZP3.

Results

The four LZP implementations are reported upon here. The compression ratio, in terms of bits per byte, is reported for the files of the Calgary Corpus. Speed, in bytes per second, is also reported, as is approximate memory use in bytes. With each algorithm is reported the performance of its major competitor(s). Speed was measured, without disk accesses, on an Amiga 3000 running at 10 MIPS on a 25 MHz CISC Motorola 68030.

	LZP1	LZP2	LZS[16]	LZRW1[4]	
speed	250-270	80-100	70-85	170-210	(kilobytes per second)
mem use	24k	25k	16k	12k	(bytes)
bib	4.146	3.292	5.247	4.753	(bits per byte)
book1	5.718	4.073	5.849	5.434	
book2	4.563	3.396	5.117	4.717	
geo	6.799	6.087	6.810	6.754	
news	4.777	3.744	5.170	4.907	
obj1	4.861	5.031	4.556	4.931	
obj2	3.765	3.598	3.982	4.100	
paper1	4.526	3.451	5.117	4.627	
paper2	4.890	3.534	5.391	4.880	
pic	1.397	1.323	1.585	2.045	
progc	4.147	3.273	4.533	4.368	
progl	2.956	2.365	3.618	3.496	
progp	2.859	2.311	3.473	3.426	
trans	2.639	2.181	3.972	3.687	
average	**4.146**	**3.404**	**4.601**	**4.438**	

	LZP3	LZFG[13]	PkZip[17]	LZP4	PPMD+[2]	
speed	30	9	10	6	3	(kbyps)
mem use	500k	180k	206k	2000k	2500k	(bytes)
bib	2.408	2.90	2.53	1.915	1.862	(bpb)
book1	3.201	3.62	3.26	2.350	2.303	
book2	2.656	3.05	2.71	2.011	1.963	
geo	5.303	5.70	5.37	4.740	4.733	
news	2.878	3.44	3.08	2.350	2.355	
obj1	4.377	4.03	3.83	3.744	3.728	
obj2	2.870	2.96	2.63	2.388	2.378	
paper1	2.927	3.03	2.79	2.378	2.330	
paper2	2.940	3.16	2.88	2.389	2.315	
pic	0.855	0.87	0.82	0.814	0.795	
progc	2.928	2.89	2.70	2.392	2.363	
progl	1.910	1.97	1.80	1.589	1.677	
progp	2.013	1.90	1.82	1.585	1.696	
trans	1.670	1.76	1.69	1.343	1.467	
average	**2.781**	**2.95**	**2.71**	**2.291**	**2.283**	

Figure 3 : Results for LZP1-4 on the Calgary Corpus

There are many modifications to LZP. For example, if high speed is critical but memory use is not, LZP1 could be given a larger window and larger hash table, which would improve compression without affecting speed. The four implementations presented here were selected because they provide a wide sampling of the functionality of LZP, and these four implementations can together answer most of the needs of data compression applications.

The performance of the LZP algorithm with different hash table sizes and window sizes has been studied in great detail. The compression of the file ‘paper2’ by an LZP implementation similar to LZP2 is here reported for various settings of these two parameters.

	Window Size					
	64k	32k	16k	8k	4k	2k
15	3.470	3.489	3.541	3.630	3.744	3.909
14	3.506	3.521	3.568	3.648	3.758	3.919
13	3.534	3.548	3.590	3.663	3.769	3.926
12	3.634	3.643	3.672	3.728	3.816	3.958
11	3.810	3.814	3.828	3.858	3.916	4.027
10	4.191	4.190	4.191	4.195	4.203	4.239

^
Number of bits in the order-3 Hash table index (H)

Figure 4 : Compression of ’paper2’ by LZP2 with various setting

It can be seen that LZP functions very well under small hash and small window conditions (much better than LZ77, which suffers a 6% compression penalty for each window halving [13]; LZP suffers (approximately) a 2% penalty for each window halving).

With a small hash table, a large window provides very little benefit; similarly, with a small window, a large hash table does little good.

A hash table larger than 15 bits results in almost no improvement in compression (in fact, an anomaly in the hash function makes a 16 bit hash worse than a 15 bit hash).

Conclusions

A new data compression technique has been presented : the LZP algorithm. This algorithm can be viewed as a practical combination of LZ77 and PPM compression methods, which achieves the best features of both. Four implementations of the LZP algorithm were presented : LZP1, which is 20% faster than LZRW1 with a Calgary Corpus average of 4.14 bits per byte (LZRW1 gets 4.43 bpb [4]) and LZP2, which attains a corpus average of 3.40 bits per byte while compressing faster than the Stac LZS algorithm (which gets 4.60 bpb), both LZP1 and LZP2 use approximately 25k bytes of memory. LZP3 achieves compression on the order of PkZip (and all other LZ77 based archivers) with an average of 2.78 bits per byte, while operating at three times the speed of PkZip. LZP4 achieves the highest compression of any algorithm to date, with a corpus average of 2.291 bits per byte, while operating at nearly twice the speed, and using less memory than, the previous state-of-the-art PPM algorithms.

Acknowledgements

Thanks to : Peter Fenwick for extensive assistance with the writing of this paper, to Tim Bell for guiding me through the authoring of my first data compression paper, and to James Scott for making this work possible.

References

1. J.G. Cleary and W.J. Teahan, “Experiments on the zero frequency problem”, Proceedings of the IEEE Data Compression Conference (DCC) 95
2. W.J. Teahan, “Probability estimation for PPM”, available by HTTP at: http://www.cs.waikato.ac.nz/papers/pe_for_ppm/ppm1.ps.gz
3. C. Bloom, “New Techniques in Context Modeling and Arithmetic Encoding”, submitted to DCC96
4. R.N. Williams, “An Extremely Fast Ziv-Lempel Data Compression Algorithm”, DCC91, p. 362
5. D.A. Huffman, “A Method for the Construction of Minimum-Redundancy Codes”, Proc. I.R.E. 40 (9) (1952) p. 1098-1101
6. R.G. Gallager, “Variations on a Theme by Huffman”, IEEE Trans. Information Theory IT-24 (6) (1978) p. 668-674.
7. P.M. Fenwick, “A New Data Structure for Cumulative Frequency Tables”, Software-Practice and Experience 24 (3) (1994) 327-336.
8. T.C. Bell, J.G. Cleary and I.H. Witten, Text Compression (1990) p. 100 - 139
9. D.S. Hirschberg and D.A. Lelewer, “Context Modeling for Text Compression”, in J.A. Storer, Image and Text Compression, (1992) p. 113 - 14
10. D.T. Hoang, P.H. Long, and J.S. Vitter, “Multiple-Dictionary Compression Using Partial Matching”, DCC95 p. 273 - 281
11. Y. Nakano, H. Yahagi, Y. Okada, and S. Yoshida, “Highly efficient Universal Coding with Classifying to Sub-Dictionaries for Text Compression”, DCC94, p. 234 - 243
12. P.C. Gutmann and T.C. Bell, “A Hybrid Approach to Text Compression”, DCC94, p.225 - 233
13. E.R. Fiala and D.H. Greene, “Data Compression with Finite Windows”, Communications of the ACM 32 (4) (1989), p. 490 - 505
14. J.A. Storer and T.G. Szymanski, “Data Compression via Textual Substitution”, Journal of the ACM 29 (4) (1982), p. 928 - 951
15. The Calgary Compression Corpus, available by FTP from: ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus/
16. The Stac LZS algorithm, as described by ANSI draft X3.241-1994
17. The PKWARE Zip algorithm, information available by HTTP from: http://www.pkware.com/product.html
18. J. Ziv and A. Lempel, “A universal algorithm for sequential data compression”, IEEE Trans Information Theory IT-23 (3) (1977), p. 377-343
19. A. Moffat, “Implementing the PPM data compression scheme”, IEEE Trans. Communications COM-38 (11) (1990), p. 1917 - 1920

Appendix: Pseudo C code for a generic LZP implementation

```
The inputs to this routine are:

      ubyte * InPtr;    // the input raw data array
      long InLen;       // the length of this array, in bytes

the variables used are:

      unsigned doublelong C; // 64-bit integer context value
      ubyte * P, * S;        // dictionary reference
      int L;                 // length of match

the routine:

      while(InLen>0)
         {
         S = InPtr;
         C = *((unsigned doublelong *)(S - 8));
         P = GetFromIndex(C);
         AddToIndex(C,S);
         if ( P )
            {
            L = 0;
            while(*S == *P) { S++; P++; L++; }
            if ( L )
               {
               WriteFlag(Match);
               if ( L >InLen ) L = InLen;
               WriteLength(L);
               InLen -= L;
               InPtr += L;
               }
            else
               {
               WriteFlag(NoMatch);
               WriteLiteral(*S);
               InLen--;
               InPtr++;
               }
            }
         else
            {
            WriteLiteral(*S);
            InLen--;
            InPtr++;
            }
         }
```