International Journal of Engineering Research and General Science Volume 4, Issue 1, January-February, 2016
ISSN 2091-2730

DEFLATE COMPRESSION ALGORITHM

Savan Oswal[1], Anjali Singh[2], Kirthi Kumari[3]

B.E Student, Department of Information Technology, KJ'S Trinity College Of Engineering and Research, Pune, India[1,2,3]

Abstract— This specification defines a lossless compressed data format that compresses data using a combination of the LZ77 algorithm and Huffman coding, with efficiency comparable to the best currently available general-purpose compression methods. The data can be produced or consumed, even for an arbitrarily long sequentially presented input data stream, using only an a priori bounded amount of intermediate storage. The format can be implemented readily in a manner not covered by patents.

Keywords— LZ77, Huffman Coding, Deflate, GZip, Zlib, Compression, Decompression, Lossy Compression, Lossless Compression

I. INTRODUCTION

The DEFLATE compressed data format consists of a series of blocks, corresponding to successive blocks of input data. Each block is compressed using a combination of the LZ77 algorithm and Huffman coding . The LZ77 algorithm finds repeated substrings and replaces them with backward references (relative distance offsets). The LZ77 algorithm can use a reference to a duplicated string occurring in the same or previous blocks, up to 32K input bytes back.

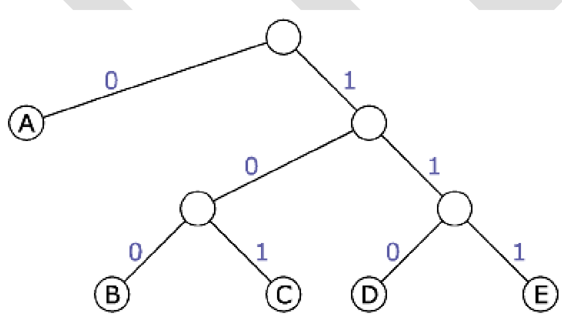
II. PURPOSED SYSTEM

HUFFMAN CODE

The algorithm as described by David Huffman assigns every symbol to a leaf node of a binary code tree. These nodes are weighted by the number of occurences of the corresponding symbol called frequency or cost.

The tree structure results from combining the nodes step-by-step until all of them are embedded in a root tree. The algorithm always combines the two nodes providing the lowest frequency in a bottom up procedure. The new interior nodes gets the sum of frequencies of both child nodes.

Code Tree according to Huffman



The branches of the tree represent the binary values 0 and 1 according to the rules for common prefix-free code trees. The path from the root tree to the corresponding leaf node defines the particluar code word

Huffman codes are part of several data formats as ZIP, GZIP and JPEG. Normally the coding is preceded by procedures adapted to the particular contents. For example the wide-spread DEFLATE algorithm as used in GZIP or ZIP previously processes the dictionary based LZ77 compression.

LEMPEL-ZIV-77 (LZ77)

Developement:

Jacob Ziv and Abraham Lempel had introduced a simple and efficient compression method published in their article "A Universal Algorithm for Sequential Data Compression". This algorithm is referred to as LZ77 in honour to the authors and the publishing date 1977.

Fundamentals:

LZ77 is a dictionary based algorithm that addresses byte sequences from former contents instead of the original data. In general only one coding scheme exists, all data will be coded in the same form:

- Address to already coded contents
- Sequence length
- First deviating symbol

If no identical byte sequence is available from former contents, the address 0, the sequence length 0 and the new symbol will be coded.

Example "abracadabra":

	Addr.	Length	deviating Symbol
abracadabra	0	0	'a'
a bracadabra	0	0	'b'
ab racadabra	0	0	'r'
abr acadabra	3	1	'c'
abrac adabra	2	1	'd'
abracad abra	7	4	''

Because each byte sequence is extended by the first symbol deviating from the former contents, the set of already used symbols will continuously grow. No additional coding scheme is necessary. This allows an easy implementation with minimum requirements to the encoder and decoder.

Restrictions:

To keep runtime and buffering capacity in an acceptable range, the addressing must be limited to a certain maximum. Contents exceeding this range will not be regarded for coding and will not be covered by the size of the addressing pointer.

Compression Efficiency:

The achievable compression rate is only depending on repeating sequences. Other types of redundancy like an unequal probability distribution of the set of symbols cannot be reduced. For that reason the compression of a pure LZ77 implementation is relatively low.

A significant better compression rate can be obtained by combining LZ77 with an additional entropy coding algorithm. An example would be Huffman or Shannon-Fano coding. The wide-spread Deflate compression method (e.g. for GZIP or ZIP) uses Huffman codes for instance.

Compression algorithm (deflate)

The deflation algorithm used by gzip (also zip and zlib) is a variation of LZ77 (Lempel-Ziv 1977, see reference below). It finds duplicated strings in the input data. The second occurrence of a string is replaced by a pointer to the previous string, in the form of a pair (distance,length). Distances are limited to 32K bytes, and lengths are limited to 258 bytes. When a string does not occur anywhere in the previous 32K bytes, it is emitted as a sequence of literal bytes. (In this description, `string' must be taken as an arbitrary sequence of bytes, and is not restricted to printable characters.)

Literals or match lengths are compressed with one Huffman tree, and match distances are compressed with another tree. The trees are stored in a compact form at the start of each block. The blocks can have any size (except that the compressed data for one block must fit in available memory). A block is terminated when deflate() determines that it would be useful to start another block with fresh trees. (This is somewhat similar to the behavior of LZW-based _compress_.)

Duplicated strings are found using a hash table. All input strings of length 3 are inserted in the hash table. A hash index is computed for the next 3 bytes. If the hash chain for this index is not empty, all strings in the chain are compared with the current input string, and the longest match is selected.

The hash chains are searched starting with the most recent strings, to favor small distances and thus take advantage of the Huffman encoding. The hash chains are singly linked. There are no deletions from the hash chains, the algorithm simply discards matches that are too old.

To avoid a worst-case situation, very long hash chains are arbitrarily truncated at a certain length, determined by a runtime option (level parameter of deflateInit). So deflate() does not always find the longest possible match but generally finds a match which is long enough.

Deflate() also defers the selection of matches with a lazy evaluation mechanism. After a match of length N has been found, deflate() searches for a longer match at the next input byte. If a longer match is found, the previous match is truncated to a length of one (thus producing a single literal byte) and the process of lazy evaluation begins again. Otherwise, the original match is kept, and the next match search is attempted only N steps later.

The lazy match evaluation is also subject to a runtime parameter. If the current match is long enough, deflate() reduces the search for a longer match, thus speeding up the whole process. If compression ratio is more important than speed, deflate() attempts a complete second search even if the first match is already long enough.

The lazy match evaluation is not performed for the fastest compression modes (level parameter 1 to 3). For these fast modes, new strings are inserted in the hash table only when no match was found, or when the match is not too long. This degrades the compression ratio but saves time since there are both fewer insertions and fewer searches.

Decompression algorithm (inflate)

The real question is, given a Huffman tree, how to decode fast. The most important realization is that shorter codes are much more common than longer codes, so pay attention to decoding the short codes fast, and let the long codes take longer to decode.

inflate() sets up a first level table that covers some number of bits of input less than the length of longest code. It gets that many bits from the stream, and looks it up in the table. The table will tell if the next code is that many bits or less and how many, and if it is, it will tell the value, else it will point to the next level table for which inflate() grabs more bits and tries to decode a longer code.

How many bits to make the first lookup is a trade off between the time it takes to decode and the time it takes to build the table. If building the table took no time (and if you had infinite memory), then there would only be a first level table to cover all the way to the longest code. However, building the table ends up taking a lot longer for more bits since short codes are replicated many times in such a table. What inflate() does is simply to make the number of bits in the first table a variable, and set it for the maximum speed.

inflate() sends new trees relatively often, so it is possibly set for a smaller first level table than an application that has only one tree for all the data. For inflate, which has 286 possible codes for the literal/length tree, the size of the first table is nine bits. Also the distance trees have 30 possible values, and the size of the first table is six bits. Note that for each of those cases, the table ended up one bit longer than the ``average'' code length, i.e. the code length of an approximately flat code which would be a little more than eight bits for 286 symbols and a little less than five bits for 30 symbols. It would be interesting to see if optimizing the first level table for other applications gave values within a bit or two of the flat code size.

Ok, you want to know what this cleverly obfuscated inflate tree actually looks like. You are correct that it's not a Huffman tree. It is simply a lookup table for the first, let's say, nine bits of a Huffman symbol. The symbol could be as short as one bit or as long as 15 bits. If a particular symbol is shorter than nine bits, then that symbol's translation is duplicated in all those entries that start with that symbol's bits. For example, if the symbol is four bits, then it's duplicated 32 times in a nine-bit table. If a symbol is nine bits long, it appears in the table once.

If the symbol is longer than nine bits, then that entry in the table points to another similar table for the remaining bits. Again, there are duplicated entries as needed. The idea is that most of the time the symbol will be short and there will only be one table look up. (That's whole idea behind data compression in the first place.) For the less frequent long symbols, there will be two lookups. If you had a compression method with really long symbols, you could have as many levels of lookups as is efficient. For inflate, two is enough.

So a table entry either points to another table (in which case nine bits in the above example are gobbled), or it contains the translation for the symbol and the number of bits to gobble. Then you start again with the next ungobbled bit.

You may wonder: why not just have one lookup table for how ever many bits the longest symbol is? The reason is that if you do that, you end up spending more time filling in duplicate symbol entries than you do actually decoding. At least for deflate's output that generates new trees every several 10's of kbytes. You can imagine that filling in a 2^15 entry table for a 15-bit code would take too long if you're only decoding several thousand symbols. At the other extreme, you could make a new table for every bit in the code. In fact,that's essentially a Huffman tree. But then you spend two much time traversing the tree while decoding, even for short symbols.

So the number of bits for the first lookup table is a trade of the time to fill out the table vs. the time spent looking at the second level and above of the table.

Here is an example, scaled down:

The code being decoded, with 10 symbols, from 1 to 6 bits long:

A: 0
B: 10
C: 1100
D: 11010
E: 11011
F: 11100
G: 11101
H: 11110
I: 111110
J: 111111

Let's make the first table three bits long (eight entries):

000: A,1
001: A,1
010: A,1
011: A,1

100: B,2
101: B,2
110: -> table X (gobble 3 bits)
111: -> table Y (gobble 3 bits)

Each entry is what the bits decode to and how many bits that is, i.e. how many bits to gobble. Or the entry points to another table, with the number of bits to gobble implicit in the size of the table.

Table X is two bits long since the longest code starting with 110 is five bits long:

00: C,1
01: C,1
10: D,2
11: E,2

Table Y is three bits long since the longest code starting with 111 is six bits long:

000: F,2
001: F,2
010: G,2
011: G,2
100: H,2
101: H,2
110: I,3
111: J,3

So what we have here are three tables with a total of 20 entries that had to be constructed. That's compared to 64 entries for a single table. Or compared to 16 entries for a Huffman tree (six two entry tables and one four entry table). Assuming that the code ideally represents the probability of the symbols, it takes on the average 1.25 lookups per symbol. That's compared to one lookup for the single table, or 1.66 lookups per symbol for the Huffman tree.

There, I think that gives you a picture of what's going on. For inflate, the meaning of a particular symbol is often more than just a letter. It can be a byte (a "literal"), or it can be either a length or a distance which indicates a base value and a number of bits to fetch after the code that is added to the base value. Or it might be the special end-of-block code. The data structures created in inftrees.c try to encode all that information compactly in the tables.

III. COMPARITIVE STUDY

A comparative study was performed between Deflate compression algorithms to that of Lempel-Ziv-Welch (LZW) data compression algorithm and the results showed that the deflate algorithm is efficient in both compression rate as well as the speed at which the compression is done.

Table 1: Compression techniques comparison

Data File Size (bytes)	Compression using		Compression Ratio using	
	Deflate (bytes)	LZW (bytes)	Deflate (%)	LZW (%)
17768	723	14740	95.93	17.04
53298	1135	29416	97.87	44.81
94038	24771	89956	73.66	4.34
99734	1637	43064	98.36	56.82
120054	1756	4780	98.54	96.02
186658	47403	156336	74.60	16.24

Below shown graph illustrates the comparison of compression ratio of Deflate algorithm to that of LZW algorithm.

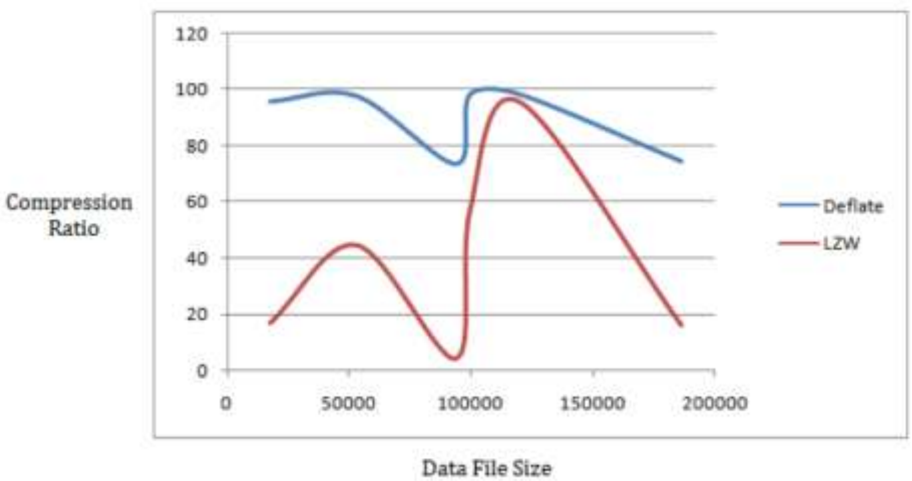


FIG 5: COMPARISON GRAPH

X-AXIS -> DATA FILES SIZE

Y-AXIS -> COMPRESSION RATIO

IV. ADVANTAGES

Compression of files offer many advantages. When compressed, the quantity of bits used to store the information is reduced. Files that are smaller in size will result in shorter transmission times when they are transferred on the Internet. Compressed files also take up less storage space. File compression can zip up several small files into a single file for more convenient email transmission. As compression is a mathematically intense process, it may be a time consuming process, especially when there is a large number of files involved. Some compression algorithms also offer varying levels of compression, with the higher levels achieving a smaller file size but taking up an even longer amount of compression time. It is a system intensive process that takes up valuable resources that can sometimes result in “Out of Memory” errors. With so many compression algorithm variants, a user downloading a compressed file may not have the necessary program to un-compress it.

V. CONCLUSION

In conclusion, data compression is very important in the computing world and it is commonly used by many applications, including the suite of SyncBack programs. In providing a brief overview on how compression works in general it is hoped this article allows users of data compression to weigh the advantages and disadvantages when working with it.

REFERENCES:

[1] Sharma, M.: 'Compression Using Huffman Coding'.International Journal of Computer Science and Network Security, VOL.10 No.5, May 2010.
[2] Huffman, D. A., "A Method for the Construction of Minimum Redundancy Codes", Proceedings of the Institute of Radio Engineers, September 1952, Volume 40, Number 9, pp. 1098-1101.
[3] Ziv J., Lempel A., "A Universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory, Vol. 23, No. 3, pp. 337-343.
[4] Howard, P.G.; Vitter, J.S. Arithmetic coding for data , compression Proceedings of the IEEE, Volume: 82 , , Issue: 6 , June 1994
[5] Liu Bin, Tian Jinwen, Zhang Tianxu, An Image Lossless Compression Algorithm Based on Integer Haar Wavelet Transform and DPCM, Journal of Huazhong University of Science & Technology, Sep. 1999.
[6] P. Deutsch, DEFLATE Compressed Data Format, Aladdin Enterprises Category: Informational May 1996.
[7] Huffman, D. A., “A Method for the Construction of Minimum Redundancy Codes”, Proceedings of the Institute of Radio Engineers, September 1952, Volume 40, Number 9, pp. 1098-1101.
[8] Jagadish H. Pujar, Lohit M. Kadlaskar. “A New Lossless Method Of Image Compression And Decompression Using Huffman Coding Techniques”, Journal Of Theoretical And Applied Information Technology, Vol. 2, No. 3, Pp 18-22.
[9] Detlev Marpe, Gabi Blättermann, Jens Ricke, and Peter Maab. “A Two-Layered Wavelet-Based Algorithm for Efficient Lossless and Lossy Image Compression”, IEEE Transactions On Circuits And Systems For Video Technology, Vol. 10, No. 7, Pp 1094-1102, 2000.
[10] R M Capocelli, R Giancarlo and I J Taneja, “Bounds on the redundancy of Huffman codes”, IEEE Transactions on Information Theory, IT-32, pp 854-857, 1986.
[11] A Turpin and A Moffat, “Housekeeping for Prefix Coding”, IEEE Transactions on Communications, 48, pp.622- 628, 2000.
[12] D. Huffman, “A method for the construction of minimum redundancy codes”, Proc. Of the Institute of Radio Engineers, Vol 40, pp 1098-1101, 1952