Analysis of Google's Brotli Compression Algorithm

Jason L. Bogle
Graduate Student
Department of Computer Science
Middle Tennessee State University
jlb2ch@mtmail.mtsu.edu

November 22, 2016

1 Introduction

People spend a large amount of time viewing websites everyday and we don't like to wait for pages to load. Every second we wait is wasted. There must be a way to speed up this process. Maybe if the is compressed on the server and the browser decompress it, the page won't take as long to load. This idea is called HTTP compression [9]. This paper will compare the Brotli compression algorithm developed by Google [14] to the two main algoithms currently used in HTTP compression.

2 The Problem: Slow Loading Websites

Some web pages are large, clunky, and just take too long to load. In fact, a 2009 article by a couple of Google engineers claimed that "every day, more than 99 human years are wasted because of uncompressed content" [10]. That same article states that uncompressed content takes 25% longer to load compared to compressed content, on average. That's a lot of wasted time.

3 The Solution: HTTP Compression

The solution to a faster loading web page is HTTP compression. The content is compressed on the server and sent to the browser where it is uncompressed. This helps reduce the size of each HTTP request, meaning there is less to send from the server to the client (the web browser) [9].

What should be compressed? Anything that is not already compressed natively. Many image files are already compressed, and so are PDF and zip files among others. However, most text-based files are not natively compressed. HTML, XML, CSS, JavaScript, and JSON files are not natively compressed [9].

Just like anything else, there are pros and cons to compressing data. The biggest pro is obviously smaller files to send over the Internet, which reduces loading time. However, this comes at the cost of cpu time on the server. As a result, it is best to not compress small files that won't result in large savings. Compressing large files can result in great space savings, especially if the file contains a lot of whitspace, such as a large CSS or HTML file [9]. The next section looks at the two algorithms currently used the most, as well as a relatively new algorithm by Google.

4 Prerequisites

Each of these compression algorithm use other algorithms as part of themselves. These smaller algorithms are described here.

4.1 Huffman Coding

The Huffman code is a prefix code. It is created by generating a tree based on how frequent each character appears in the data. In a typical text document, much of the ASCII character set is not used, so the tree can be small. The tree is used to determine the codes for each character. This results in characters that are used more frequently having shorter codes. When traversing the tree, branching to the left is a 0 and branching to the right is a 1. It also means that characters not used have no code, which allows for potentially smaller codes for all characters used. Once every character in the data is replaced with the new Huffman code that corresponds to it, the data should take up less space. The process is described in [7]. Algorithm 1 shows a pseudocode for the algorithm.

Algorithm 1 Huffman Coding Pseudocode

```
Input: msg: the message being compressed
Output: outmsg: the compressed message; the Huffman Code tree
 1: procedure Huffman(msg, outmsg)
 2:     f := empty forest to be filled with single-node trees
 3:     for character in msg do                ▷ count the frequency of each character
 4:         if character in f then
 5:             f.character.count++                   ▷ increment character count
 6:         else
 7:             f.character := new single-node tree   ▷ add new character to the forest
 8:             f.character.count = 1
 9:     while $f.size > 1$ do                  ▷ while there is more than 1 tree in the forest
10:         node := new empty node
11:         min := the node in f with the lowest count
12:         f.remove(min)
13:         node.left := min
14:         min := the node in f with the lowest count
15:         f.remove(min)
16:         node.right := min
17:         node.count := node.left.count + node.right.count
18:         f.append(node)                          ▷ put new node back in f
19:     outmsg := “”
20:     for character in msg do
21:         add the huffman code for character to outmsg as bits
22:     return the Huffman Code tree
```

Let's analyze the time complexity of this algorithm. Let's say that *msg* is n characters long. First, the algorithm cycles through each character to build the forest of single-node trees with the frequency of each character. Let k be the number of unique characters in *msg*. Now the algorithm builds the Huffman code tree. This takes k iterations. Next, the algorithm goes through every character again and puts the corresponding Huffman code in *outmsg*, which takes n iterations. So, equation 1 represents the number of actions that have to take place. It simplifies to equation 2 and Big-O notation is found in equation 3.

$$T(n) = n + k + n \tag{1}$$

$$T(n) = 2n + k \tag{2}$$

$$O(n) = n \tag{3}$$

Decompressing a file that has been compressed via Huffman coding is pretty simple. The algorithm must rebuild the tree from information at the beginning of the file. Then it goes through the rest of the file bit-by-bit as it replaces the codes from the tree with the real 8-bit codes for the characters. The pseudocode is in Algorithm 2.

Algorithm 2 Huffman Decompression Pseudocode

```
Input: $msg$: the message being decompressed
Output: $outmsg$: the decompressed message
 1: procedure HuffmanDecode($msg, outmsg$)
 2:     construct the tree
 3:     $currnode := root$
 4:     for bit in $msg$ do
 5:         if $currnode-> left! = null$ then
 6:             $outmnsg+ = currnode-> data$        ▷ add the char from $currnode$ to the output
 7:             $currnode := root$
 8:         if bit is 0 then
 9:             $currnode := currnode-> left$        ▷ go left in the tree
10:         else if bit is 1 then
11:             $currnode := currnode-> right$       ▷ go right in the tree
```

4.2 LZ77

LZ77 compression works by finding repeating sequences of data. A “sliding window” is used in finding these repetitive sequences. When a repeating sequence is found, it is replaced with two numbers. One number denotes how far back the original string begins. The other number defines how long the sequence is. A common size for the dictionary buffer (the buffer where the algorithm checks for matches) is 4096 bits. This is because it allows the offset to be anywhere from 0 to 4095, which can be encoded with just 12 bits. If a WORD (16 bits) is used to encode the offset and the length, then that leaves 4 bits for the length. This means the length can only be 0 to 15. However, we only save space if the length of the matched string is 3 or more characters. As a result, we can simply add 3 to the length when decoding. This allows the length to be from 3 to 18 and the look-ahead buffer should only be 18 characters [6].

When implementing the algorithm, it appears to be common practice to search the sliding from right to left, meaning the offset starts at 1, not 4095. This seems to really help when compressing something like HTML where the tags are often quite close to one another. This algorithm is also described very well in [7]. Algorithm 3 shows a pseudocode for it.

Figuring out time complexity for LZ77 is a little more difficult than it is for Huffman Coding. Let’s start with the best case, which would mean *msg* is a string that simply repeats eighteen characters over and over. In this scenario, *cp* would only look at each character once. This means it would only take n comparisons if n is the length of *msg*, as seen in equation 4. Worst case would

Algorithm 3 LZ77 Pseudocode

Input: *msg*: the message being compressed

Output: *outmsg*: the compressed message

```
 1: procedure LZ7($msg, outmsg$)
 2:     $minmatch := 3$                              ▷ the smallest string size match possible
 3:     $maxmatch := 18$                             ▷ the largest string size match possible
 4:     $dictsize := 4096$                           ▷ size of dictionary buffer
 5:     $dp$ := point to first character             ▷ the pointer to the start of the dictionary
 6:     $cp$ := point to first character             ▷ the pointer to the current character
 7:     while $msg[cp]$ is not the end of $msg$ do
 8:         $bestlen := 0$                           ▷ the length of the longest matched string
 9:         $offset := 0$                            ▷ the offset of the longest matched string
10:         $curroffset := 1$                        ▷ the current offset to search from
11:         while $cp - curroffset != dp$ AND $bestlen < maxmatch$ do
12:             $i := 0$
13:             while $i < maxmatch$ AND $msg[cp - curroffset + i] = msg[cp + i]$ do
14:                 $i{+}{+}$
15:             if $i < bestlen$ then
16:                 $bestlen := i$
17:                 $offset := curroffset$
18:             $curroffset - -$
19:         if $bestlen > minmatch$ then
20:             mark as encoded                      ▷ done by adding a 0-bit to $outmsg$
21:             add $< offset, length >$ pair to $outmsg$
22:         else
23:             mark as not encoded                  ▷ done by adding a 1-bit to $outmsg$
24:             add character to $outmsg$
25:     append the number of bits used to the front of $outmsg$
26:     return $outmsg$
```

mean *msg* is a string that never repeats a string longer than two characters. In this scenario, *cp* would look at each character thrice for every character in the dictionary, which is 4096 characters long, except for the first two and last two characters. The first and last characters would only be compared once per character in the dictionary, and the second and second to last characters would each be compared twice per character in the dictionary. This is because the algorithm requires a match of length three or greater, and the algorithm simply moves to the next character if the length is less than 3. As a result, equation 5 shows the total number of comparisons. The Big-O for both cases is shown in equation 6.

$$B(n) = n \tag{4}$$

$$W(n) \approx n * 3 * 4096 - (6 * 4096) \tag{5}$$

$$O(n) = n \tag{6}$$

Decompressing a file that was compressed with LZ77 is also simple. The algorithm simply has to go bit-by-bit and do the appropriate operation. It reads the bit that tells it whether the next few bits are encoded or not. If they are not encoded, then the algorithm reads the next eight bits as a 1-byte character and adds it to the output. However, if they are encoded, then it expands the $< offset, length >$ pair by going back $offset$ characters in the output and copies $length$ characters from there to the current location in the output.

Algorithm 4 LZ77 Decompression Pseudocode

Input: msg: the message being decompressed

Output: $outmsg$: the decompressed message

```
 1: procedure LZ7($msg, outmsg$)
 2:     $totalbits$ = read the bit count from $msg$
 3:     $bit := 0$
 4:     while $bit < totalbits$ do
 5:         if $msg[bit] = 0$ then
 6:             $bit++$
 7:             expand $< offset, length >$ pair to $outputmsg$
 8:             $bit+ = 16$
 9:         else
10:             $bit++$
11:             copy next 8 bits as a character to $outputmsg$
12:             $bit+ = 16$
13:     return $outmsg$
```

5 The Algorithms

The two incumbent is Deflate [7]. The newcomer is Brotli, by Google [14]. Both are lossless data compression algorithms, meaning no data is lost in the compression and decompression process. This is important because no one wants to view a website that is missing some of the data. In the following subsections, the compression process for each algorithm is explained. The decompression processes are not explained as they are just undoing the compression process and the decompression process for each sub-algorithm is already explained in the paper.

5.1 Deflate

A patent for the Deflate algorithm was filed in 1990 and awarded in 1991 [11], so this algorithm has been around for a while. The Deflate algorithm is created by combining the strategies behind Huffman coding and LZ77 compression [7].

There are two main reasons why Deflate is still used: 1) Support is strong on existing systems 2) Deflate is simple and fast [2]. The Deflate algorithm first performs the LZ77 compression and then the Huffman code algorithm is applied. This is done in blocks instead of performing it on the whole of the content in one go [7].

These two steps are applied to the input in blocks, meaning that the input msg is not all processed at once, but as smaller blocks. There are different strategies for determining the blocks. Some set a limit on the number of literals allowed in a block while others look ahead and try to determine the best place to start a new block on the fly. Algorithm 5 chooses to just break the message into blocks that are 16k bytes. The $combine(arg1, arg2)$ function used in the pseudocode simply appends $arg2$ to $arg1$ and returns it.

The Big-O time complexity is pretty simple for Deflate. The algorithm first runs the msg through $LZ77$ which is $O(n)$, and then runs the result of that through $Huffman$ which is also

Algorithm 5 Deflate Pseudocode

Input: *msg*: the message being compressed

Output: *outmsg*: the compressed message

```
1: procedure Deflate($msg, outmsg$)
2:     $blocksize := 16,000$
3:     $blucknum := 0$
4:     $blockend := min(msg.length, (blocknum + 1) * blocksize)$
5:     $currblock := msg[blocknum * blocksize...blockend]$
6:     while $currblock.length > 0$ do
7:         $outblock1 :=$ empty
8:         $LZ77(currblock, outblock1)$
9:         $outblock2 :=$ empty
10:        $hufftree := Huffman(outblock1, outblock2)$
11:        $outblock := combine(hufftree, outblock2)$
12:        $outblock := combine(outblock2.length, outblock)$
13:        $outblock := combine(hufftree.length, outblock)$
14:        add $outblock$ to $outmsg$
15:        $blocknum{+}{+}$
16:        $blockend := min(msg.length, (blocknum + 1) * blocksize)$
17:        $currblock := msg[blocknum * blocksize...blockend]$
```

$O(n)$. Now, the input for $Huffman$ should be smaller than the original *msg*, so it is actually less than $O(n)$ with respect to the length of the original *msg*. As a result, it is easy to see that Deflate runs in $O(n)$ time.

5.2 Brotli

The Brotli compression algorithm was published by Google in September of 2015 with the results of a study they performed themselves comparing Brotli to other compression algorithms [2]. The study showed that even a raw implementation of Brotli performs well in comparison with existing algorithms.

Brotli is not just a compression algorithm though. Google has also defined a new file format that the algorithm uses, which is defined in [3]. Google claims the new format can provide 20-26% more compression compared to deflate-compatible formats, which is a very common format [14].

Brotli is "a modern variant of the LZ77 algorithm, Huffman coding and 2nd order context modeling." It is claimed to run as fast as Deflate, but with a higher compression ratio, the ratio of the original size to the compressed size [8]. Brotli also uses a static dictionary that consists of "13,504 words or syllables of English, Spanish, Chinese, Hindi, Russian and Arabic, as well as common phrases used in machine readable languages, particularly HTML and JavaScript" [2]. This library helps with faster lookup for common phrases.

Context modeling is one of the features that makes Brotli different. This feature essentially means multiple Huffman code trees can be used for each block of data. Also, how the blocks of data are stored differ from the other algorithms. Each block is a sequence of commands. Each command has three parts. The first part is a word consisting of a pair of numbers <insert, copy>. "Insert" defines how many literals follow the word. "Copy" defines how many literals to copy. The second part of a command is a sequence of literals. The final part defines what is being copied. It can refer to the literals earlier in the command or to a location in the dictionary [12].

There are three alphabets here. One is for the literals. The second is for distance values. And the third is for <insert, copy> pairs. Now, because Brotli cares about context, there are 256 possible Huffman code trees for each of the three alphabets. The algorithm switches between trees based on context. This can really help when the file is made up of different types of characters, such as different languages [12]. The psudocode in Algorithm 6 is very high level. This is because the whole algorithm is quite complex as Google worked hard to optimize it in many ways.

Algorithm 6 Brotli Pseudocode

```
Input: in: the message being compressed, but as a custom BrotliFileIn class that also contains a
    buffer and buffer size and a read method
Output: out: the compressed message, but as a custom BrotliFileOut class that also contains a
    write method; returns true on success and false otherwise.
 1: procedure Brotli($in, out$)
 2:     $final\_block := false$
 3:     load static dictionary
 4:     setup sliding window (called $ringbuffer$)
 5:     create output buffer
 6:     while not $final\_block$ do
 7:         copy one block to $ringbuffer$ from $in$, return number of bytes read to $in\_bytes$
 8:         $final\_block := (in\_blocks = 0$ or $BrotliInIsFinshed(in))$ ▷ did we reach the end of $in$?
 9:         $out\_bytes := 0$
10:         $success := WriteBrotliData(final\_block, out\_bytes, output)$        ▷ do the compression
11:         if not $success$ then
12:             return $false$
13:         if $out\_bytes > 0$ and not $out.write(output, out\_bytes)$ then          ▷ write to file
14:             return $false$
15:     return $true$
```

In algorithm 6, the $WriteBrotliData$ method is where the compression is actually done. That method essentially calls the Brotli versions of $LZ77$ followed by $Huffman$. Of course, the Brotli versions are more complicated. The Brotli version of $LZ77$ includes the static dictionary, so the compressed data can reference previous uncompressed literals or it can reference a word in the static dictionary. The Brotli version of $Huffman$ is where context mapping comes in. It uses the previous two bytes to determine the context, which is done by performing two look-ups in static arrays. It also creates multiple Huffman trees, as previously discussed. Algorithms 7 and 8 are attempts at writing high level pseudocode for the Brotli versions of LZ77 and Huffman.

Again, these are very high level pseudocodes. This is because the code that Google has made available [8] is quite challenging to breakdown.

The time complexity is also hard to determine, however, here I go with an attempt. Brotli breaks the input into blocks and performs both LZ77 and Huffman algorithms on them. Normal LZ77 has $O(n)$ complexity, and so would the Brotli version as there is nothing in it to make it exponential or factorial. The same applies for the Huffman portion. So, each block needs $O(n) + O(n)$ which is $O(n)$ time. This is not a very in depth look at the time complexity. That should come in the final paper. But this does make sense because Google claims that it is as fast as Deflate [8], which I determined to have $O(n)$ complexity.

Algorithm 7 BrotliLZ77 Pseudocode

Input: *inbuffer*: the data to compress; *dict*: the static dictionary
Output: *commands*: an array of commands generated by the algorithm

```
1: procedure BrotliLZ77(inbuffer, dict, commands)
2:     setup sliding window (ringbuffer)
3:     while not at end of inbuffer do
4:         bufMatch := longest match in ringbuffer
5:         dictMatch := longest match in dict
6:         if both matches are below minimum match length then
7:             if current command is NOT for literals then
8:                 create new literal command and add to commands array
9:             add this literal to the command
10:            increment literal count for this command
11:        else if bufmatch is better then
12:            create command and add it to commands array
13:            put the insertlength and distance from bufmatch into the command
14:        else
15:            create command and add it to commands array
16:            put the insertlength and distance from dictmatch into the command
17:        add chars to ringbuffer
18:        advance position in inbuffer
```

Algorithm 8 BrotliHuffman Pseudocode

Input: *commands*: an array of commands generated by the algorithm
Output: *litTrees*:Huffman trees for decoding literals; *backTrees*: Huffman trees for decoding backward references; *distTrees*: trees for decoding distance values; *contexts*: array of context changes

```
1: procedure BrotliHuffman(commands, litTrees, BackTrees, DistTrees, contexts)
2:     create histogram for literals
3:     create histogram for backward references
4:     create histogram for distances
5:     for each cmd in commands do
6:         determine context of cmd
7:         add it to contexts
8:         update appropriate histogram
9:     construct Huffman trees from the histograms ▷ similar to regular Huffman, but a lot more trees
```

6 The Experiment

6.1 Implementations

Each of these algorithms already have readily available C/C++ implementations. I planned to implement Huffman coding, LZ77, and Deflate in Python. I also planned to attempt an implementation of Brotli in Python. However, with everything else this semester, it proved too much. I switched from Python to C++ using Visual Studio 2015. Also, I didn't get my version of Deflate completed. All of the bit-manipulation required to implement these algorithms took me a while to comprehend and set me back a few days. The code for my implementations and the experiment can be found at [5].

I also planned to use other versions of Deflate and Brotli. I originally planned on using the zlib version of Deflate, but I ended up using the 7-Zip version, which can be found at source [1].

The Brotli executable I used can be found at source [4] and was compiled using Visual Studio 2015. It is Brotli version 0.3.

6.2 Methods

In order to compare all four algorithms, my main C++ function was just two nested loops. The outer loop selects the directory the files for compressing were in. The inner loop picked a file from that directory. Each file is compressed and decompressed using each algorithm.

The original file size, compressed file size, and uncompressed file size were all logged along with compression time and decompression time. All of these measurements were logged for each algorithm on each file. The logs are saved to a csv file for easy calculations later. Calculated data was bytes saved, percentage saved, compression ratio (original / compressed), and the compression percentage.

Three directories of files were used. The first one contained files from the Canterbury Corpus as originally planned. However, I removed any files that weren't readable as text files because my implementations of Huffman and LZ77 had trouble with those. The other two directories were full of files from personal projects of mine, which are web-based projects and have some good candidates for compression. There were a total of 29 files.

The experiment was compiled and run on my PC. It is running the developer edition of Windows 10 and has 16GB of RAM with an eight core AMD FX-8320 processor at 3.5GHZ.

6.3 Results

The spreadsheet that contains all of the data can be found at [13]. When I calculated the averages for each algorithm, the first thing I noticed is that my implementation of Huffman Coding is really slow and does not save much space. In fact, it was the worst in both categories. My LZ77 implementation was much faster and achieved better compression than Huffman, but was still worse than Deflate and Brotli in both categories. However, for decompression, there was no contest as LZ77 blew the others away. Brotli and Deflate both achieved much better compression than the other two, and they were faster on average. Brotli achieved slightly better compression, but Deflate was faster. This is shown in Table 1.

However, if we take a closer look at the statistics, we learn that the Huffman and LZ77 algorithms are faster than Brotli and Deflate on small files. We also see that Deflate does not do well compressing very small files, as it ends up making the files bigger. This supports the idea that it is only beneficial to compress larger files before sending them across the Internet. Table 2 shows that files between 4096 and 10000 bytes are much closer in compression time between Deflate and

	Algorithm			
Data	Brotli	Deflate	Huffman	LZ77
Compression Time (microseconds)	210102.48	84473.38	2198031.97	299144.07
Decompression Time (microseconds)	66137.14	72967.21	410799.97	7537.97
Compression Ratio	3.83	2.97	1.49	2.27
Percentage Saved	68.28	50.51	31.72	49.46

Table 1: Averages Across All Files

Brotli. LZ77 blows both of them away in speed on files this size, but it doesn't achieve a very good compression ratio. Brotli clearly has the best compression ratio for files this size.

	Algorithm			
Data	Brotli	Deflate	Huffman	LZ77
Compression Time (microseconds)	87586.50	70984.17	296708.33	15,670.50
Decompression Time (microseconds)	68134.50	71385.83	58918.67	1095.00
Compression Ratio	4.26	3.37	1.53	2.71
Percentage Saved	74.61	68.02	34.66	60.41

Table 2: Averages Across All Files Between 4096 and 10000 Bytes

But if we look at even bigger files, ones that definitely need to be compressed, we see a little bit different story. Table 3 shows averages for files greater than 10000 bytes. Brotli and Deflate are a little closer in compression ratio here, but Deflate is quite a bit faster. We also see that LZ77 has slowed down in the compression time category, in relation to Deflate and Brotli.

	Algorithm			
Data	Brotli	Deflate	Huffman	LZ77
Compression Time (microseconds)	430161.27	97527.82	5536604.82	776073.18
Decompression Time (microseconds)	65944.36	73779.55	1028474.36	18418.55
Compression Ratio	4.93	4.23	1.60	2.68
Percentage Saved	76.63	73.02	37.27	59.20

Table 3: Averages Across All Files Larger Than 10000 Bytes

7 Conclusion

What does this all mean? Can Brotli potentially replace Deflate as a common compression algorithm for HTTP? Potentially, yes. Google is still developing Brotli and may improve it quite a bit in the next couple years. The results here show that Brotli works as good or better than Deflate for small to medium sized files (under 10kb), but Deflate appears better overall for larger files.

One more test that would provide good statistics would be to get a secure server and setup it up to use each algorithm and see how well they do at loading files from a server to a browser. Unfortunately, I do not have access to a server I could do this on, so this is something for further research.

References

[1] 7-zip. http://www.7-zip.org/, Oct. 4 2016.

[2] Alakuijala, J., Kliuchnikov, E., Szabadka, Z., and Vandevenne, L. Comparison of brotli, deflate, zopfli, lzma, lzham and bzip2 compression algorithms. http://www.gstatic.com/b/brotlidocs/brotli-2015-09-22.pdf, Sep 2015.

[3] Alakuijala, J., and Szabadka, Z. Brotlie rfc. https://www.ietf.org/rfc/rfc7932.txt, Jul 2016.

[4] Brotli - text/plain. https://textslashplain.com/2015/09/10/brotli/, Sept. 10 2015.

[5] Code. https://github.com/kicker10BOG/BogleBrotliProject, Nov. 21 2016.

[6] Dipperstain, M. Lzss (lz77) discussion and implementation. http://michael.dipperstein.com/lzss/, Mar. 11 2015.

[7] Feldspar, A. An explanation of the deflate algorithm. http://www.zlib.net/feldspar.html, Apr 2002.

[8] Brotli. https://github.com/google/brotli, Aug 2016.

[9] Hume, D. A. *Fast ASP.Net websites.* Manning Pub., 2013.

[10] Jain, A., and Glasgow, J. Use compression to make the web faster. http://googlecode.blogspot.co.uk/2009/11/use-compression-to-make-web-faster.html, Nov 2009.

[11] Katz, P. String searcher, and compressor using same, Sept. 24 1991. US Patent 5,051,745.

[12] Krasnov, V. Results of experimenting with brotli for dynamic web content. https://blog.cloudflare.com/results-experimenting-brotli/, Oct 2015.

[13] Results. https://docs.google.com/spreadsheets/d/188MPRgumhP-ybG2Y-0_1IxZMRCR-mPUnfmqzcufJ5nE/edit?usp=sharing, Nov. 21 2016.

[14] Szabadka, Z. Introducing brotli: a new compression algorithm for the internet. https://opensource.googleblog.com/2015/09/introducing-brotli-new-compression.html, Sep 2015.