Modified Huffman Code for Bandwidth Optimization Through Lossless Compression

101

Alexander Hansen and Mark C. Lewis

Abstract

In the interest of minimizing bandwidth usage, a modified Huffman code structure is proposed, with an accompanying algorithm, to achieve excellent lossless compression ratios while maintaining a quick compression and decompression process. This is important as the usage of internet bandwidth increases greatly with each passing year, and other existing compression models are either too slow, or not efficient enough. We then implement this data structure and algorithm using English text compression as the data and discuss its application to other data types. We conclude that if this algorithm were to be adopted by browsers and web servers, bandwidth usage could be reduced significantly, resulting in cut costs and a faster internet.

Keywords

Compression · Bandwidth · Internet · Lossless · Space optimization

101.1 Introduction

In 2016, over 96,000 petabytes of data were transferred across networks in the world [1]. This is up from 72,000 petabytes in 2015, and 12 petabytes in 1998. Shaving off just a kilobyte or even a few bytes off of every server response could have a significant impact on total bandwidth when requests are being processed en masse. We propose a quick, lossless compression algorithm that could work on multiple data types and minimize bandwidth usage.

As memory and data storage device prices decrease every year, it is becoming less and less important to be efficient in space complexity. We seek to take advantage of this modern trend by increasing the space complexity of the standard Huffman code in exchange for a smaller compressed file size. The majority of all traffic on the internet is either text, image, audio, or video and these can all be tokenized and used in Huffman encoding schemes. Because of this, our proposed Huffman-based encoding could work with any of these data types. Some extremely useful examples of text-based contexts are Javascript, HTML, and CSS. This solution could deliver compressed web languages very quickly and efficiently, with extremely quick decompression.

101.2 Proposed Solution

101.2.1 Modified Huffman Code

From a high level perspective, the modified version of the Huffman code we present has two major changes. The addition of a variable data context, and the offloading of the Huffman code itself into the implementation specification, so the file does not have to store the code. The addition of a data context has been discussed before, but it has not been as large or offloaded in such a manner as ours is [2]. We also introduce a concept of a compression benefit. That is, the benefit of compressing something big is more than compressing something small.

The prototypical Huffman code encrypts English and uses letters as tokens. The code itself will never be larger than the amount of unique letters in the document. As the size of the code itself is not important in our system, we want to choose a less granular symbol than letters. Sticking with English text, we could choose a word or small phrase. With these less granular symbols, in our case words, we can then construct a much larger tree based on the text. This tree should not be based on one individual file, rather, a corpus based on some

A. Hansen (✉) · M. C. Lewis
Trinity University, San Antonio, TX, USA
e-mail: ahansen2@trinity.edu

S. Latifi (ed.), *Information Technology – New Generations*, Advances in Intelligent Systems and Computing 738,
https://doi.org/10.1007/978-3-319-77028-4_101

subset of the English language as a whole. These subsets should be similar in genre, providing a more accurate tree. As an example, if one were to construct a Huffman code from all papers submitted to a computer science journal and then use that code to compress the next submitted paper, it would probably do a decent job. If that code was then used to compress all tweets made in the past hour, it would not. This context of academic papers is then stored in the compression/decompression program and used to compress similar files in the future.

There is also the consideration of benefits. If the word "a" happens frequently, it will end up towards the top of the Huffman code and be compressed down to a very small size. However, if we compress the word "the" down to that same size, even if "the" is less frequent, we could end up with a smaller file. The benefit of compressing "the" is higher. We calculate the benefit of a symbol as the frequency which it occurs times the size of that symbol. We then use these benefit values instead of the normal frequency value used by a standard Huffman code.

Decoding a file compressed in this manner is very quick [3]. We also considered using different branching factors to minimize the depth of the tree, but this turned out to have no real impact on size.

101.2.2 Implementation Specifications

An encoded file consists of the context identifier, the compressed data, and then any words that were not found in the context at the end, uncompressed. This is necessary because, as the contexts are not generated based on the input file itself, it is possible for an input file to have symbols in it that are not in the context. There will be a node in the Huffman code that is in the least likely position, a very far leaf, that will be a placeholder. The uncontained words are then stored in their original form at the end of the file, in respective to the placeholders. A malicious file could circumvent all available contexts and contain a lot of words that no context contained and cause a file expand instead of compressing. This would be extremely unlikely and easily preventable.

The compression and decompression program would be a Huffman code traverser with all of the contexts loaded into it.

101.2.3 Advantages and Disadvantages

The solution to the problem of bandwidth optimization is the primary advantage. This algorithm can cut down on file sizes in transmission by immense amounts. This algorithm also can be implemented easily, without much more difficulty than a normal Huffman code. This is important, as the adoption of this algorithm would require browsers to implement decoders and servers to implement encoders. It is also adaptable to many formats and can provide many contexts, allowing for the context to fit the input data very well.

The size of the program that compresses and decompresses could get large as it must hold all contexts, but this is not a large concern as the implementer could mitigate this with other forms of compression or only hold the necessary contexts. Corner cases do exist that could cause bad compression rates, but are unlikely.

Table 101.1 Compression results in various contexts

Wikipedia	Itself	Similar data
With benefits	80.0	63.3
Without benefits	79.4	60.0

Table 101.2 Compression results in various contexts

Social media	Itself	Similar data
With benefits	78.2	79.3
Without benefits	72.3	70.0

Table 101.3 Compression results in various contexts

News article	Itself	Similar data
With benefits	83.9	62.1
Without benefits	81.2	60.1

101.2.4 Results

We did implement this program and achieve good compression results (Tables 101.1, 101.2 and 101.3).

In all examples, the compression ratios are better when the benefits of compressing a symbol are calculated. We can postulate the reasons for these numbers based on the inherent properties of the text, perhaps their likelihood to use the same word more often, or longer words, as well.

101.3 Conclusion

We hope to promote the adoption of this technique by implementing a more fully featured program, with the previously mentioned optimizations for selecting the proper context, handling frequent short phrases as one token, and providing resources such as browser plugins to begin using it. We also hope to implement tokenizers for other forms of audio such as video, audio, and images, so that it can be compressed as well.

References

1. Cisco Systems. *Visual Networking Index*. Available at: https://www.cisco.com/c/en/us/solutions/service-provider/visual-networking-index-vni/index.html, Accessed 10 Mar 2018
2. M.J. Weinberger, G. Seroussi, G. Sapiro, LOCO-I: a low complexity, context-based, loss-less image compression algorithm, in *Proceedings Data Compression Conference, 1996*, pp. 140–149 (1996). https://doi.org/10.1109/DCC.1996.488319
3. C. Hong-Chung, W. Yue-Li, L. Yu-Feng, Memory-efficient and fast Huffman decoding algorithm. Inf. Process. Lett. **69**(3), 119–122 (1999). ISSN: 0020–0190. https://doi.org/10.1016/S0020-0190(99)00002-2. http://www.sciencedirect.com/science/article/pii/S0020019099000022