A Faster Decoding Technique for Huffman Codes Using Adjacent Distance Array

Chapter · May 2020

DOI: 10.1007/978-981-15-3607-6_25

3 authors, including:

Mir Lutfur Rahman
North East University Bangladesh
5 PUBLICATIONS **7** CITATIONS
SEE PROFILE

Pranta Sarker
North East University Bangladesh
4 PUBLICATIONS **7** CITATIONS
SEE PROFILE

Chapter 25
A Faster Decoding Technique for Huffman Codes Using Adjacent Distance Array

Mir Lutfur Rahman, Pranta Sarker, and Ahsan Habib

1 Introduction

D. A. Huffman provided a minimum redundancy coding scheme in the area of data compression. It is used in lossless data compression field. It assigns variable-length codes to each input symbol that depends on the occurrence of the symbols. The maximum occurred symbol gets the smallest code and least occurred gets the longest code with a unique prefix for ignoring the ambiguity [1]. A lot of works has been done to improve the space efficiency of Huffman decoding. For Huffman tree, Chung et al. [2] represent a data structure based on an array. This data structure requires a memory size of $2n - 3$, where n is the number of symbols. Chen et al. [3] updated the data structure and reduced the memory size to $3n/2 + (n/2)\log n + 1$. Hashemian [4] provided a $O(d)$ time-decoding algorithm by reducing the effect of sparsity due to single side growth in the Huffman tree. It introduced ordering and clustering to alleviate the requirement of extra memory and reduce the memory from $O\big((n+d)\log_2 n\big)$ bits to $O\big(2^d \log_2 n\big)$ bits. Using the same data structure, Chung et al. [5] reduced the memory requirement much less than Hashemian [4]. Later, Chowdhury et al. [6] represented a new technique where circular leaf nodes were used to determine a Huffman tree which reduced the memory to $3n/2$. In 2004, Hashemian [7] introduced a new technique for designing and decoding Huffman codes by defining condensed Huffman

M. L. Rahman (✉) · P. Sarker
Department of Computer Science and Engineering, Shahjalal University of Science and Technology, Kumargaon, Sylhet 3114, Bangladesh
e-mail: mirlutfur.rahman@gmail.com

P. Sarker
e-mail: prantacse04@gmail.com

A. Habib
Institute of Information and Communication Technology, Shahjalal University of Science and Technology, Kumargaon, Sylhet 3114, Bangladesh
e-mail: ahabib-iict@sust.edu

M. S. Uddin and J. C. Bansal (eds.), *Proceedings of International Joint Conference on Computational Intelligence*, Algorithms for Intelligent Systems,
https://doi.org/10.1007/978-981-15-3607-6_25

table (CHT). It led to the fast decoding of time complexity $O(t)$. The required memory space is $O\big((n+4t)\log_2 n\big)$ bits; where t was the number of distinct code length that could be in the range of 1–n. There were a lot of Huffman-based lossless data compression techniques, some of these performed well in space optimization and some of these were good in decoding performance.

Lee et al. [8] established a Huffman decoding method. This method was specially designed for the MPEG-2 AAC audio. They implemented one-dimensional array data structure based on the numerical exposition of the incoming bit stream and its utilization of the offset-oriented nodes allocation. Their proposed method supplemented the processing efficiency of the standard Huffman decoding realized with the ordinary tree search method. Wang et al. [9] improved the Hashemian [7] decoding performance by introducing a new Huffman decoding algorithm. In a single growing Huffman (SGH) tree, each decoding was accomplished by a constant number of instructions accessing the two memories without any consideration of Huffman codes. And this process made them arrive at the conclusion that the decoding is to reach a nearly constant time. In the research [10], Kavousianos et al. proposed an optimal selective Huffman encoding technique for the test data compression. They compared theoretically and based on a set of conditions showed that their proposed encoding method outperform any other existing encoding technique in compressing test data. Kodituwakku and Amarasinghe [11] compared few Huffman-based lossless data compression techniques for text data. Lin et al. [12] analyzed about the decoding efficiency of a Huffman tree. It used numerical interpretation to implement recursion-based Huffman tree, where it could decode more than one symbol at a time. Thus, it gained more decoding speed. But the major drawback of this technique was large memory requirement and the fact that it performed well for small block sizes. Habib et al. [13, 14] provide the concept of the quaternary tree which actually generated the optimal codeword and balance between the decoding speed and the memory usage for Huffman codes.

The requirement of memory is decreasing while time efficiency is gaining increasing demand. This research focuses on the time efficiency of Huffman decoding. Common decoding algorithms scan the input stream bit by bit and traverse the path from the root to a leaf node. It decodes the symbol when it matches during the traversing of the path. We introduce a new data structure named adjacent distance array with a threshold value to speed up the decoding process. An adjacent distance array is used to store the distance of adjacent symbols and the threshold value is used for identifying the adjacent and reduces long code block of adjacent symbols from the encoded file. The distances can be calculated by observing the symbol positions, for example, in ASCII characters, '*A*' and '*C*' have a distance of 2 [From '*C*' to '*A*' → (–2) and '*A*' to '*C*' → (–2)]. During the decoding process, adjacent distance array is used to decode the adjacent symbols instead of traversing the full path of the Huffman tree.

2 Architecture

This research addressed a new method for encoding and decoding processes using the concept of adjacent distance array. This adjacent distance array is used in the decoding process for optimizing the decoding time. In Huffman-based techniques, the most frequent symbols get the smallest code but least frequent symbols codeword increase sequentially. It can be large and takes much time to read the encoded file in the decoding process. In this method, we used the threshold value to identify the adjacent symbols and the adjacent distance array stores the distance of the all adjacent symbols. For a particular symbol, S_i, the adjacent symbols will be $S_{i+1} + S_{i+2} + \cdots + S_{i+m}$,if the distance between S_i and S_{i+1} is less than or equal to the threshold value T.

$$|S_i - S_{i+1}| \leq T \tag{1}$$

If the symbol, S_i, satisfies the condition, the Huffman-generated code will be stored into the encoded file and adjacent distance array will store the distances for all satisfied adjacent symbols $S_{i+1} + S_{i+2} + \cdots + S_{i+m}$. Whenever the condition is not satisfied, the particular symbol should be treated as a new symbol in the encoded file and this process will be continued for all the next adjacent symbols. In the encoded file, for each particular symbol, there is a separator bit '0' in adjacent distance array. The distances of adjacent distance array are represented by a specific coding scheme. The coding scheme for each distance depends on the threshold value T. Each code will be generated by two types of bit patterns; the first type is 2 bits pattern where first bit is always '1' indicating the start of a code and another is '0' or '1' indicating positive or the negative value of distances. Second type is the binary representation of distances. The binary representation for all distances has the same length of bits; the bits length is equal to the maximum bit representation of threshold value.

From this concept, it can be calculated as:

$$T = 2^x - 1 \tag{2}$$

where x is the maximum number of bits required to represent threshold value T. And for this method, the threshold value always satisfies this equation.

The memory representation for the encoded file will be:

$$\sigma_1 = \sum_{i=1}^{N} (F_i - A_i).C_i \tag{3}$$

where

F_i total frequency or occurence for a symbol
A_i reduced frequency from encoded file into adjacent distance array
C_i code bits from Huffman coding scheme
N total number of symbols in encoded file.

The memory representation for adjacent distance array will be:

$$\sigma_2 = \sum_{i=1}^{M} (A_i.x) \tag{4}$$

where

x number of bits required to represent distance
M total number of adjacent symbol distances

So, total memory is required to store the full message will be:

$$\sigma = \sigma_1 + \sigma_2 + H_T + S_N \tag{5}$$

H_T Huffman tree as header
S_N total number of seperators in adjacent distances.

In our proposed method, during the decoding time, the first symbol is decoded from the encoded file. Then, the next adjacent symbols for that particular symbol will be decoded by computing the distances from the adjacent distance array. Therefore, this process will be continued until it finds out any separator or reaches at the end of the adjacent distance array. There is no need to traverse in Huffman code list for adjacent symbols that actually speed up the decoding process. Assume an encoded text string of length n and an alphabet of k symbols. For every encoded symbol, one has to traverse the tree in order to decode that symbol. The tree contains k nodes on average; it takes $O(\log k)$ node visits to decode a symbol. So the time complexity would be $O(n \log k)$. But according to the proposed method, it does not need to visit all nodes for decoding adjacent symbols rather it can be done using normal arithmetic operation which takes $O(1)$. So the complexity for the proposed method would be $O(((n - a) \log n) + a)$; where a is the code length of total adjacent symbols.

3 Implementation

We have mentioned about the concept of adjacent distance array above, and this technique reduces decoding time by storing adjacent symbols in consideration of threshold value (T). There are two processes to implement this technique.

(a) Encoding process
(b) Decoding process.

3.1 Encoding Process

The proposed method uses adjacent distance array and threshold value for encoding process. After starting the encoding process, it takes a message as input then it sets the threshold value (T) and generates code list (L) using Huffman principle. Hence for each symbol (S_i) from the message, it will store its Huffman code into encoded file and check its adjacent symbol distances with respect to threshold value (T) and also check the code length of Huffman and adjacent distance; whether the adjacent distance code length is minimum or not. If both conditions are satisfied, then it will store the distances code into adjacent distance array. If there is any mismatch in condition, it will treat the adjacent symbol as a new symbol in encoded file and repeat the previous process. After storing all adjacent symbol distances for a symbol from encoded file, it uses a separator bit '0' in adjacent distance array. This process will continue until end of the message. After completing the full process, there will be two coded files; one is encoded file codes and another one is adjacent distance array codes (Fig. 1).

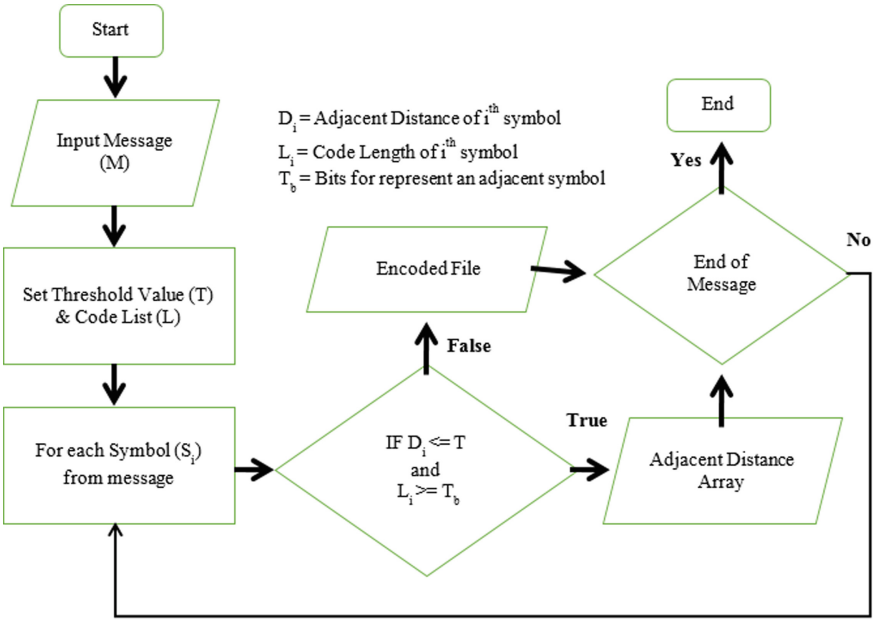

Fig. 1 Flowchart of the encoding process

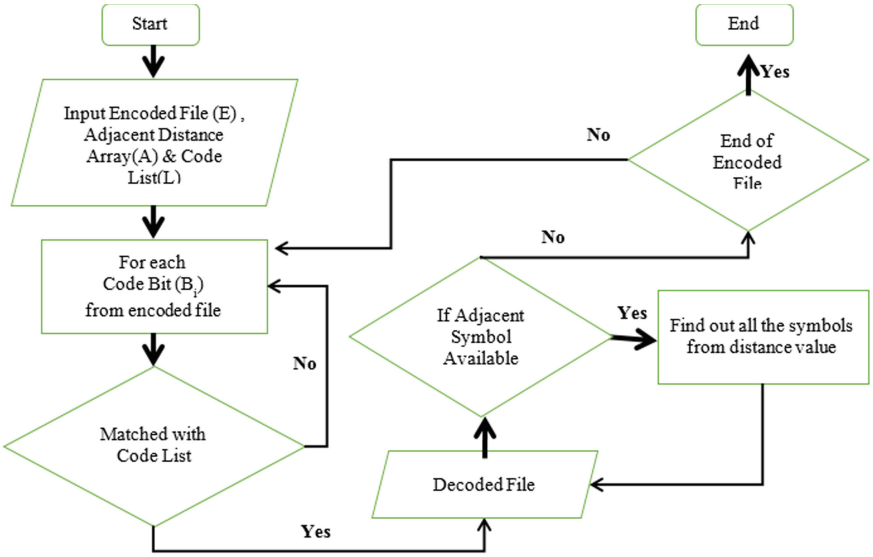

Fig. 2 Flowchart of the decoding process

3.2 Decoding Process

First of all, it takes as input, the encoded file (E), adjacent distance array (A), and code list (L). After that, it will decode a symbol from the encoded file and then, it will check for the adjacent symbols from adjacent distance array. It will decode all the adjacent symbols by computing distances until it gets a separator or reach the end of the adjacent distance array (Fig. 2).

4 Results and Discussions

For testing the performance of the proposed method, it was run on renowned corpus data. The results are compared with Huffman-based methods. The target is to justify enhancement of time with a compression ratio in consideration to Huffman-based methods. The experiment was run on the following environment: OS: Linux (Ubuntu 14.04 LTS) 64-bit, Memory: 7.7GiB, Processor: Intel(R) Core(TM) i5-6500 CPU @ 3.20 GHz × 4, Graphics: Gallium 0.4 on llvmpipe (LLVM 3.4, 256 bits), Language: C++, IDE: CodeBlocks 16.01. Table 1 shows the result of compressed file size and compression–decompression speed on Canterbury Corpus. The Canterbury Corpus [15] is a 1158.08 KB file with 87 distinct characters. The result indicates that the compression performance is highest for Zopfli [17] than Huffman algorithm [18] but compression and decompression speeds are very slow with respect to the proposed method. The proposed method used two types of threshold values, where for $T = 7$

Table 1 Performance comparison for Canterbury Corpus (1158.08 KB)

Method/Algorithm	Space (KB)	Compression–decompression time (s)
Proposed method with ($T = 7$)	788.18	1.449
Proposed method with ($T = 15$)	796.97	1.433
Huffman algorithm	681.48	4.318
Zopfli	418.5	3.431

Table 2 Performance comparison for Brown Corpus (6040.63 KB)

Method/Algorithm	Space (KB)	Compression–decompression time (s)
Proposed method with ($T = 7$)	4037.61	7.064
Proposed method with ($T = 15$)	4093.39	7.037
Huffman algorithm	3470.66	22.353
Zopfli	2230.37	13.574

compression performance was relatively better than the other one $T = 15$. But the compression and decompression performances of $T = 15$ were slower than $T = 7$.

Table 2 shows the result of compressed file size and compression–decompression speed on Brown Corpus [16]. The Brown Corpus is a 6040.63 KB file with 106 distinct characters. This result also indicates that the compression performance is highest for Zopfli then Huffman algorithm but compression and decompression speeds are very slow with respect to the proposed method. The proposed method used two types of threshold values, where for $T = 7$ compression performance was relatively better than the other one $T = 15$. But the compression and decompression performances of $T = 15$ were slower than $T = 7$.

The experimental result shows that the compression performance of the proposed method is relatively better when the threshold value is minimum, it means that the number of adjacent symbols is less. But the encoded file size will be increased due to less number of adjacent symbols. For this reason, it consumes more time. From this observation, we can say that, threshold value can be used to tune the performance. We also observed that the compression performance is not satisfactory compared with others but the compression and decompression times are comparatively faster and in some cases, it is almost half of the other Huffman-based techniques.

5 Conclusion and Future Work

Maintaining the Huffman principle, a new technique has been introduced in this research. We have demonstrated the representation of adjacent distance array as well as the behavior of threshold value. The encoding and decoding processes are also described for the proposed technique. As we are focusing on to improve the

decoding time for Huffman codes, after analyzing the result, it is shown that the adjacent distance array is performing significantly better than any other Huffman-based algorithms by coordinating the threshold value. We have used an extra separator bit for tracking each adjacent symbol; the proposed method takes some extra memory spaces in comparison to other Huffman-based methods. So our future plan is to work on omitting the separator bits to reduce the compression size.

References

1. Huffman DA (1952) A method for the construction of minimum redundancy codes. Proc IRE 40(9):1098–1101
2. Chung KL, Lin YK (1997) A novel memory-efficient Huffman decoding algorithm and its implementation. Sig Process 62(2):207–213
3. Chen HC, Wang YL, Lan YF (1999) A memory efficient and fast Huffman decoding algorithm. Inf Process Lett 69(3):119–122
4. Hashemian R (1995) Memory efficient and high-speed search Huffman coding. IEEE Trans Commun 43(10):2576–2581
5. Lin YK, Chung KL (2000) A space-efficient Huffman decoding algorithm and its parallelism. Theor Comput Sci 246(1–2):227–238
6. Chowdhury RA, Kaykobad M, King L (2002) An efficient decoding technique for Huffman codes. Inf Process Lett 81(6):305–308
7. Hashemian R (2004) Condensed table of Huffman coding, a new approach to efficient decoding. IEEE Trans Commun 52(1):6–8
8. Lee SJ, Jeong HJ, Chang GT (2005) An efficient method of Huffman decoding for MPEG-2 AAC and its performance analysis. IEEE Trans Speech Audio Process 13(6):1206–1209
9. Wang CP, Lee LC, Chang YH (2007) An efficient algorithm of Huffman decoder with nearly constant decoding time. In: CSS 2007 proceedings of the 5th IASTED international conference on circuits, signals and systems. ACM Digital Library, Banff, Alberta, Canada, pp 131–135
10. Kavousianos X, Kalligeros E, Nikolos D (2007) Optimal selective Huffman coding for test-data compression. IEEE Trans Comput 56(8):1146–1152
11. Kodituwakku SR, Amarasinghe US (2013) Comparison of lossless data compression algorithms for text data. Indian J Comput Sci Eng 1(4):416–426
12. Lin YK, Huang SC, Yang CH (2012) A fast algorithm for Huffman decoding based on a recursion Huffman tree. J Syst Softw 85(4):974–980
13. Habib A, Rahman MS (2017) Balancing decoding speed and memory usage for Huffman codes using quaternary tree. Appl Inform 4(1):1–15
14. Habib A, Islam MJ, Rahman MS (2018) Huffman based code generation algorithms: data compression perspectives. J Comput Sci 14(12):1599–1610
15. The Canterbury Corpus. https://corpus.canterbury.ac.nz/resources/cantrbry.zip. Accessed 25 Jan 2019
16. The Brown Corpus. https://ia800306.us.archive.org/21/items/BrownCorpus/brown.zip. Accessed 25 Jan 2019
17. The source code of Zopfli. https://github.com/google/zopfli. Accessed 25 Jan 2019
18. The source code of Huffman algorithm. https://www.geeksforgeeks.org/greedy-algorithms-set-3-hufmancoding/. Accessed 25 Jan 2019