Modification of Adaptive Huffman Coding for use in encoding large alphabets

Mikhail *Tokovarov*[1,*]

[1]Lublin University of Technology, Electrical Engineering and Computer Science Faculty, Institute of Computer Science, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract. The paper presents the modification of Adaptive Huffman Coding method – lossless data compression technique used in data transmission. The modification was related to the process of adding a new character to the coding tree, namely, the author proposes to introduce two special nodes instead of single NYT (not yet transmitted) node as in the classic method. One of the nodes is responsible for indicating the place in the tree a new node is attached to. The other node is used for sending the signal indicating the appearance of a character which is not presented in the tree. The modified method was compared with existing methods of coding in terms of overall data compression ratio and performance. The proposed method may be used for large alphabets i.e. for encoding the whole words instead of separate characters, when new elements are added to the tree comparatively frequently.

1 Introduction

Efficiency and speed – the two issues that the current world of technology is centred at. Information technology (IT) is no exception in this matter. Such an area of IT as social media has become extremely popular and widely used, so that high transmission speed has gained a great importance. One way of obtaining high communication performance is developing more efficient hardware. The other one is to develop the software that would allow to compress the data in such a way that would reduce the size of data but not affect its information content. In other words, to encode the data by the method called *lossless data compression*. This term means that the methods of this type allow the original data to be perfectly reconstructed from the encoded message.

Binary or entropy encoding are the most popular branches among lossless data compression methods. The term *entropy encoding* means that the length of a code is approximately proportional to the negative logarithm of the occurrence of the character encoded with the code. Simplifying it may be said: the higher probability of the character is, the shorter is its code [1].

Several methods of entropy encoding exist, these are the most frequently used methods of this branch:
- arithmetic coding,
- range coding,
- Huffman coding,
- asymmetric Numeral Systems.

Arithmetic and Range coding are quite similar with some differences [2], but arithmetic coding is covered by patent, that is why, due to lack of patent coverage, Huffman coding is frequently chosen for implementing open source projects [3]. The present paper contains the description of the modification that may help to improve the algorithm of adaptive Huffman coding in terms of data savings.

2 Study of related works

Huffman coding has been developed in 1952 by David Huffman. He developed the method during his Sc. D. study if MIT and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes" [1]. Huffman coding is the most optimal among methods encoding symbols separately, but it is not always optimal compared to some other compression methods, such as e.g. arithmetic and Lempel-Ziv-Welch coding [4]. However, the last two methods, as it has been said, are patent-covered, so developers often tend to use Huffman coding [3]. Comparative simplicity and high speed due to lack of arithmetic calculations are the advantages of this method as well. Due to these reasons Huffman coding is often used for developing encoding engines for many applications in various areas [5].

The fact that billions of people are exchanging gigantic amount of data every day stimulated development of compression technologies. This topic constantly presents significant interest for researches, the following works may be presented as the examples:

Jarosław Duda et al. worked out the method called Asymmetric Numeral Systems, the method was developed on the basis of the two encoding methods: arithmetic and Huffman coding. It combined the advantages from the two methods:

* Corresponding author: m.tokovarov@pollub.pl

- near accurate symbol probabilities hence better compression ratio of arithmetic coding, and capability to fast encoding and decoding of Huffman coding [6];
- Facebook Zstandard algorithm is based on LZ77 dictionary coder and tANS – effective entropy encoding based on the Huffman method [7];
- Brotli coding algorithm which is used in most of the modern Internet Browsers, such as Chrome, Opera.

Similarly, to the Zstandard it is based on the combination of LZ77 and modified Huffman coding [8]. So, it may be clearly seen that despite its long history the Huffman encoding algorithm still presents great interest for application.

3 Huffman coding explained

3.1 Static Huffman coding algorithm

The concept of Huffman coding is based on the binary tree. The tree consists of two kinds of nodes: *intermediate nodes*, i.e. the nodes having descendant nodes and the nodes which do not have descendants. These nodes are called *leaves*. A character may be stored only in a leaf node, this condition ensures the character codes to be *prefix-free* [1]. It means, that no character has the code, that would be the initial segment of another character's code. As the example of **prefix** codes, the following bit sequences may be used: 110101 and 110. The code 110 is identical to the initial segment of the code 110101, so these two codes may not be decoded unambiguously. The tree is organized according to the following principles [4]:
a) any node in the tree may not have a single descendant: either two or none;
b) each node in the tree has the number assigned to it. This number is called *weight*. Depending on the type of the node its weight may have the following meanings:
- if the node is a leaf, the weight value is equal to the number of times the character stored in the leaf occurs in the message sent;
- if the node is an intermediate node its weight value is equal to the sum of its descendants' weights.
c) the weight of the right descendant should be not less than the weight of the left descendant.

The codes for every character are defined as the path in the binary tree from the root to the leaf containing the character (See Figure 1), e.g. the blank space character which is the most frequent character in the tree has the code 111, and the 'p' character, which occurs only once has the code 10011 [4].

The Figure 1 presents the tree constructed in accordance with static method of Huffman coding. The main feature of it is that the tree is constructed before the transmission is started on the basis of analysis of the probabilities of separate characters in the whole message [1].

The data compression ratio(DCR) is described by the formula (1):

$$DCR = \frac{numberOfBitsInCompressedMessage}{numberOfBitsInOriginalMessage} \quad (1)$$

But the compression ratio does not show actual space saving as besides of encoded message the table with the code-character pairs should be transmitted. For that reason, the other index should be introduced. The sent-to-original-bits ratio(SOBR) is described in accordance with the formula below:

$$SOBR = \frac{numOfBitsInCompMes + numOfBitsToSendTab}{numberOfBitsInOriginalMessage} \quad (2)$$

The only way to make SOBR equal to DCR is to use one coding tree for all messages, but this tree will not be optimal since the character probabilities may be different in various messages.

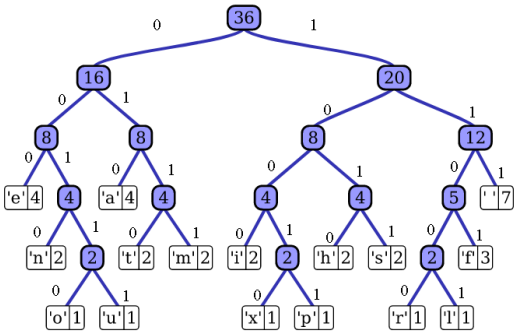

Fig. 1. Huffman tree generated based on the phrase "this is an example of a Huffman tree".

3.2 Adaptive Huffman coding algorithm

The method of adaptive Huffman Coding (AHC) reviewed in the article was proposed by Jeffrey Vitter in his paper published in 1987 [9].

The algorithm working during creating the tree may be observed in Figure 2.

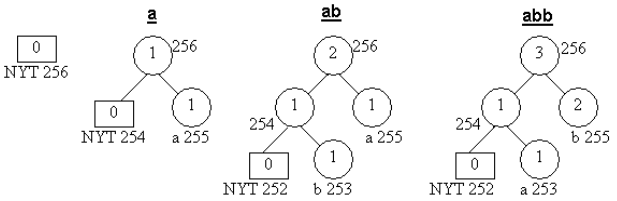

Fig. 2. The algorithm of AHC, example for encoding the word "abb".

This method is based on the same principles as the static method plus the following extensions [9]:
a) every node in the tree has its *key number*, the key numbers are arranged in the following way:
- maximum value of the key number in the tree may be calculated as:

$$keyNum = 2 \cdot maxNumOfCharInAlphabet + 1 \quad (3)$$

- the root has the largest key number;
- an ancestor has larger number than any of its descendants;
- the right descendant should have larger key number than the left descendant;
b) after input of any character the tree is updated;

c) a special leaf node called NYT (not yet transmitted) is used for both indicating the place for a new character and for signalizing that the new character is obtained, its key has the least value in the tree, its weight always equals to zero;
d) a set of nodes with equal weight values is called a block.

The encoding algorithm is described in Listing 1. The algorithm contains only the process of updating the tree and does not consider communication. If the communication algorithm based on the AHC is used, the process becomes more complicated. The idea may be presented as follows:
- transmitter and receiver have identical trees which are updated in accordance with the algorithm described in the Listing 1;
- the communication operates in the way presented in the Listing 2.

As it may be seen from the Listing 2, in case of AHC not only the codes of the characters are transmitted. Auxiliary codes such as the code of *NYTNode* and ASCII codes of the new-coming characters are being transmitted as well. The necessity to transmit auxiliary codes negatively influences SOBR causing it to increase. Due to this fact overall data saving decreases. But along with higher degree of complexity compared to static Huffman coding ACH may still present interest as lossless data compression technique after implementing the modifications described in the next chapters.

3.3 Encoding words instead of separate characters

The title of the article contains the phrase "large alphabet". But the chapter is focused at encoding entire words. What is the connection between these two facts? The idea is that in the method proposed, the words are treated as separate characters, so the leaves of the coding tree store not characters, but complete words, so these words are treated as separate characters in a large alphabet. The author does not claim, that this idea belongs to him. This technique is quite well known and was presented in many works [5, 10].

The greatest success may be achieved in the case of applying this method for encoding the words of an *analytic language*. An analytic language is the type of language where grammatical relationships are established by using strict word order, prepositions, postpositions, particles and special auxiliary words that do not have individual meaning and only indicate some grammatical categories. Analytic languages do not have extensive systems of conjugation and declension as *synthetic languages* do [11].

In many cases a word in a synthetic language may have several forms which are treated as individual words by encoding algorithm. This approach would cause the coding tree to be excessively extensive. It is worth to note that the statistics show that around 95% of all common English texts may be covered by 7000 words [11, 13]. The situation becomes even more optimistic when communication in social networks and mass media is considered.

Listing 1. Pseudocode presenting updating tree of AHC.

```
UpdateTree(character ch)
1  if ch is not found in the tree
2     make a newLeafNode as the right descendant of the
      oldNYTNode;
3     newLeafNode's keyValue := oldNYTNode's
      keyValue-1;
4     make a newNYTNode as the left descendant
      of the oldNYTNode
5     newNYTNode's keyValue := oldNYTNode's
      keyValue-2;
6     NYTNode  := newNYTNode;
7  end if
8  activeNode := node containing ch;
9  do
10    find the nodeWithTheLargestKeyNumber
      in the activeNode's block;
11    if the nodeWithTheLargestKeyNumber's keyNumber
      > activeNode's keyNumber
12       swap nodeWithTheLargestKeyNumber
         and activeNode;
13       swap nodeWithTheLargestKeyNumber's
         keyNumber and activeNode's keyNumber;
         //key numbers should not change the place
14    end if
15    activeNode's weight := activeNode's weight+1;
16    activeNode := activeNode's ancestor;
17 while activeNode <> root
```

Listing 2. Communication with the use of data compression based on AHC.

Transmitter	Receiver
Transmit (character *ch*) 1 if *ch* is found in the tree 2 send the code of *ch* bit by bit; 3 else 4 send the code of *NYTNode* bit by bit; 5 send the ASCII code of *ch*; 6 end if 7 UpdateTree(*ch*);	Receiving is performed in the stream, i.e. in infinite loop. Receive() 1 while (true) 2 receive one bit from the transmitter, 3 add received bit to *bitSequence*; 4 if *bitSequence* leads to a leaf node 5 obtain the character *ch* stored in the node; 6 add *ch* to decoded message; 7 UpdateTree(*ch*); 8 set *bitSequence* empty; 9 end if 10 if *bitSequence* leads to *NYTNode* 11 receive 8 bit; 12 convert received 8 bit to character *ch*; 13 add *ch* to decoded message; 14 UpdateTree(*ch*); 15 set *bitSequence* empty; 16 end if 17 end while

To prove the feasibility of the method the term of *information entropy* should be mentioned. This term has several definitions:
- measure of unpredictability of the state;
- expected (mean) value of information contained in a message.

The value of entropy is calculated by the following formula:

Entropy of *i*-th character in a message:

$$I_i = -log_m p_i \quad (4)$$

Average entropy of a message:

$$H = \sum_{i=1}^{n} -p_i log_m p_i = \sum_{i=1}^{n} -p_i log_m I_i \quad (5)$$

where: p_i - probability of *i*-th character in the message; *n* - number of unique characters in the message; *m* - logarithm base, usually taken equal to 2, as binary system is used in computer technics.

Practically it may be stayed, that the average entropy of a message is equal to the least possible average code length of the characters contained in the message [1].

It is well known, that amongst discrete distribution with equal number of states the uniform distribution has the maximum value of entropy [12]. Every state of the uniform distribution has the same probability equal to 1/n, where n is the number of states, which yields the entropy:

$$H_{uniform} = log_2 n \quad (6)$$

To estimate the maximum entropy, the maximum number of words encoded should be defined. It has been decided to take the max number of words equal to 16384, which yields the maximal entropy equals to 14 bits. The max word number is taken because the practical number of stored words may be higher than 7000 because of some capitalized words, abbreviations, mistakes and user-defined words. To roughly estimate the compression ratio, the average length of English word is used, its value approximately equals to 4 letters. In case if ASCII coding is taken into consideration the average length in bits equals to 32. Based on this value the average compression ratio equals to $14/32 \approx 0.43$. This estimation is fairly promising as average compression ratio for separate-character AHC is about 0.55 [4].

As it is stated in the previous chapter, sent-to-original bit ratio tends to be significantly higher than data compression ratio for complete-word AHC. This effect is especially noticeable during the phase of initial building the coding tree, when new words are coming especially frequently. There are two factors that affect SOBR in this case: sending the complete ASCII codes of the new-coming words and sending the NYT bit sequence.

Precoded dictionaries may be used to decrease the influence of first factor. This possibility is not considered in the current paper.

However, the second factor, i.e. sending the NYT bit sequence will be optimized within the frame of this research. As it is described in the chapter 2 the NYT is used for both indicating the place in the tree the new-coming word is attached to and for sending the signal meaning that the ASCII code of the new-coming signal is going to be sent. This fact provides the opportunity for optimising the algorithm.

4 Modification

4.1 Introduction of NCW node

The NYT node should only act as the pointer for the new-coming word. Its weight still should be 0 and its key number should have the least value in the tree. The new node NCW should be introduced to the tree. Its initial weight should be equal 0. This node should be used for sending the "new-coming word" signal. The introduced NCW node should be treated as a usual leaf node, i.e. after sending the bit sequence corresponding to the NCW node, the procedure described in the lines 8-17 of the Listing 1 should be carried out for the NCW node. The Listing 3 presents the modified algorithm.

Listing 3. Modified algorithm.

Transmitter	Receiver
Transmit (character *ch*) 1 if *ch* is found in the tree 2 send the code of *ch* bit by bit; 3 else **4 send the code of *NCWNode* bit by bit;** 5 **updateTree(*NCWNode*);** 6 send the ASCII code of *ch*; 7 end if 8 UpdateTree(*ch*);	Receiving is performed in the stream, i.e. in infinite loop. Receive() 1 while (true) 2 receive one bit from the transmitter, 3 add received bit to *bitSequence*; 4 if *bitSequence* leads to a leaf node 5 obtain the character *ch* stored in the node; 6 add *ch* to decoded message; 7 UpdateTree(*ch*); 8 set *bitSequenc* empty; 9 end if 10 **if *bitSequence* leads to *NCWNode*** 11**updateTree(*NCWNode*);** 12 receive 8 bit; 13 convert received 8 bit to character *ch*; 14 add *ch* to decoded message; 15 UpdateTree(*ch*); 16 set *bitSequence* empty; 17 end if 18 end while

The Figure 3 presents the difference between modified and non-modified method. The phrase "A friend in need is a friend indeed" was used for the test. As it may be noticed the weight of the NCW node is equal to the number of the unique leaf nodes.

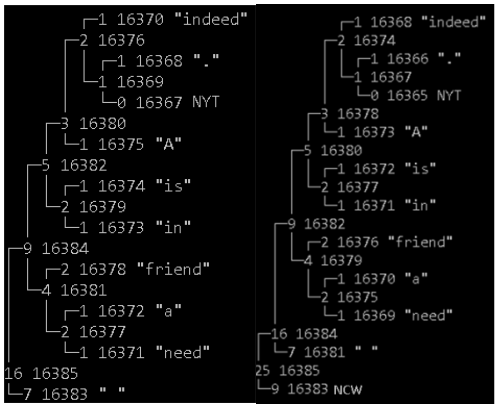

Fig. 3. Comparison of full-word AHC trees. Left – two-purpose NYT node, right – separate NCW and NYT nodes.

4.2 Forgetting

It is a quite frequent fact in real social media application, when a user makes some mistakes in the text. In the case of the complete-word Huffman algorithm it would cause the coding tree to be overgrown and not optimal, since these mistyped words are used very rarely, but keep the place in the tree, causing the entropy to be larger. The same thing may be stated about rare words. Another problem that may raise is overflowing of dynamically allocated memory. In the current application dynamically, allocated arrays are used instead of linked lists due to their better performance rates. To deal with these problem, the algorithm of forgetting should be introduced.

This algorithm is based on the estimation of the relevance function of the stored words. The relevance function may be defined as Euclidean norm:

$$WF = \sqrt{weight^2 + agingFactor^2} \qquad (7)$$

where: *agingFactor* is the value characterizing how long cycles ago the word was used last time; *weight* is the number of word's appearances in the text, the more the *weight*, the more often appears the word in the text.

The forgetting function is based on the algorithm presented in Listing 4.

Listing 4. Forgetting function.

```
1 if numberOfStoredWords > thresholdWordNum
2    Sort the array of the leaf nodes basing on WF in
     descending order;
3    while numberOfStoredWords > desiredWordNum
4      delete the last word from the leaf nodes array;
5    end while
6 end if
```

It is reasonable to set *thresholdWordNum* close to the max number of stored words and *desiredWordNum* – approximately 10-15% less to ensure periodical "cleaning" of the tree.

The Figure 4 presents the example of forgetting function operation. The threshold number of words is equal to 16000 and the desired number of words is 15000.

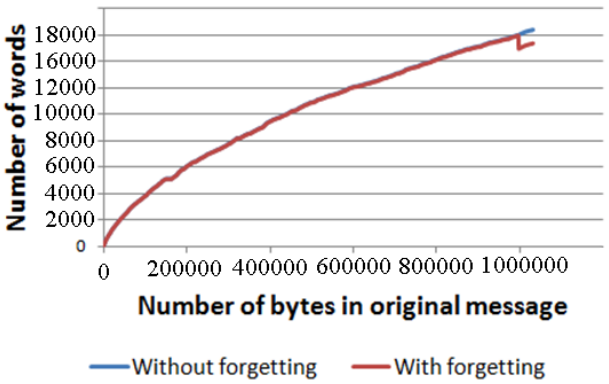

Fig. 4. Example of forgetting function operation.

5 Experiment and results

The tests were conducted on the data set containing more than million characters. The data set was composed on the basis text collected from various sources: news posts, social network comments and private correspondence. Use of forgetting function was completely feasible as many rare words, such as proper names and mistypes appeared. The results have proven that another application of complete-word AHC may be composing a dictionary for dictionary-type coding. Table 1 presents some frequent words and symbols found in the test texts.

Table 1. The most frequent words and symbols used in the text.

Word	Weight	Code
he	937	101001001
have	887	100111001
as	866	100110101
be	866	100110100
are	815	100011101
from	796	100011001
his	793	100011000
has	781	100010101
at	770	100010011
Trump	755	100000000
not	752	1111111110

The next issue that arose during the tests and comparison of the algorithms was the selection of features for analysis. The overall sent-to-original bits ratio has been chosen first. The Figure 5 presents the comparison of SOBR for the three implemented methods: separate-character AHC, unmodified complete-word AHC and modified complete-word AHC.

It may be clearly seen from the picture that even unmodified complete-word AHC demonstrates better sent-to-original bit ratio, as it was stated in the chapter 3.3. The modified method allows to decrease the SOBR even more. The end difference between overall SOBR values for modified and unmodified methods equals to 2.3 %.

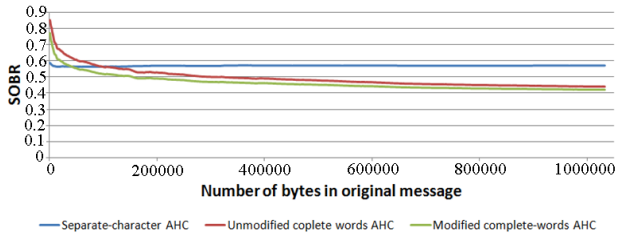

Fig. 5. Comparison of SOBR for the implemented AHC methods.

Another feature that may help to compare the two complete-words methods is specified sent-to-original bits ratio (*SSOBR*). Its value may be computed as first order divided difference of SOBR:

$$SSOBR_i = \frac{NBSM_i - NBSM_{i-1}}{NBOM_i - NBOM_{i-1}} \quad (8)$$

where: *NBSM* is the number of bits in the sent message; *NBOM* is the number of bits in the original message; *i* is the number of step.

The value of the denominator corresponds to the step or the number of bits the specified SOBR is computed per. In the current research the step was chosen to be equal to 1 Kbyte. The Figure 6 presents the differences of SSOBR of unmodified and modified AHC.

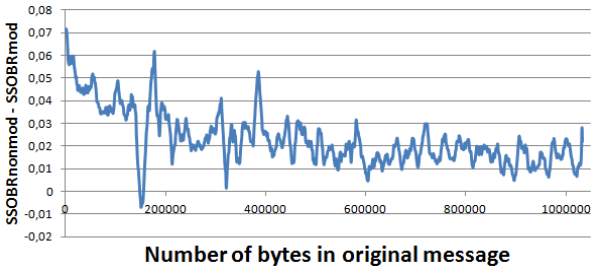

Fig. 6. Differences between SSOBR of unmodified and modified AHC methods.

The Figure 6 shows that the difference between SSOBR of the both methods tends to be more in the periods when a lot of new words is added to the tree (up to 6 percentage point). However, it might have quite low and even negative values, as in case of absence of new words unmodified AHC method becomes more optimal.

6 Conclusion

The comparison of single-character AHC and complete words AHC have been conducted, the research has proven the following:

- overall sent-to-original bits ratio of complete word AHC is approximately 12.8 percentage points lower than separate-words AHC one;
- better SOBR comes at a cost: the algorithm needs more memory to store the tree, the search of the node in the tree is slower due to its size, initial sent-to-original bits ratio is higher than this of separate-character AHC.

The implemented modification allowed to further improve the SOBR of complete-word AHC. The tests have shown that:

- sent to original bits ratio is 15.1 percentage points less compared to separate-character AHC method;
- the proposed modification proved to be more effective during the periods when large number of words is being added to the tree, and less when the number of new words is decreasing. Possible solution, that would be useful in that case, would be using the modified algorithm while the tree is being formed and then deleting the NCW node by the means of the function used for "forgetting" and using unmodified algorithm.

References

1. D.A. Huffman, *Proceedings of I.R.E,* pp. 1098–1101, (1952)
2. G. Nigel N. Martin, *Video & Data Recording Conference*, pp. 612-620, (1979)
3. B. Ryabkot, *Data Compression Coding Conference Proceedings*, pp 246-252, (2004)
4. J. Van Leeuwen, *ICALP*, pp. 382-410, (1976)
5. J. Lee, *MIT Undergraduate Journal of Mathematics*, pp. 122-130, (2007)
6. J. Duda; K. Tahboub; N.J. Gadgil; E.J. Delp, *Picture Coding Symposium (PCS)*, pp. 65-69, (2015)
7. https://code.facebook.com/posts/1658392934479273/smaller-and-faster-data-compression-with-zstandard/ - last access 07.07.2017
8. J. Alakuijala, Z. Szabadka, *Internet Engineering Task Force* (IETF), (2016)
9. J. S. Vitter, *Journal of the ACM*, 34(4), pp 825–845, (1987)
10. M.B. Baer, *2013 IEEE International Symposium on Information Theory Proceedings (ISIT)*, pp. 1749-1753, (2013)
11. V.V. Bochkarev, A.V. Shevlyakova, V.D. Solovyev, Social *Evolution & History*, Vol.**14**, number 2, pp. 153-175 (2015)
12. H. Yokoo, *2016 International Symposium on Information Theory and Its Applications (ISITA)*, (2016)
13. http://www.ravi.io/language-word-lengths - last access 07.07.2017