Journal of Computer Science 2 (12): 885-888, 2006
ISSN 1549-3636
© 2006 Science Publications

# Efficient Data Compression Scheme using Dynamic Huffman Code Applied on Arabic Language

[1]Sameh Ghwanmeh, [2]Riyad Al-Shalabi and [2]Ghassan Kanaan
[1]Director of Computer Center, Yarmouk University, Irbid-Jordan
[2]Arab Academy for Banking and Financial Sciences, Amman-Jordan

**Abstract:** The development of an efficient compression scheme to process the Arabic language represents a difficult task. This paper employs the dynamic Huffman coding on data compression with variable length bit coding, on the Arabic language. Experimental tests have been performed on both Arabic and English text. A comparison is made to measure the efficiency of compressing data results on both Arabic and English text. Also a comparison is made between the compression rate and the size of the file to be compressed. It has been found that as the file size increases, the compression ratio decreases for both Arabic and English text. The experimental results show that the average message length and the efficiency of compression on Arabic text is better than the compression on English text. Also, results show that the main factor which significantly affects compression ratio and average message length is the frequency of the symbols on the text.

**Key words:** Data compression, dynamic Huffman code, Arabic language

## INTRODUCTION

Among numerous types of information produced broadcasted or recorded by electronic information processing systems and tools, the degree of duplication in data is adequately high to allow the output flow to contain considerably fewer bits than the input flow. In conditions, the output flow may contain more bits than the input flow. The real ratio of the numbers of bits is dependent on the uniqueness of the actual input data flow. Compression by this algorithm is lossless, i.e. it is probable to restore exactly the original version of data by earnings of a corresponding decompression algorithm. The algorithm contains features which assist its implementation in data storage and retrieval tools which handles, in a sequential way, data records of varying extent. Compression is used just about everywhere. All the images you get on the web are compressed, typically in the JPEG or GIF formats, most modems use compression, HDTV will be compressed using MPEG-2 and several file systems automatically compress files when stored [1].

The oldest and most widely used codes, ASCII and EBCDIC, are examples of block-block codes, mapping an alphabet of 64 (or 256) single characters onto 6-bit (or 8-bit) codewords. The codes featured in this survey are of the block-variable, variable-variable and variable-block types. For example, in text file processing each character may constitute a message, or messages may be defined to consist of alphanumeric and non-alphanumeric strings[2]. The compression algorithms allow for records of different size and compressibility, along with File Marks, to be efficiently encoded into an output stream in which little or no additional control information is needed to later decode user data[3].

In addition to the categorization of data compression schemes with respect to message and codeword lengths, these methods are classified as either static or dynamic. A static method is one in which the mapping from the set of messages to the set of codewords is fixed before transmission begins, so that a given message is represented by the same codeword every time it appears in the message ensemble. The classic static defined-word scheme is Huffman coding. In Huffman coding, the assignment of codewords to source messages is based on the probabilities with which the source messages appear in the message ensemble. Messages which appear more frequently are represented by short codewords; messages with smaller probabilities map to longer codewords. These probabilities are determined before transmission begins. A code is dynamic if the mapping from the set of messages to the set of codewords changes over time. For example, dynamic Huffman coding involves computing an approximation to the probabilities of occurrence "on the fly", as the ensemble is being transmitted. The assignment of codewords to messages is based on the values of the relative frequencies of occurrence at each point in time. A message x may be represented by a short codeword early in the transmission because it occurs frequently at the beginning of the ensemble, even though its probability of occurrence over the total ensemble is low. Later, when the more probable messages begin to occur with higher frequency, the short codeword will be mapped to one of the higher probability messages and x will be mapped to a longer codeword[2]. There are two methods to represent data before transmission:

**Corresponding Author:** Sameh Ghwanmeh, Director of Computer Center, Yarmouk University, Irbid, Jordan

**Fixed length code:** In this method it allows computers to efficiently process data which means that all characters are of equal length, even though they are not transmitted with equal frequency E.g. vowels, blanks used more often consonants, etc. ASCII is an example of a fixed length code. There are 100 printable characters in the ASCII character set and a few non printable characters, giving 128 total characters. Since log 128 = 7, ASCII requires 7 bits to represent each character. The ASCII character set treats each character in the alphabet equally and makes no assumptions about the frequency with which each character occurs[4].

**Variable length code:** In this method most frequently transmitted characters are compressed; represented by fewer bits than least frequently transmitted and compressing data saves on data communications costs. A variable length code is based on the idea that for a given alphabet, some letters occur more frequently than others. This is the basis for much of information theory and this fact is exploited in compression algorithms. To use as few bits as possible to encode data without "losing" information. More sophisticated compression techniques can use compression techniques that actually discard information[4].

There are two dimensions along which each of the schemes discussed here may be measured, algorithm complexity and amount of compression. When data compression is used in a data transmission application, the goal is speed. Speed of transmission depends upon the number of bits sent, the time required for the encoder to generate the coded message and the time required for the decoder to recover the original ensemble. In a data storage application, although the degree of compression is the primary concern[2]. Many lossless data compression algorithms exist. Some of the main techniques in use are the Huffman[5].

When compressed data is retrieved from storage or received over a communications link, it is expanded back to its original form, based on the code.

A Huffman Code is an optimal prefix code that guarantees unique decodability of a file compressed using the code. The code was devised by Huffman as part of a course assignment at MIT in the early 1950s. Huffman code constructed by using a code tree, but starting at the leaves and it provide compact code by using the binary Huffman code construction method and it has uniquely decodable code or a prefix- free code which requires that no codeword is a proper prefix of any other codeword[4].

The Huffman coding algorithm produces an optimal variable length prefix code for a given alphabet in which frequencies are preassigned to each letter in the alphabet. Symbols that occur more frequently have shorter Code words than symbols that occur less frequently. The two symbols that occur least frequently will have the same codeword length[6]. Entropy is a measure of the information content of data. The entropy of the data will specify the amount of lossless data compression can be achieved. However, finding the entropy of data sets is non trivial[7]. We have to notice that there is no unique Huffman code because Assigning 0 and 1 to the branches is arbitrary and if there are more nodes with the same probability, it doesn't matter how they are connected.

The average message length as a measure of efficiency of the code has been adopted in this work.

$$\text{Avg } L = L1 * P(1) + L2 * P(2) + \ldots + Li * P(i)$$
$$\text{Avg } L = \sum Li * P(i)$$

Also the compression ratio as a measure of efficiency has been used.

Comp. Ratio = Compressed file size / source file size * 100 %

The task of compression consists of two components, an encoding algorithm that takes a message and generates a "compressed" representation (hopefully with fewer bits) and a decoding algorithm that reconstructs the original message or some approximation of it from the compressed representation[1].

## Demonstration examples

**Example 1:** This example is derived from reference[4]

Suppose that we have a file of 100K characters. To keep the example simple, suppose that each character is one of the 8 letters from a through h. Since we have just 8 characters, we need just 3 bits to represent a character, so the file requires 300K bits to store. Suppose that we have more information about the file: the frequency which each character appears. The idea is that we will use a variable length code instead of a fixed length code (3 bits for each character), with fewer bits to store the common characters and more bits to store the rare characters. At one obvious extreme, if only 2 characters actually appeared in the file, we could represent each one with just one bit and reduce the storage from 300K bits to 100K bits (plus a short header explaining the encoding). It turns out that all characters can appear, but that as long as each one does not appear nearly equally often (100K/8 times in our case), then we can probably save space by encoding.

For example, suppose that the characters appear with the following frequencies and following codes as shown in Table 1. Then the variable-length coded version will take not 300K bits but 45K. 1 + 13K. 3 + 12K. 3 + 16K. 3 + 9K. 4 + 5K. 4 = 224K bits to store, a 25% saving. In fact this is the optimal way to encode the 6 characters present.

We consider only codes in which no code is a prefix of any other code; such codes are called prefix codes (though perhaps they should be called prefix-free

Table 1: Fixed length vs. variable length representation

| | a | b | C | d | e | f | g | h |
|---|---|---|---|---|---|---|---|---|
| Frequency | 45k | 13k | 12k | 16k | 9k | 5k | 0k | 0k |
| Fixed length code | 000 | 001 | 010 | 011 | 001 | 101 | 110 | 111 |
| Variable length code | 0 | 101 | 100 | 111 | 1101 | 1100 | - | - |

codes). The attraction of such codes is that it is easy to encode and decode data. To encode, we need only concatenate the codes of consecutive characters in the message. So for example "face" is encoded as "110001001101". To decode, we have to decide where each code begins and Ends, since they are no longer all the same length. But this is easy, since, no codes share a prefix. This means we need only scan the input string from left to right and as soon as we recognize a code, we can print the corresponding character and start looking for the next code. In the above case, the only code that begins with "1100..." or a prefix is "f", so we can print "f" and start decoding "0100..." get "a", etc.

To see why the no-common prefix property is essential, suppose that we tried to encode "e" with the shorter code "110" and tried to decode "1100"; we could not tell whether this represented "ea" or "f". (Furthermore, one can show that one cannot compress any better with a non-prefix code, although we will not show this here.)

We can represent the decoding algorithm by a binary tree, where each edge represents either 0 or 1 and each leaf corresponds to the sequence of 0s and 1s traversed to reach it, i.e. a particular code. Since no prefix is shared, all legal codes are at the leaves and decoding a string means following edges, according to the sequence of 0s and 1s in the string, until a leaf is reached. Fig. 1 shows the tree for the above code.

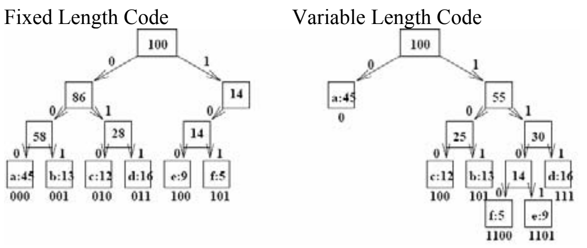

Fig. 1: Fixed length vs. variable length representation

Each leaf is labeled by the character it represents (before the colon), as well as the frequency with which it appears in the text (after the colon, in 1000s). Each internal node is labeled by the frequency with which all leaf nodes under it appear in the text (i.e. the sum of their frequencies). The bit string representing each character is also shown beneath each leaf.

**Example 2:** Suppose we want to decode the message "بسم الله الرحمن الرحيم" using Huffman code. Table 2 shows the Huffman data compression and Fig. 2 shows the Huffman representation tree.

Table 2: Huffman data compression

| Symbol | Occurrence | Probability | Codeword |
|---|---|---|---|
| ب | 1 | 1/22 | 000001 |
| س | 1 | 1/22 | 000000 |
| م | 3 | 3/22 | 101 |
| ا | 3 | 3/22 | 001 |
| ل | 4 | 4/22 | 01 |
| هـ | 1 | 1/22 | 10000 |
| ر | 2 | 2/22 | 1001 |
| ح | 2 | 2/22 | 0001 |
| ن | 1 | 1/22 | 10001 |
| ي | 1 | 1/22 | 00001 |
| Space | 3 | 3/22 | 11 |
| Sum | 22 | 1 | |

**Average search length:**

Avg L = $\sum$ Li * P (i)

= 6*1/22 + 6*1/22 + 3*3/22 + 3*3/22 + 2*4/22 + 5*1/22 + 4*2/22 + 4*2/22 + 5*1/22 + 5*1/22 + 2*3/22

= 75/22

= 3.41

Space required to store the original message = 22*8=176 bit

Space required to store the decoded message= 75 bit

Comp. Ratio = 42.61 %

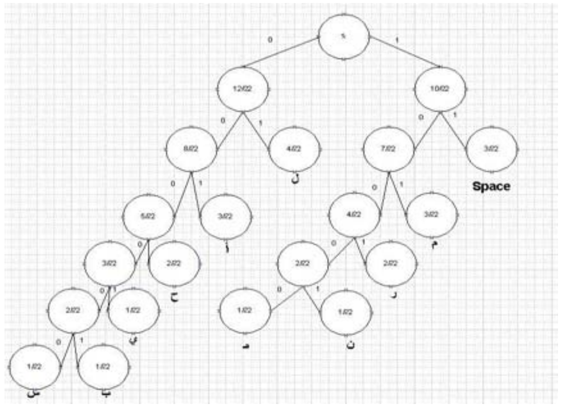

Fig. 2: Huffman tree

## RESULTS

A series of experimental tests have been performed to measure the efficiency of compressing data results on both Arabic and English text. We compare both Arabic and English texts of different sizes using Huffman compression algorithm. It has been shown that as the file size increases, the compression ratio on Arabic Text decreases as well as the average message length. This is expected because when the file size increases the frequency of the symbols will increase, so we have better compression on larger files.

Table 3: The effect of file size on compression for Arabic text

| File Size | | Statistics | | |
|---|---|---|---|---|
| Source | Compressed | Comp. Rate | Avg. ML | Time |
| 1 KB | 653 | 63.77% | 5.102 | 0.02 |
| 2 KB | 1236 | 60.35% | 4.828 | 0.03 |
| 3 KB | 1814 | 59.05% | 4.724 | 0.05 |
| 4 KB | 2408 | 58.79% | 4.703 | 0.07 |
| 5 KB | 3009 | 58.77% | 4.702 | 0.08 |
| 6 KB | 3601 | 58.61% | 4.689 | 0.1 |
| 7 KB | 4175 | 58.24% | 4.66 | 0.1 |
| 8 KB | 4729 | 57.73% | 4.618 | 0.13 |
| 9 KB | 5323 | 57.76% | 4.621 | 0.14 |
| 10 KB | 5872 | 57.34% | 4.588 | 0.15 |

Table 4: The effect of file size on compression for English text

| File Size | | Statistics | | |
|---|---|---|---|---|
| Source | Compressed | Comp. Rate | Avg. ML | Time |
| 1 KB | 691 | 67.48% | 5.398 | 0.02 |
| 2 KB | 1401 | 68.41% | 5.473 | 0.04 |
| 3 KB | 2015 | 65.59% | 5.247 | 0.06 |
| 4 KB | 2636 | 64.36% | 5.148 | 0.07 |
| 5 KB | 3265 | 63.77% | 5.102 | 0.09 |
| 6 KB | 3745 | 60.95% | 4.876 | 0.09 |
| 7 KB | 4226 | 58.96% | 4.717 | 0.11 |
| 8 KB | 5099 | 58.86% | 4.709 | 0.16 |
| 9 KB | 5969 | 57.74% | 4.619 | 0.16 |
| 10 KB | 6300 | 57.74% | 4.627 | 0.16 |

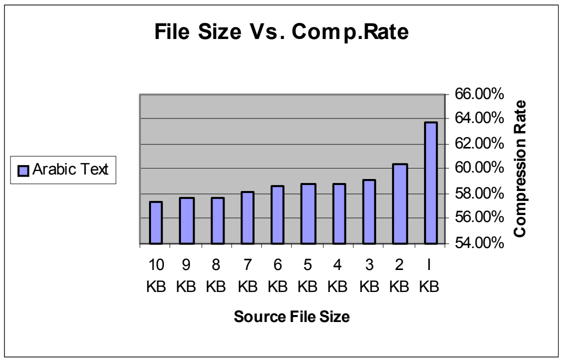

Fig. 3: The effect of file size on compression for Arabic text

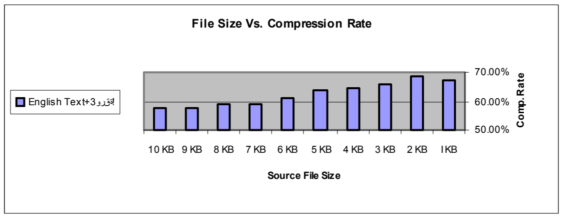

Fig. 4: Effect of the file size on compression for English text

Table 3 and Fig. 3 show the effect of file size on compression for different Arabic text. Also same resulted for English text, when the file size increases the compression rate and the average message length decreases. Clearly Table 4 and Fig. 4 show the effect of file size on compression for different English text. Also we found that the time required to compress files increases as file size increases. However, the rate of

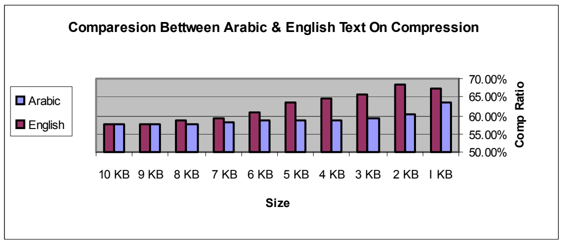

Fig. 5: Comparison between compression on Arabic and English Text

change of the time required to compress files with larger sizes is small (Table 3 and Table 4). Additionally it has been found that the efficiency of compression on Arabic language is better than on English language (Fig. 5).

## CONCLUSION

It has been found that the compression on both Arabic and English text saves space and reduces transmission time. Also we found that as the file size increases the compression ratio decreases for both Arabic and English text as will as the average message length and that is expected because when the file size increases the frequency of the symbols will increase, so we expect to have better compression on large files. We also found that the efficiency of compression on Arabic text is better than compression on English text. Additionally, results show that the main factor which significantly affects compression ratio and average message length is the frequency of the symbols on the text.

## ACKNOWLEDGMENTS

The authors gratefully acknowledge and highly appreciate the financial support and the remarkable resources provided by Yarmouk University, Irbid, Jordan.

## REFERENCES

1. Blelloch, E., 2002. Introduction to Data Compression. Computer Science Department, Carnegie Mellon University.
3. Cormak, V. and S. Horspool, , 1987. Data compression using dynamic Markov modelling. Comput. J., 30: 541–550.
2. Vo Ngoc and M. Alistair, 2006. Improved word-aligned binary compression for text indexing. IEEE Trans. Knowledge & Data Engineering, 18: 857-861.
4. Kaufman, K. and T. Shmuel, 2005. Semi-lossless text compression. Intl. J. Foundations of Computer Sci., 16: 1167-1178.
5. Capocelli, M., R. Giancarlo and J. Taneja, 1986. Bounds on the redundancy of Huffman codes. IEEE Trans. Inf. Theory, 32: 854–857.
6. Gawthrop, J. and W. Liuping, 2005. Data compression for estimation of the physical parameters of stable and unstable linear systems. Automatica, 41: 1313-1321.
7. Kesheng, W., J. Otoo and S. Arie, 2006. Optimizing bitmap indices with efficient compression. ACM Trans. Database Systems, 31: 1-38.