FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

DEGREE PROGRAMME IN ELECTRONICS AND COMMUNICATIONS ENGINEERING

BACHELOR'S THESIS

HUFFMAN SOURCE CODING

Author	Antti Koivula
Supervisor	Antti-Heikki Tölli
Second Supervisor	Mohammadjavad Salehi

June 2021

Koivula A. (2021) Huffman Source Coding. University of Oulu, Degree Programme in Electronics and Communications Engineering. Bachelor's Thesis, 25 p.

ABSTRACT

In this work, A Huffman source coding system is studied and implemented. The work will go through the basics of the source coding theorem, standard Huffman code is introduced, its weaknesses in a practical system are presented, and finally, methods and algorithms are introduced to overcome these weaknesses. In Particular, the preset dictionaries and Vitter algorithm are introduced. Then, the implementation is presented and the performance is studied by compressing text files.

Keywords: Source coding, Huffman, Vitter.

Koivula A. (2021) Huffman Lähteenkoodus. Oulun yliopisto, tieto- ja sähkötekniikan tiedekunta, elektroniikan ja tietoliikennetekniikan tutkinto-ohjelma. Kandidaatintyö, 25 s.

TIIVISTELMÄ

Tässä työssä tutkitaan ja toteutetaan Huffman lähteenkoodaus järjestelmä. Työssä käydään läpi lähteenkoodauksen teoriaa, standardi Huffman koodaus, sen heikkoudet käytännön järjestelmässä, ja lopuksi keinoja näiden heikkouksien yli pääsemiseksi. Erityisesti huomioidaan etukäteen lasketut lähdekoodit ja dynaaminen Vitter algoritmi. Lopuksi työ toteutetaan ohjelmistona ja eri koodaustapoja verrataan keskenään kompressoimalla tekstitiedostoja.

Avainsanat: Lähteenkoodaus, Huffman, Vitter.

TABLE OF CONTENTS

FOREWORD

This work was a part of Bachelor’s degree programme in electronics and communications engineering in university of Oulu. The thesis subject was selected from a given set.

I would like to give my thanks to the project supervisors Prof. Antti-Heikki Tölli and Mohammadjavad Salehi. I also would like to thank my mother, father, and Minna.

Oulu, June 23, 2021

Antti Koivula

LIST OF ABBREVIATIONS AND SYMBOLS

ASCII	American Standard Code for Information Interchange
CPU	Central processing unit
Unicode	Universal Coded Character Set
UTF-8	Unicode Transformation Format

c_i	ith codeword
C	Set of codewords
D	Total word length
H	Entropy
I	Self-information
l	Average codeword length
l_i	Length of ith codeword
n	Number of symbols
p_i	Probability for ith symbol
s_i	ith symbol
S	Set of symbols
t_i	Size of an alphabet for ith iteration
T	Huffman tree
w_i	Weight for ith codeword
Z_t	Average codeword length for alphabet of size t

ζ	Set of codeword lengths
$\Uparrow$	Preamble symbol

$\log_2()$	binary logarithm

1 INTRODUCTION

We have been collecting and producing digital information in an increasingly faster manner ever since the digital computer was commercialized [1]. Now, the internet has grown to a point where the aim is to integrate internet connected computing into anything; one of the visions of Internet of Things (IoT) is to gather data from hundreds of thousands nodes and send it to back-end servers [2]. The large amount of data requires a large number of resources: storage and channel capacity.

Being a key enabler for modern information systems, source coding allows more effective utilization of resources. It improves performance of communication channels and storage services by reducing the size of data [3]. Source coding is a common term in communication systems while it is called data compression in computer systems.

Being simple and in an information theoretic sense close to optimal, Huffman code is the source code that is taught first, when one starts to learn about source codes [4]. Even though it is simple, it involves and showcases the information theory in an intuitive manner. In this work, standard Huffman code is studied by compressing text files.

Text is one of the major data types in computer systems, but is often written with other alphabets than Latin. By using a fixed length code, character encodings represent Latin characters commonly with a byte while other alphabets are represented with two or more bytes. However, in the context of a file, only one alphabet might exist [5]. These character encodings increase the file sizes for the other alphabets, and therefore, being simple and intuitive, text files are a good way to study data compression.

There are multiple methods that improve the standard Huffman coding. The most basic, but effective, is to use precalculated code dictionaries, which are present both at encoder and decoder [6]. In this work the preset dictionaries were derived by analyzing over 1000 books by counting the occurrence frequencies of the words. Two dictionaries were used: one with words and one with text characters. The book analysis was conducted using Matlab analysis platform.

The second class of methods for improvement are adaptive source coding algorithms. They use dynamically generated frequency-sorted binary trees, which are updated always after a symbol has been coded. The main advantage is that there is no need to transmit or store the code; both the encoder and decoder synchronously generate their own code. The main disadvantage is that if the standard Huffman code is used, the adaptive algorithm is increasingly slower as the Huffman binary tree grows [7].

In this work, the adaptive Huffman coding was improved using the Vitter algorithm. It builds on so-called sibling property [7]. The sibling property and its allowed Huffman tree transformations were studied.

The main part of this work was the implementation of the Huffman source coding algorithms. The static standard Huffman source coding was implemented for word and text character symbols. The adaptive source coding was implemented for text characters only. The implementation was done using the Python programming language.

The performance of the different Huffman source coding methods were studied to find out how close to the source coding theorem limit they can get. The study was done by compressing text files, calculating the limit and comparing to the average codeword length. Also, performance metrics were calculated from compressed file sizes. The study showcases why source coding is important.

2 SOURCE CODING

Source coding systems fundamentally have five parts, Figure 1. First, a data source in digital form. For example, ASCII (American Standard Code for Information Interchange) encoded text is represented with an 8-bit code. The second part is an encoding algorithm, and third is the compressed output data of the compressor. The fourth part is the decoding algorithm, and the last part is the decoded output data of the decompressor. Source coding systems are complete only when both encoding and decoding are provided [4].

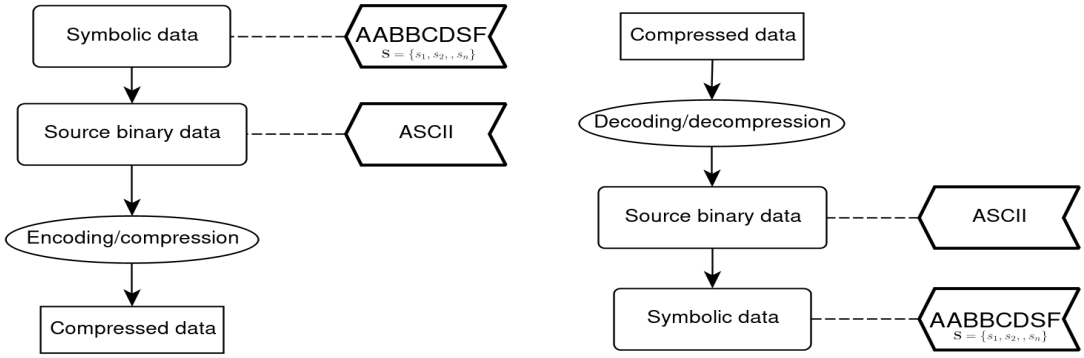

Figure 1. Structure of the source coding systems.

In addition to image and sound, text is one of three types of source data in computer systems. Information a line of text carries can be thought to be constructed from symbols, which can represent single characters from a standard alphabet or whole words [4]. In this work, three common computer system character encodings are used: ASCII, ISO 8859 (International Organization for Standardization) and UTF-8 (Universal Coded Character Set Transformation Format-8-bit). These character encodings represent the character symbols with fixed length codes.

The alphabet of a source is $S=(s_1, s_2, \cdots, s_n)$. A character encoding representing the alphabet with binary code $C=(c_1, c_2, \cdots, c_n)$, where c_i is the codeword for a symbol s_i. For ASCII, the length of the codewords is seven bits [8], and for ISO 8859 it is eight bits [9]. UTF-8 has 8-bit code units, which means that the fixed length units can encode a character symbol in one to four bytes. The size *n* of the UTF-8 alphabet can be 2^{31} symbols [10]. Most source files use only a subset of a larger alphabet.

Source coding systems use variable length codes for data compression. A character encoding is designed to represent as many symbols as is thought to be needed in computer systems using bytes as data units, and therefore, they do not take into account the possibly different sources. In comparison, a source coding system uses a mathematical model of the source to produce a code *C* that assigns short codewords to the most frequently used symbols. Therefore, a model identifies redundancy in the source data and is the basis for effective compression [4].

A source coding system is characterized by its space-saving and time efficiency. For lossless compression (the output of the decoder is exactly the same as the input to the encoder), the first measure is compression ratio, which is defined as the ratio of the compressed data size to the source data size, Equation 1:

$$Compression\,ratio = \frac{Length\,after\,compression}{Length\,before\,compression} \tag{1}$$

The second measure, saving percentage, is the shrinkage between the source and compressed data, Equation 2:

$$Saving\ percentage = \frac{Length\ before\ compression - Length\ after\ compression}{Length\ before\ compression}\% \quad (2)$$

The space-saving measurements are instance based, the results depend on the used source. In addition to space-saving, compression time is an equally important measure, and the encoding and decoding times are considered separately. The computational complexity of an algorithm affects the compression time, and it is important to not only have a good model of the source but also a good model of the data structures that are used to generate the codes [4].

If the source with alphabet *S* is modeled with probability distribution $P=(p_1, p_2, \cdots, p_n)$, and the compression system has a code *C* with codeword lengths $\zeta=(l_1, l_2, \cdots, l_n)$, the average code length is computed as in Equation 3:

$$\bar{l}(P,\zeta)=\sum_{j=0}^{n} p_j l_j \quad (3)$$

Since the fixed length character encodings commonly represent the most used letters in English with 8 bits, the average length must be less than that. The goal is to have as small average code length as possible [4].

The effectivenes of a source code can also be studied with redundancy left in the code, which can be computed using the Shannon's source coding theorem. For lossless compression, the smallest average code length is lower-bounded by the entropy of the source, Equation 4:

$$H(P) \leqslant \sum_{j=0}^{n} p_j l_j \quad (4)$$

The entropy of the source with probability distribution *P* is defined as, Equation 5:

$$H(P)=-\sum_{j=0}^{n} p_j \log_2(p_j) \quad (5)$$

where $I(s_j)=-\log_2(p_j)$ is self-information of a source symbol. The self-information is inversely proportional to the probability of a symbol, and it can be interpreted as a number of bits needed to describe the symbol as the outcome of a random test. Then, the entropy is the average number of bits per symbol that is needed to describe the source data. The redundancy is the difference between average code length and the entropy, Equation 6:

$$R=\left|-\sum_{j=0}^{n} p_j \log_2(p_j) - \sum_{j=0}^{n} p_j l_j\right| \quad (6)$$

When the redundancy is zero, the code is optimal. The redundancy is a feasible measurement of the effectiveness of the code only when the statistical model *P* is known [4].

2.1 Huffman coding

Huffman codes are variable length, uniquely decodable prefix codes [11]. A prefix is the first bits of a codeword. Ensuring that there is only one way to decode compressed messages, the prefix property prohibits any codeword from being a prefix for another codeword [4]. Then, without looking ahead, a Huffman encoded message can be decoded by reading bits in one at the time and the resulting codeword will be unique.

The original Huffman coding is a static system: the mathematical model used to derive the code stays unchanged from compression to decompression [4]. In other words, before a code is generated, the static Huffman code knows a precalculated symbol occurrence model. The occurrence model presents the occurrence frequencies of symbols in an alphabet. In comparison, for an adaptive method, the occurrence model is always updated after a symbol is read from a source, and a new code is generated.

Because the prefix codes can be represented as binary trees, the Huffman codes are derived by constructing the binary tree from leaves to root. The standard Huffman code is called a canonical and minimum variance code, and it minimizes the weighted path lengths (Equation 3) among all binary trees. The standard Huffman code has the following approach [4], Figure 1:

Input: a ordered list of *n* leaf-nodes for an alphabet, from lowest frequency to highest.
Output: *n* Huffman codewords

Process (continue until the sorted list has one node left)**:**
1: take two leaf-nodes with minimal weights from the sorted list.
2: merge the two nodes. Combine symbols and weights.
-When combining: the node with larger frequency is assigned 1, the other gets 0.
3: place the new item to highest possible position on the sorted list.

Output every path from the root to a leaf.

Figure 2. Standard Huffman code algorithm.

The process of deriving the Huffman tree and code is shown in Figure 3. It starts with an ordered list of leaf-nodes. A leaf-node corresponds to a symbol and its occurrence frequency. The ordering is from the smallest frequency to the largest. Then, two leaf nodes with the smallest weights are combined into one internal node and moved to the highest possible position in the ordered list. The two lowest frequency nodes are combined until one internal node, the root node, remains. When combining the nodes, a code is generated by assigning 1 for the left child and 0 for the right child. The path from the root to a leaf node defines the code.

The encoding happens by reading and outputting a code from the path. With static Huffman methods, this tree structure needs to be known in the decoding process, and a binary string is decoded by traversing the tree from the root node to a leaf-node. The route is defined by the bit values in the binary string. For example, the binary string 1111 leads to the leaf-node “E” of the Huffman tree defined in Figure 3.

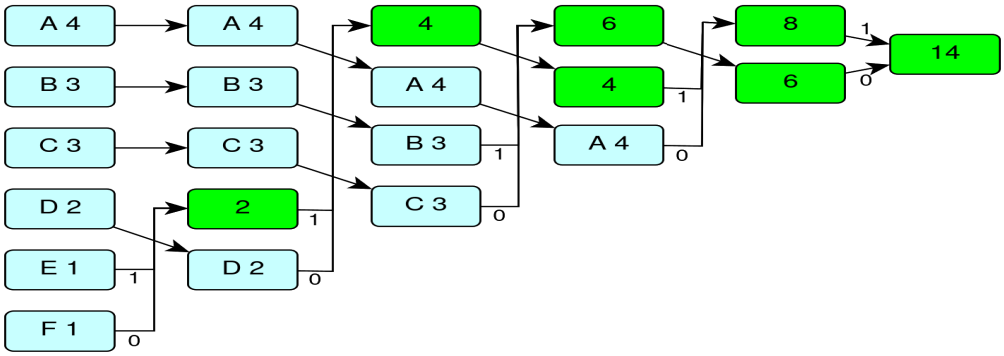

Figure 3. Deriving of the Huffman tree.

A static Huffman compression system can either be a two pass method, where the first pass counts the symbol frequencies and the second encodes [11], or the compression system could use predefined code dictionaries computed outside of a compression instance [6]. The penalty for the two pass method is the longer compression time and the data heap for the code. Because a decoder does not know the codes used in encoding, the code structure (heap) must be transmitted to the encoder. However, this method has the most accurate statistical model and can reach the lower bound for optimality [4].

In contrast, the predefined dictionary methods are faster because the frequency models and the Huffman trees are precalculated. Therefore, they also do not include the heap. A predefined dictionary method uses multiple predefined dictionaries. The predefined dictionaries are precomputed Huffman tree structures computed for a specific symbol alphabet. For example, the specific alphabets can be word alphabets and text character alphabets. Multiple dictionaries are needed so that all possible symbols can be decoded.

Because the predefined methods have multiple dictionaries, the systems must use preambles (flag symbols and codes) when traversing from common dictionaries to the rarer ones and back [6]. Therefore, if the source data includes rare symbols, the preambles start to dominate and the compression performance degrades. Also, the methods introduce saving losses due to model mismatches.

The Huffman codes reach entropy lower bound (Equation 4) only if the probabilities for the codewords are negative powers of two [4]. The standard Huffman code has the minimal average codeword length over all Huffman codes. By comparing the average codeword length (Equation 3) to the entropy (Equation 5), it is noticed that they are equal only if the self-information $I(s_j)$ per symbols are integers. Self information is an integer only if the probability is a negative power of two. Therefore, for general sources, the standard Huffman code is not optimal, because the occurrence probabilities are not negative powers of two.

2.2 Adaptive Huffman coding

Adaptive Huffman is a one pass method that, in contrast to the two pass methods, maintains a dynamic Huffman tree. The general algorithm is defined in Figure 4. In the start of an encoding or decoding process, the initial alphabet is empty, except for a flag symbol that always has a frequency of zero; if compared to the standard Huffman tree generation process, the initial ordered list has only the flag symbol. Therefore, the initial binary tree has only the root node [4].

The adaptive encoding algorithm reads the input symbols one by one and, if the symbol is known, uses the dynamic code. The dynamic code also is defined by a path from the root

node to a leaf-node. Since the adaptive systems initially do not have any known symbols, the first occurrences must be coded with an encoding known to the computer system, for example UTF-8 [11]: a first occurrence is encoded with the code for the flag symbol followed by, for example, UTF-8 character code. After, the symbol has been encoded, the occurrence model is updated and the Huffman tree is generated anew. This makes the adaptive Huffman coding a two dictionaries method, where the preamble (the flag) marks the start of an unknown symbol [4].

The first three steps of the adaptive Huffman encoding algorithms (Figure 4) are the same for all adaptive Huffman algorithms [4]. The fourth step, update, defines the complexity of the algorithm. If the standard Huffman code is used for the update, the whole binary three is computed anew after every iteration of the process, and the whole system slows down when the size of the alphabet increases [7].

Input: alphabet with only the flag S = { ⇑ }, one node Huffman tree T

Process (continue until all symbols are encoded)**:**
1: read a symbol → s.
2: if the symbol is known:
-output the code from the Huffman tree T(s)
3: else:
-output the code for the flag T(⇑) and the code for the symbol from the known character encoding system g(s).
4: update the alphabet S and and the Huffman tree T

Figure 4. Adaptive Huffman encoding algorithm.

The big advantage is that the encoding and decoding use the same algorithms. Therefore, both of them have the exact same copy of the Huffman tree, and the tree needs not be transmitted with the compressed data. If the complexity of the update process does not hinder the algorithm, the other big advantage is that there is no preprocessing overhead [7].

2.3 Vitter algorithm

Vitter algorithm improves the performance of the update method of the adaptive Huffman coding. It builds on the so-called sibling property of Huffman trees, which states that two conditions must hold for a binary tree to be a Huffman tree. First, the leaves must have nonnegative weights and the weights of all internal nodes are the sums of their children. Second, the nodes can be ordered and numbered by weight in nondecreasing order. Then, the nodes $2j-1$ and $2j$ for $1 \leqslant j \leqslant p-1$, where p is the number of the leaves, are siblings and their parent node is higher in the numbering [7].

Figure 5 visualizes the first and second conditions. On the left are two initial Huffman trees. Their internal nodes are shown to have weights that are sums of their children. Also, alongside of the nodes are marked the numbering: the nodes are in nondecreasing order, and the leaves $2j-1$ and $2j$ are siblings.

The sibling property allows transformations of the Huffman trees so that the resulting binary tree is still a Huffman tree. The first transformation type to be defined is interchange of nodes. Figure 5 also visualizes these interchange transformations. With interchange transformation, the nodes, and their children, can be interchanged with a node of the same weight at the highest number and the binary tree is still a Huffman tree [7].

The weight of the internal parent nodes are the same after interchange. If the interchange happens between nodes at the same depth, the depth of their children and the nondecreasing order is preserved. Otherwise, the initial tree would not have been a Huffman tree. If the interchange is to a depth higher than the target node, the target node was at the highest number on its level and its children had weights smaller or equal to the weights of the other nodes at that level. Therefore, the ordering of the weights is preserved and the numbering can be done anew. Their properties are shown in Figure 5 [7].

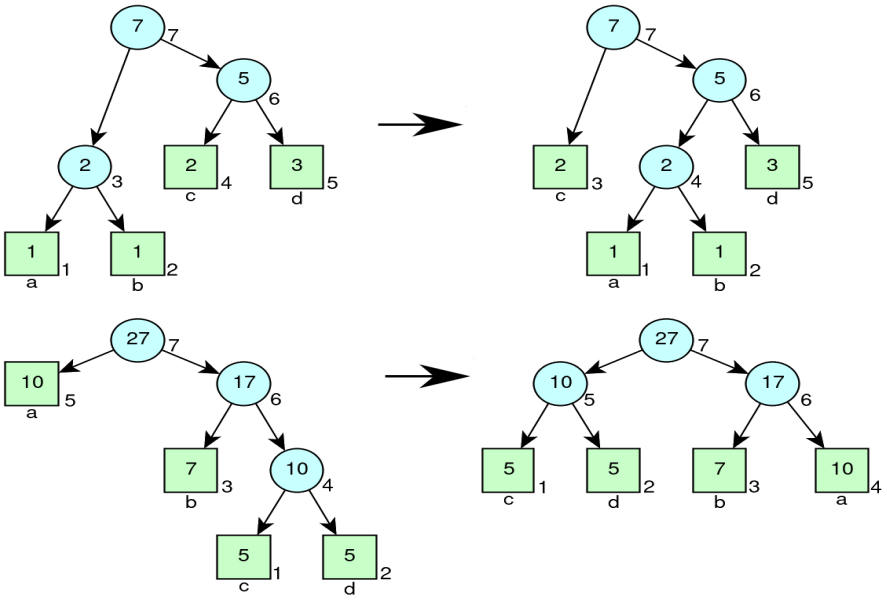

Figure 5. Interchange transformations of Huffman trees.

The second transformation type allows incrementation. If a node is at the highest number for its weight, incrementing its weight cannot make it larger than the weight of the next node in the numbering. Then, the sibling property is preserved and the binary tree is a Huffman tree [7].

As is seen from Figure 5, The interchanges affect the parents of the target nodes, but the children of the target nodes are unaffected. The incrementations propagate through the tree, from a child to its parent, until the root node is incremented. These transformations change the structure of the Huffman tree, but the affected nodes only see themselves and their direct relatives [7].

The sibling property and the defined transformations allow the Huffman tree to be updated by focusing on nodes that are incremented. When a Huffman tree is updated, a leaf-node corresponding to a symbol to be updated is called the first target and is the initial node to be transformed. Its weight can be incremented, and it can possibly interchange with other nodes initially higher at numbering. After a node is transformed, the target switches to its parents.

The Vitter algorithm, that heavily relies on these transformations, was designed to minimize the average codeword length (Equation 3), the total word length (Equation 7), and the maximum length of any codeword $max(l_j)$, where j is $1 \leqslant j \leqslant t$, t is the size of the alphabet that has been processed so far. If Z_t is the average codeword length for the static Huffman code for the alphabet of size t, the Vitter algorithm then minimizes the difference $D_t - Z_t$. Also, the worst case is upper bounded by $D_t < Z_t + t$ [7].

$$D=\sum_j l_j, \tag{7}$$

The Vitter algorithm defines additional conditions that ensure these goals. First, in addition to the numbering defined by the sibling property, the Vitter algorithm uses implicit numbering, where the numbering is in increasing order by the tree depth. Nodes at higher depth have numbers larger than the Nodes at lower depth, and nodes at same depth are numbered in increasing order from left to right. With this kind of data structure, target nodes cannot interchange to lower depths [7].

In order to minimize the difference $D_t - Z_t$, the Vitter algorithm limits the number of interchanges to one. Through the update process, only a leaf node corresponding to the known symbol to be updated can interchange. For this to happen, the Vitter algorithm introduces an invariant property. It states that all leaves of weight w precede all internal nodes of weight w; the leaves of weight w have smaller implicit numbers than the internal nodes of weight w [7].

This orders the nodes in groups called blocks. A block is an ordered group of either only leaf nodes or internal nodes, and they have equal weights. The leader of any block has the highest implicit number [7].

Finally, the Vitter algorithm only allows one interchange if the target leaf-node corresponds to a previously known symbol. Otherwise, the Vitter algorithm introduces a new transformation: an important part of the Vitter update process is the slide and increment operation shown in Figure 6. The operation is done with a block and a node [7].

The slide operation is as follows: if the target node is a leaf node with weight w, then the internal nodes with weight w are shifted to left in implicit numbering, and the leaf node is moved (slided) to the leader position of the internal nodes. Due to the invariant property, the internal nodes of weight w initially succeed the leaf node, but after the shift operation their numbering are decremented by one.

When a target node is slided to the leader position of its target block, the target leaf node is incremented, and the next target node is set to be the parent of the leaf node [7].

With internal nodes, the process is the same, except the target block to be sifted has weight w+1, and the next target node is the initial first parent node of the target [7].

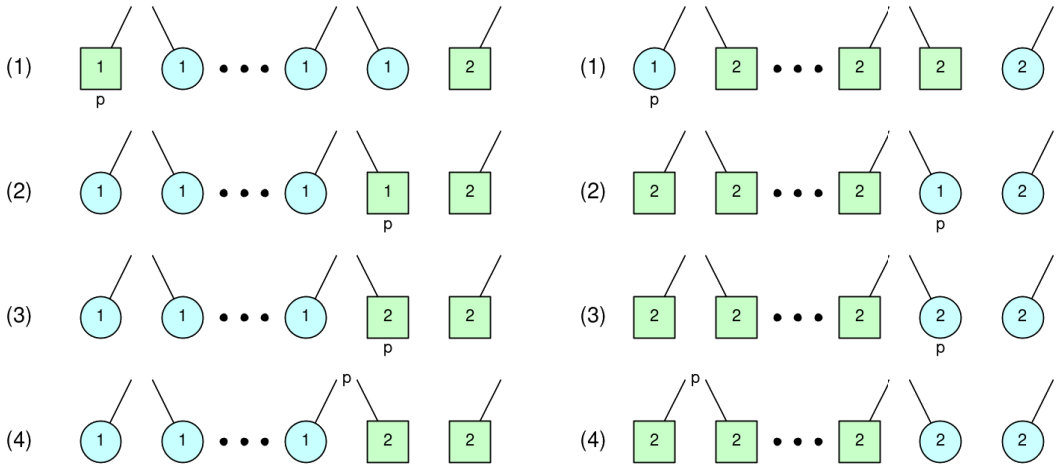

Figure 6. Slide and increment.

Figure 5 shows the Vitter algorithm. In the initial phase, the only allowed interchange is performed if the symbol is previously known. Otherwise, the preamble node of zero weight is transformed to an internal node whose left child is the preamble and the right child is the new

symbol leaf. If the target represents a new symbol, or it is a sibling to the preamble node, then the target leaf-node is set to update after the other parts of the tree is updated, and the target is switched to a parent. This is done because, as is seen from Figure 6, a leaf-node of weight zero can only slide to the leader position of zero weighted internal nodes. [7]. After the initial phase, the slide and increment operation is performed until the target node is set to be the root node.

```
Process update( symbol ):
    leaf_to_increment = None
    q = leaf node corresponding to the target symbol
    if q is the preamble (unknown symbol):
        -Replace q by an internal node with two leaf node children with zero weights.
         The right child corresponds to the new symbol.
        q = the new internal node
        leaf_to_increment = the right child of the new internal node
    else:
        Interchange q with leader of its block
        if q is the sibling of the preamble node
            leaf_to_increment = q
            q = parent of the q

    while q is not root of the Huffman tree:
        q = SlideAndIncrement( q )

    if leaf_to_increment is not None:
        SlideAndIncrement( leaft_to_increment )

Process SlideAndIncrement( p ):
    wt = weight of p
    p_f = former parent of p
    if p is an internal node:
        b = the block of leaves of weight wt + 1
        Slide p in the tree higher than the nodes in b
        p = p_f
    else if p is a leaf node:
        b = the block of internal nodes of weight wt
        Slide p in the tree higher than the nodes in b
        p = new parent of p
    return  p
```

Figure 7. Vitter algorithm.

As is seen from Figure 6, the next target node is always an internal node. Also, as is seen from the algorithm (Figure update(symbol):), the slide and increment is the most operation intensive part of the Vitter algorithm. However, one iteration of the algorithm's **while** loop will only operate on a maximum of one block and node, and because the Huffman tree is a binary tree, the sizes of the blocks will decrease from leaf to root. These make the Vitter update algorithm faster than running the standard Huffman code after every symbol [7].

3 IMPLEMENTATION OF HUFFMAN SOUCE CODING SYSTEMS

Multiple Huffman source coding methods were implemented. The methods are either static or adaptive. The implemented static methods either generate a code from a source instance or use a predefined code dictionaries generated from a statistical analysis. Two adaptive methods were implemented: a naive method that uses the standard Huffman code, and another that uses the Vitter algorithm for the dynamic Huffman tree. In this chapter, the implementations of the methods are presented. First, the requirements are listed. Second, programming language and analysis tools are presented. Then, the structure of the implementation. Finally, the implementations are evaluated using space-saving metrics: compression factor and space-saving percentage.

3.1 Requirements

The main requirement for any compression system is that the output of a decoder is bit-by-bit equal to the uncompressed input of the encoder. In addition to the general requirements, the following includes requirements for text specific compression system:

- The system must be lossless
  - Input and output are bit-by-bit equal
- The system must process text files as input
  - Recognize files
- The system must produce text files as output
  - Different operating systems use different control symbols. Even though the human-readable text files may look the same, the code for new-lines can differ.
- The compressed data must be output as binary file
- The system must transmit a header
  - A file signature for the implementation
  - Metadata of the input text file
  - Possibly the code
  - Padding size
- The compressed data must be padded to full bytes.
- With static coding, the implementation must be able to read words as symbol
- The system must be able to create and analyze binary data
- The system must support large number of text characters

3.2 Programming language and analysis tools

The chosen programming language was the general purpose language Python. The programs developed with it are highly portable since it can be used with virtually all operating systems. Python has a large number of modules for both scientific computing and low level operations. Also, it uses a dynamic typing system: variables and their types do not need to be declared. This is taken advantage of since the typing system uses UTF-8 character encoding, and it can be used to effortlessly translate between raw bytes and text characters. Therefore, the implementation easily can be made to support a large number of text characters.

Python can have bad memory efficiency and its garbage collection system can cause unnecessary cache misses. However, many third-party libraries are memory efficient. In this work, the following Python modules were used:

- **bitstring**: Used for creation and manipulation of binary data. The binary data can be sliced, joined and even searched. Also, it allows reading the binary file as a stream of binary data [12].
- **nltk**: Natural Language Toolkit. Used to read a text file as words instead of text characters [13]. Relevant functions:
  - nltk.word_tokenize
- **argparse**: Used for complementary command-line interface.
- **Python-magic:** python interface to libmagic file type identification library. Used to detect text file encodings. The implementation was done with Linux operating system, but libmagic binaries exist for Windows operating system also.

For analyzing written texts, Matlab and its Text analytics Toolbox were chosen. They allow fast analysis of a huge number of text books and provide utilities for modeling and plotting the statistical results.

3.3 Code structure

The main structure of the source code files is shown in Figure 8. The source codes under the I/O -operations handle reading and writing of the text and binary files. The module for binary files handles also the generation and parsing of the binary file headers. The module for text files can read the input text either as text character or as whole words.

The source codes under the Huffman coders implement the algorithms introduced in chapter 2. Each coder module has a class that dynamically can be constructed either as an encoder or decoder. The standard Huffman code module implements the algorithm described in Figure 2, while the Vitter tree module implements the Vitter algorithm (Figure 5). The Vitter tree module also implements a method for retrieving the code for a given symbol and a method for retrieving a symbol for a given code.

The symbol frequency module implements methods for counting symbol frequencies for a given text file. The symbol can either be text characters or words. The frequency model from a file reads the frequency data from a file that has predefined frequencies for symbols.

Command-line interface is a complementary, and therefore, completely optional. A complete system can be built from the previous modules without it. Process settings module implements a class for holding setup data, and it is commonly used by all methods.

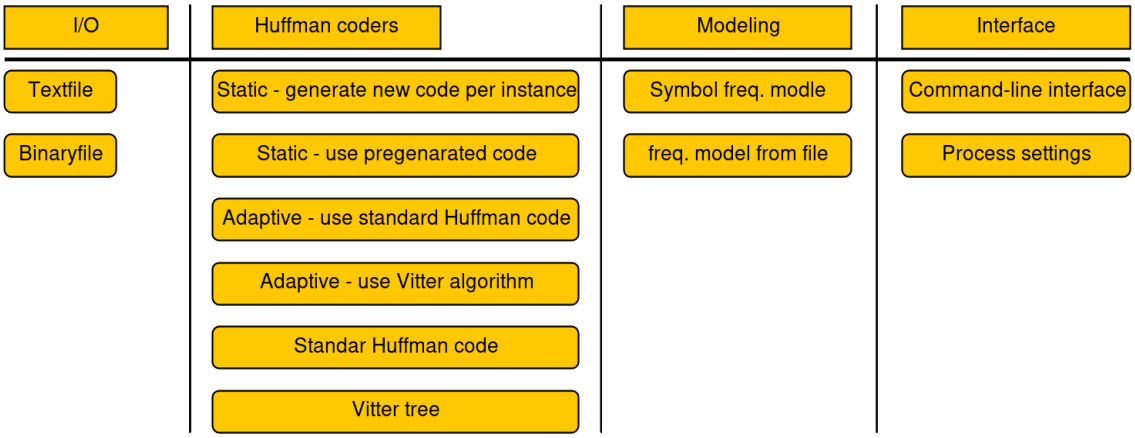

Figure 8. Organization of the source code.

The general operation of the implementation is shown in Figure 9. The encoders and decoders are constructed dynamically. The inner workings of the encoder and decoders differ but the interfaces they offer are the same.

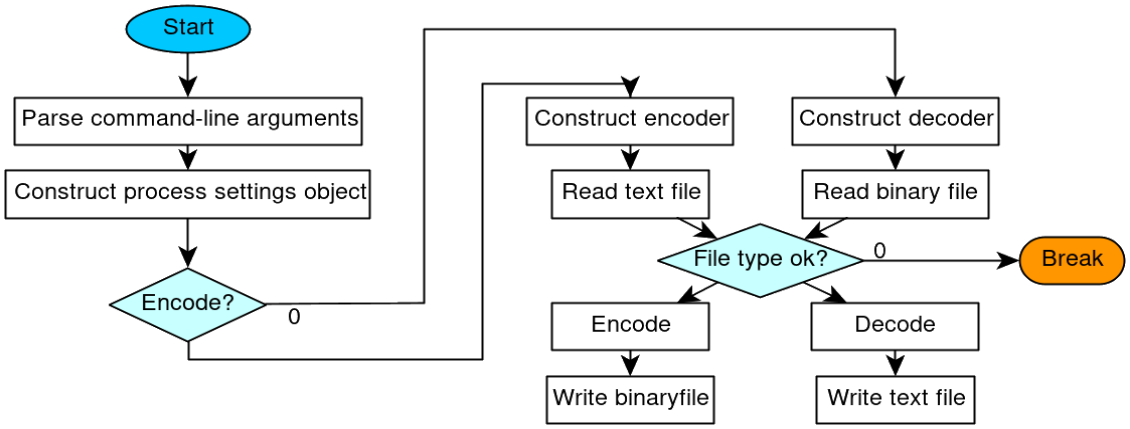

Figure 9. Operation of the general encoder/decoder.

3.4 Precalculated frequencies

One of the implementations uses precalculated frequencies and creates a predefined code dictionary. An analysis was conducted to determine what words should be selected. Over 1000 books in English were downloaded from the Project Gutenberg libraries [14], and by using Matlab, all of the books were analyzed to count the occurrences for all words that were present. As is seen from Table 1, the 100 most common words in English account for almost 55% of all written text. The Figure 10 presents the probability of occurrence and self-information as a function of word indexes, from the most common to the most rare. As is seen from the collected data, written texts have large amounts of redundancy.

For the implementation, the 1000 most common words were selected. They account for almost 75% of all written text in English. Since the implemented design only has 1000 words, it must use multiple dictionaries. Initially it will try to use the word dictionary. If an input is not found from that dictionary, a preamble marks a change to a text character based dictionary. A simplified text character dictionary will encode and decode the rest, 25% of the words, not included in the word dictionary. The simplified character dictionary uses the 100

most common characters, which were derived from the same 1000 books analysis. If a text character is not found from the second dictionary, then it is encoded with UTF-8. After a word is encoded with the character dictionary, an appendix is inserted, which will tell the decoder to switch back to the word dictionary.

Since the analysis suggested that, with the dictionary of 1000 words, the probability of a miss is about 0.25, the preamble was positioned to front in the ordered weight list, just after the most common words. With the character dictionary, the probability of a miss was 0.0003, and position was found from the end of the ordered weight list. Since the appendix is always used with missed words, it was given the same frequency as for the word miss.

Table 1. Groups of the most common words in English and their ratios to all counted word occurrences.

Number of words from the most common to the most rare	**Frequency**	**Frequency / total number of occurrences (90465270)**
First 100	49371983	0.5458
First 1000	67519479	0.7464
First 7500	81391104	0.8997

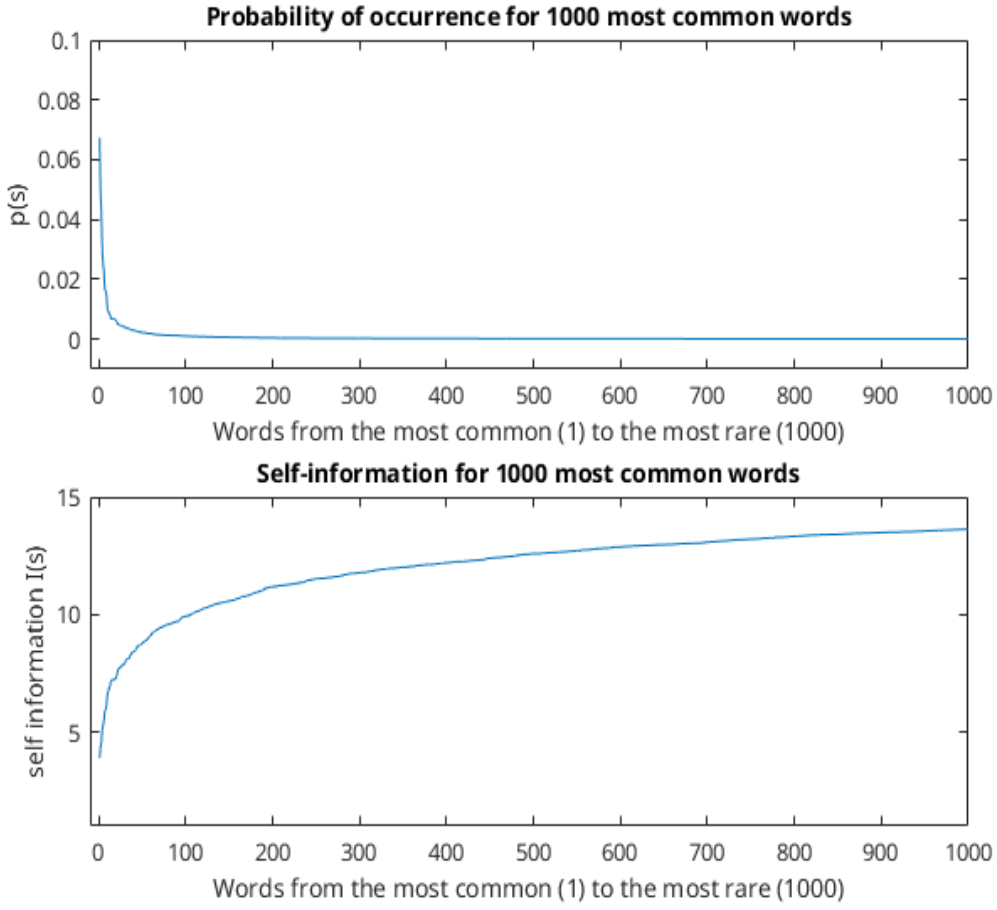

Figure 10. The probability of occurrence and self-information for the 1000 most common words.

3.5 Evaluation

The implementation can successfully compress and decompress ASCII, ISO-8859 and UTF-8 files. The outputs of the decompressor are equal to the inputs to the compressor, except for the static word based compressor. This is due to a feature of the Natural Language Toolkit module, which transforms quotation marks (“) to double grave accents (``). Otherwise, the output is always equal to the input.

The adaptive, standard Huffman code based encoding, was not included in the performance studies because its execution time was too long. Also, as the result will show, the Vitter algorithm based solution is very close to the standard Huffman coding in performance, and its execution time performance is far superior to the slow adaptive version.

3.5.1 Performance

The performance of the implementation was studied with three randomly selected books printed in English and with one specifically selected book printed in Greek. Book titles, authors, encodings, and file sizes are shown in Table 2. UTF-8 encodes the Greek alphabet with two bytes per character.

Table 2. The books used in performance study.

File	**Book title**	**Author**	**Language**	**Character encoding**	**File size [bytes]**
File1	Rhetoric and Poetry in the Renaissance - A Study of Rhetorical Terms in English Renaissance Literary Criticism.	Donald Lemen Clark	English	UTF-8	310620
File2	Sermons on Evil-Speaking	Isaac Barrow	English	US-ASCII	251816
File3	Miscellaneous Essays	Thomas de Quincey	English	ISO-8859-1	426855
File4	Parva Naturalia (Little Physical Treatises), Vol. 1	Aristoteles	Greek	UTF-8	296742

Table 1 shows the total number of symbols per file. It also shows the entropies, the average codeword lengths, and the compressed file sizes. Only the static methods were considered here, and the compressed size is the padded size without the header. Based on the optimality criteria defined in Equation 4, the Huffman coding is shown to be very close to the lower bound, but it cannot reach it because the probabilities for the codewords are not negative powers of two. This was predicted in chapter 2.1.

Even though the entropies for the character based encodings are smaller than the ones for the word based encodings, the total counts of symbols are smaller for the word encodings, and therefore, their compressed sizes are smaller. The entropies are larger because the alphabet sizes with word based encodings are larger than the ones with character based encodings.

Table 3. Statistical performance metrics.

File	Compression method	Total count of symbols	Entropy	Average codeword length	Compressed size [bytes]
file1	Static - characters	304262	4.63	4.66	177071
file1	Static - words	110957	6.26	6.31	87451
file2	Static - characters	247507	4.42	4.46	137883
file2	Static - words	95096	5.91	5.93	70528
file3	Static - characters	420310	4.50	4.53	237943
file3	Static - words	159057	6.32	6.36	126495
File4	Static - characters	175678	5.23	5.26	115480
File4	Static - words	64736	6.30	6.34	1279

Table 4 presents the space-saving metrics for the different compression methods. The compressed size is the padded size. As is seen from Table 4, the static compression methods have good performance for texts printed in English. However, the fascinating result is the one for the text printed in Greek. With word based encoding it achieved a saving percentage of 99.5%. On the other hand, with character based encoding, the saving percentage was smaller than with English texts. This is because the entropy of the Greek text is larger; the alphabet of actually used characters is larger for the Greek text (156) than for the English text (131).

The performance of the precalculated code dictionaries was good. This is mostly because the implementation can reduce the number of text characters that have to be encoded individually. Pregenerated codes can be too general; The 1000 most common words might not represent the vocabulary of an author or translator, or subjects covered in a book might need a vocabulary that is a complete mismatch. However, in this case the match was good. As the word coding dictionary size is increased, the compressed file size starts to decrease. Since the code need not be transmitted, the multiple dictionary method can be considered superior to the standard Huffman code based static method with character symbols.

The performance of the Vitter algorithm based implementation was studied with one book in English and with the book in Greek. By comparing the compressed sizes to the static compression, it can be seen that the sizes are almost equal. However, the adaptive method does not need to transmit the code to the decompressor. Therefore, based on the results, the Vitter algorithm based adaptive Huffman coding is superior to the standard Huffman code based static method with character symbols.

Table 4. Space-saving performance metrics.

File	Compression method	Compression ratio	Saving percentage [%]	Compressed size
File1	Static - characters	0.57	43.0	177071
File1	Static - words	0.28	72.0	87451
File2	Static - characters	0.55	45.2	137883
File2	Static - words	0.28	72.0	70528
File3	Static - characters	0.56	44.3	237943
File3	Static - words	0.30	70.0	126495
File4	Static - characters	0.39	61.2	115480
File4	Static - words	0.0043	99.5	1279
File1	Static - precalculated freq.	0.54	46.0	166537
File1	Adaptive - Vitter	0.57	43.0	177288
File4	Adaptive - Vitter	0.39	61.0	115761

Table 5 presents the average CPU (central processing unit) times for code generation, encoding and decoding. The static character based method has smaller CPU time for the code generation than the word based method. This is because the alphabet of the word based method is much larger. However, the larger alphabet resulted to smaller encoding and decoding times. The static predefined dictionaries method had the same execution times as the static character based method but it did not have the code generation overhead. Therefore, it shows the advantage of the reduced complexity of the predefined methods. In comparison to the static methods, the adaptive Vitter code based method had a long CPU time, which is due to the inherent complexity of the dynamic Huffman trees. The longer encoding than decoding time is because the code generation from a leaf to the root requires more operations than using a code to traverse from root to a leaf node.

Table 5. The average total CPU times.

Compression method	Code generation	Encoding	Decoding	Total
Static - characters	91 ms	1.83 s	38 s	39.921 s
Static - words	11.7 s	695 ms	18.8 s	31.195 s
Static - precalculated freq.	-	1.83 s	38 s	39.83 s
Adaptive - Vitter	-	81 s	52.5 s	133.5 s

4 DISCUSSION

When the source data contained large amounts of redundancy, the performance of the static and preprocessed Huffman source coding systems were impressive. The performance and the amount of inherent redundancies were shown to be dependent on how, in comparison to the original representation, the source data symbols were modeled in a compression system. In Equation 6, the redundancy was defined as the difference between entropy and average codeword length. As the results showed, when comparing the redundancies, the number of symbols per a file accounted for more than the small difference in entropies. The insight learned was that, as the granularity is decreased, the average codeword length increases, and that the symbol model must find enough redundancy that small codewords can be used.

Another insight learned was that compression systems should adapt to the source data type. For example, if a source file contains two books, one written in English and the another in Greek, the compressed size of the combined file is larger than if the books were compressed individually in their own files. In this scenario, the Latin alphabet was more dominant, which increased the codeword length for the Greek alphabet. Therefore, a modern compression system should incorporate artificial intelligence for better results.

The preset dictionaries performed well, but, if more compression is wanted, the word dictionary size should be larger. However, since no code is transmitted with the compressed data, the preset dictionaries method, even with small dictionaries, is better than methods where the code is transmitted to the decompressor. Perhaps the preset dictionaries are the best solution for specific situations where a very limited vocabulary is used. For example, text messages.

The most challenging part of this thesis was the implementation of the Vitter algorithm. Even though binary trees are common, the Vitter algorithm imposes strict conditions on the data structure. However, the challenge was worth it because the results were exceptional. The adaptive system was able to compress the source data with minimal loss when compared to the standard Huffman code.

These lead to an insight that more advanced compression systems must be a combination of preset dictionaries and adaptive algorithms. The key is that no code should be transmitted. Also, artificial intelligence could be used to gain better results with diverse source data.

The Huffman source coding is relatively simple, but it presents the fundamentals that are needed for more advanced methods. First, the underlying statistical models for a source data stays the same even if the source coding Algorithm is changed. Second, the standard Huffman code is close to being optimal, and with static encoding, it can be used as a benchmarking tool. Third, the multiple preset dictionaries and the adaptive coding are methods that are used to build more advanced compression algorithms. Also, there are other parts than the encoding and decoding in a complete compression system; compressors need interfaces that pre- and post-process uncompressed and compressed data. Therefore, the work also gave an insight into separation of concerns: how a compression system should be partitioned so that most of the source code can be reused.

5 SUMMARY

This work was an introduction to the science of source coding systems. First, the general theory of the source coding was studied. Then, Huffman source coding and different methods to improve it were considered. After the theory of the algorithms were complete, they were implemented and tested using books with different character encodings.

A source coding system consists of data, encoder and decoder. The effectiveness of a code depends on the model of the data. In this work, the model considered was occurrence frequencies of symbols. The main result was that any possible code that can be generated for a model is lower-bounded by the entropy of the source data; without introducing loss, no source code can have an average codeword length smaller than the entropy.

Huffman codes are variable length and uniquely decodable prefix codes. By using a frequency model, the standard Huffman coding can generate a code that is close to the lower-bound. Since the standard Huffman code requires the frequency data at the decoder also, at least the shape of code must be transmitted together with the compressed data. In order to improve this, two methods were introduced: preset dictionaries and adaptive algorithms. The preset dictionaries use precalculated frequency models to generate codes which are known both at encoder and decoder. The adaptive methods generate codes by first reading a symbol from the input, then updating the frequency model and finally generating a new code. The standard Huffman code can be used as the update method but is slow. Alternative is to take advantage of the sibling property of the Huffman trees. This leads to the Vitter algorithm, which updates the frequency model and the code simultaneously. Its biggest advantage is the ability to only update parts of the Huffman tree, leading to improved execution times.

Standard Huffman code, preset dictionaries method, and adaptive algorithms were implemented using Python programming language. The implementation can process different text character encodings as inputs. Also, it can use either text characters or words as symbols. The preset dictionaries were implemented by first conducting a study of word frequencies. The study considered over 1000 books in English and counted every occurrence of every word and text character. The result was a preset word dictionary of the 1000 most common words and a text character dictionary of the 100 most common text characters.

The best space-saving performance was with word based static Huffman code, because it had the best frequency model of the data. With a text printed in Greek, the size of the reduction was 99.5% of the initial size. The preset dictionaries method and the Vitter algorithm based adaptive method outperformed the static Huffman code. The preset dictionaries method resulted in compressed file sizes smaller than the static method, and the Vitter algorithm had only a small loss compared to the static method. Together with having no need to transmit the generated code, they were superior to the static method.

6 REFERENCES

[1] Cortada, J. (2006) The ENIAC's influence on business computing. IEEE Annals of the History of Computing, Vol. 28, Issue 2. p. 26-28.

[2] Atzori L. & Iera, A. & Morabito, G. (2010) The Internet of Things: A survey. Computer Networks, Vol. 54, Issue 15. p. 2787-2805.

[3] Severance, D. (1982) A practitioner’s guide to database compression. 1983 Information Systems. Vol 3, Issue 1. p. 51-62.

[4] Pu, I. (2006) Fundamental Data Compression. (1.ed), p. 11-98.

[5] Gleave, A. & Steinruecken, S. (2017) Making compression algorithms for Unicode text. arXiv:1701.04047. p. 1-10.

[6] Abhishek, K. & Syed, M. & Vipin, K. (2011) English text compression for small messages. ICIMU 2011: Proceedings of the 5th international Conference on Information Technology & Multimedia. p. 1-5.

[7] Vitter, J. (1987) Design and Analysis of Dynamic Huffman Codes. Journal of the Association for Computing Machinery, Vol. 34, No. 4, p. 825-845.

[8] asciitable.com (Accessed 14.6.2021) ASCII Table and Description. URL: http://www.asciitable.com/

[9] kb.iu.edu (Accessed 14.6.2021) The differences between ASCII, ISO 8859, and Unicode. URL: https://kb.iu.edu/d/ahfr

[10] ietf.org (Accessed 14.6.2021) (2003) UTF-8, a transformation format of ISO 10646. URL: https://www.ietf.org/rfc/rfc3629.txt

[11] Shanmugasundaram, S. & Lourdusamy, R. (2011) A Comparative Study Of Text Compression Algorithms. 2011 International Journal of Wisdom Based Computing. Vol. 1. p. 68-76.

[12] bitstring.readthedocs.io (Accessed 14.6.2021) Manual. URL: https://bitstring.readthedocs.io/en/latest/

[13] nltk.org (Accessed 14.6.2021) nltk.tokenize.api module. URL: https://www.nltk.org/api/nltk.tokenize.html

[14] gutenberg.org (Accessed 14.6.2021) Free eBooks. URL: https://www.gutenberg.org/