UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Artjom Šiškov

Development of the text archivers using linguistic features of the language

Bachelor's Thesis (9 ECTS)

Supervisor:
Irina Bocharova, PhD

Tartu 2025

Development of the text archivers using linguistic features of the language

Abstract:

This thesis focuses on the process of the developing the text archiver that uses linguistical features of the language, such as word endings, digraphs, prepositions etc. The main idea is to develop the archiver with pre-made dictionary that holds most commonly used words or parts of the words from given language. The thesis has four parts. The first part introduces the reader to the main idea. The second part has the information about various compression algorithms and necessary preliminaries. The third part is about the developed archiver itself and the testing results, where the idea of the pre-made dictionary is proven to be effective. The fourth part concludes all the thesis in general and describes the future plans for the development and improvement of the archiver.

Keywords: Algorithms, compression, lossless compression, archiver, archiving, linguistics, text archiving, LZW algorithm, LZ77 algorithm, LZ78 algorithm, Huffman algorithm, arithmetic coding, Shannon algorithm, Shannon-Fano-Elias algorithm, development, Russian language, English language, Estonian language, entropy, information content, codeword

CERCS: P170 Computer science, numerical analysis, systems, control P175 Informatics, systems theory P160 Statistics, operation research, programming, actuarial mathematics

Tekstiarhiveerijate arendamine kasutades keelelisi iseärasusi

Kokkuvõte:

See lõputöö keskendub tekstiarhiveerija arendamise protsessile, mis kasutab keele keelelisi iseärasusi, nagu sõnalõpud, digraafid, eessõnad jne. Põhiidee on arendada arhiveerijat eelnevalt valmistatud sõnastiku abil, mis sisaldab antud keelest kõige sagedamini kasutatavaid sõnu või sõnaosi. Lõputöö koosneb neljast osast. Esimene osa tutvustab lugejale põhiideed. Teine osa annab teavet erinevate tihendusalgoritmide ja vajalike ettevalmistavate teadmiste kohta. Kolmas osa käsitleb väljatöötatud arhiveerijat ennast ja testimistulemusi, kus eelnevalt valmistatud sõnastiku idee osutus tõhusaks. Neljas osa võtab lõputöö üldiselt kokku ja kirjeldab arhiveerija arendamise ja täiustamise tulevikuplaane.

Keywords: Algoritmid, tihendamine, kadudeta tihendamine, arhiveerija, arhiveerimine, keeleteadus, teksti arhiveerimine, LZW algoritm, LZ77 algoritm, LZ78 algoritm, Huffmani algoritm, aritmeetiline kodeerimine, Shannoni algoritm, Shannon-Fano-Eliase algoritm, arendus, vene keel, inglise keel, eesti keel, entroopia, infosisaldus, koodsõna

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria) P175 Informaatika, süsteemiteooria P160 Statistika, operatsioonanalüüs, programmeerimine, finants- ja kindlustusmatemaatika

Contents

1. Introduction

Today, people use computers for various purposes: studying, working, storing data, reading texts, etc. In all these cases, text is one of the most important data types. Furthermore, text files are frequently compressed using different archivers. K. Sayood in his book [1] writes that data compression algorithms are used to reduce the number of bits needed to represent the compressed file. In other words: ”data compression is the art or science of representing information in a compact form” [1]. Archivers rely on algorithms that process individual symbols or entire words [2]. However, algorithms that do not take into account the dependencies between various word components miss the opportunity to improve the efficiency of compression algorithms [2].

To determine how words should be partitioned into components, people usually use linguistic studies. Linguistics helps identify unique features and rules present in human languages, such as grammatical endings, suffixes, prefixes, and postpositions, which are essential for proper language use. Their regularity and predictability may enhance the compression process.

For instance, J. Ševčík and J. Dvorský demonstrated in their research [3] on Czech language text compression that each word is classified as either ”inflected” or ”inflexible”. Inflexible words remain unchanged, while inflected words are morphologically analysed and divided into a lemma (the base form) and a morphological tag containing symbols that represent grammatical categories such as gender, number, tense, and case. They used up to 16 morphological categories, which may vary depending on the language. The following example from Estonian language can be given: base form - ’iseärasustega’, lemma - ’iseärasus’, morphology categories - plural, comitative case (kaasaütlev kääne).

Another approach was presented by I. Akman and his team [2], who analysed text compression by dividing Turkish words into syllables. Since the formation of syllables in Turkish follows predictable rules, incorporating this knowledge into compression significantly reduced file sizes. Akman and his team compared the efficiency of various algorithms, including Huffman and Lempel-Ziv-Welch algorithms. In the same research, it was mentioned that their own algorithm is suitable for compressing texts in any language with rich morphology (which includes Estonian). Their algorithm performed better as the text size increased, achieving a compression percentage ((original file size - compressed file size) / original file size) improvement from 13% up to 43.2%.

The aim of this thesis is to develop an archiver that is based on one of the lossless compression algorithms and uses language peculiarities in order to improve the performance of the compression process. But before getting to the implementation of the archiver itself, in the Overview of data compression techniques 2 part the reader will be introduced to necessary preliminaries of the lossless compression and some lossless compression algorithms which operating principle are explained and shown by example. Archiver 3 part of the thesis is about the implementation and development process of the author's archiver. Finally in the Conclusion part 4 the whole thesis is summarised and the future plans for the development and improvement of the archiver are described.

2. Overview of data compression techniques

2.1 Preliminaries

Before overviewing basic algorithms, some necessary preliminaries have to be defined.

In this thesis compression ratio (R) is defined as a ratio between compressed file size (F_c) and original file size (F_o)

$$R = F_c/F_o * 100\%.$$

Source of information is a set X of messages x with assigned probabilities $p(x)$[4]

$$x \in X \qquad X = \{x, p(x)\}.$$

Self-information or information content $I(x)$ shows an amount of information contained in the message x. It can also be interpreted as quantifying the level of a priori uncertainty of x

$$I(x) = -\log_2 p(x).$$

The information content is calculated by simply finding the logarithm of the message x probability. The base of the logarithm may be changed in order to fit the answer for other purposes, but here logarithm base is set on 2 so the outcome is calculated in bits.

The aim of any archiving algorithm is to reduce the amount of memory needed for the storage. So, one of the main indicators of the efficiency of the archiving algorithm is the average codeword length that can be described using the following formula:

$$\bar{l} = \mathrm{E}[l_i] = \sum_{i=1}^{M} p_i l_i$$

Here the source is $X = \{1, ..., M\}$ that contains letters with probabilities $\{p_1, ..., p_M\}$. The code is $C = \{\boldsymbol{c}_1, ..., \boldsymbol{c}_M\}$ with codeword lengths $l_1, ..., l_M$.

The main criterion for lossless coding is the source entropy. C. Shannon in his work [5] describes "The Entropy of an Information Source". In this case, an information source is a text file (usually a fiction written in different languages). The entropy of it is the unpredictability of the occurrence of each source symbol, or in other words, the entropy characterises the minimum possible average codeword length [6]. Thus, the entropy shows the amount of information per symbol of a source and is the ideal average codeword length that every efficient lossless encoding technique strives for. The information content mentioned above $I(x)$ is closely related to the idea of entropy,

because the entropy represents the average information content of the whole source. The formula for the entropy of the source is the following:

$$H(X) = -\sum_{x \in X} p(x) \log_2 p(x),$$

where $H(X)$ is the entropy of a source X, M is a number of symbols in the source and $p(x)$ is a probability of a symbol x. The efficiency of the compression algorithm can also be measured by its coding redundancy:

$$r = \bar{l} - H(X),$$

here, r - coding redundancy, $\bar{l}$ - average codeword length and $H(X)$ - entropy. In essence, coding redundancy shows the number of redundant bits per amount of information. Coding redundancy can be reduced by using block lossless coding, because the redundancy of block coding tends to zero as the block length tends to infinity. However, increasing block length leads to increasing coding complexity.

The lossless coding, also known as entropy-coding, is a type of coding where the compressed bitstream after reconstruction (decoding) will coincide bit-by-bit with the original input file [6]. When talking about encoding the text files lossless coding is basically the only acceptable option here, because no one wants to after decompressing their files find out that some bits (or even symbols) were omitted for the sake of better compression rate.

B. Kudryashov in his book [4] mentions a fair amount of various algorithms with different efficiency indicators. In order to successfully encode a text that consists of various length symbols, using only binary coding, an algorithm must utilise an unambiguous coding method: thus encoding every symbol to a codeword without using any additional separation symbols [4]. ”A code is unambiguously decodable when any symbol sequence from *A* is broken into separate code words in only one way” [4].

Unambiguous decoding is provided by the prefix property. In other words, the prefix code is a solution to this problem. A code has a prefix property if every codeword in it cannot be found at the beginning of any other codeword. Having a prefix property means that the code is unambiguous, but not every unambiguous code is prefix [4]. Usually a prefix code can be displayed as a binary tree structure, where every leaf corresponds to a symbol with its encoding and branches (every node has two outgoing ones) are labelled by a code symbols: 0 or 1.

After taking all of this information into the account the question may appear: ”Was the optimal solution lost because the class of unambiguously decoded codes was narrowed, amongst which the best is being sought was narrowed?” [4] The answer is proven to be negative.

A necessary and sufficient condition of the existing prefix code with M codewords with length $l_1, ..., l_M$ can be formulated as Kraft's inequality.

$$\sum_{i=1}^{M} 2^{-l_i} \leq 1,$$

where M is the prefix code size and $l_1, ..., l_M$ are the codeword lengths.

2.2 Classification of the compression algorithms

There exists a wide variety of compression algorithms (see, for example: [2], [7], [8], [5]). Various compression algorithms utilise different prefix encoding methods, but in general most of them can be classified into two major groups: those that use probability of each symbol(s) and the ones that rely only on source model implicitly.

The idea of the ”probability-based” algorithms is quite simple: in order to achieve a minimal average codeword length, each symbol codeword should depend on its probability within a source. That, of course, means that some codeword lengths may be different and information about symbol codes must be transmitted somehow to the decoder in order to retrieve all of the information losslessly. Here are some examples of these algorithms: Huffman algorithm, Shannon algorithm and arithmetic coding.

The other group of algorithms encodes source symbols of different lengths (typically patterns in the text) by codewords of the same or variable length. This requires more computer memory usage but can help to achieve very high compression rates in comparison with algorithms that use symbol probabilities, especially if the amount of data that needs to be compressed is big.

Some algorithms, like Huffman algorithm and Shannon algorithm are also called symbol-by-symbol coding algorithms. As the name suggests, they encode each symbol separately, while the other algorithms use blocks of symbols to encode the source. They are called the block-coding algorithms (for example arithmetic coding and LZ* algorithms). For this archiver block-coding algorithms are more suitable for finding repeated patterns and utilising the pre-made ones using dictionary.

2.2.1 Shannon-Fano-Elias algorithm

Shannon-Fano-Elias algorithm is an example of a prefix algorithm [5] with time complexity $O(n)$ (further n in time complexity formula stands for the length of the input sequence). It utilises each symbol probability within a text. A code for each symbol is defined by getting first $l_m = -\lceil \log p_m \rceil$ from a binary representation of the cumulative probability q_m where $q_m = \sum_{i=1}^{m-1} p_i$, $m-1$ is a number of symbols preceding the symbol with number m in the alphabet and p_m is its probability [4]. The algorithm steps can be described in the following way:

1. The algorithm analyses the whole text and composes a list of symbol probabilities.
2. The list of probabilities is sorted by descending order.
3. A cumulative probability (q_m) is computed for each symbol. The cumulative probability is a sum of all probabilities of preceding symbols in the list.
4. A codeword is computed for each symbol as a length l_m binary representation of q_m, $l_m = \lceil -\log p_m \rceil$.
5. The algorithm encodes the message using the obtained codewords.

To bring more clarity in this coding technique, here is an example: Let us assume that $X = \{a, b, c, d, e, f\}$ and the corresponding probabilities are $\{0.35, 0.2, 0.15, 0.1, 0.1, 0.1\}$. An example of Shannon code constructed for this probability distribution is presented in Table 1.

After thorough analysis of this table, the average codeword length can be calculated $\bar{l} = 2.95$ and entropy $H(X) = 2.4016$, which means that the Shannon algorithm codewords have $r = 0.5484$ redundant symbols on average. When comparing two encoding algorithms, the average codeword length is one of the main indicators, as was mentioned earlier. But when it comes to the realisation

x	p_m	q_m	l_m	binary of q_m	codeword c_m
a	0.35	0.00	2	0.00...	00
b	0.20	0.35	3	0.0101...	010
c	0.15	0.55	3	0.10001...	100
d	0.10	0.70	4	0.10110...	1011
e	0.10	0.80	4	0.11001...	1100
f	0.10	0.90	4	0.11100...	1110

Table 1. Shannon code computation results

of the algorithm and its practical usage, complexity is the second most important indicator of the algorithm efficiency.

2.2.2 Huffman algorithm

Huffman coding developed in 1952 by D. A. Huffman [8] has time complexity of $O(n)$ and it was , assigns shorter codewords to more frequent symbols and longer ones to less frequent symbols. The algorithm also utilizes a binary tree structure, but the way it assigns codewords to each symbol is completely different. The Huffman algorithm steps can be defined as follows:

1. The algorithm analyses the whole text and gets each symbol probability.
2. The algorithm starts composing a binary tree as follows:
   (a) The algorithm chooses two symbols with the smallest probabilities from the list.
   (b) Two nodes are created, with values 0 and 1 arbitrarily assigned to them.
   (c) The probability of the node is the sum of probabilities of the merged nodes.
   (d) The processed probabilities are removed from the list. Instead, a new sum of the two removed probabilities is added to the list.
   (e) The algorithm starts to choose two symbols from the list again until the list has one entry left.
3. The algorithm computes each symbol codeword by travelling through the tree from the root to the leaf with a specified symbol. Going from parent to the left child node adds '0' to the codeword and adds '1' if it is the right child node.
4. The text is encoded with the obtained codewords.

Let us use the same example from the Shannon section to demonstrate the Huffman algorithm principle and put all its computations into Table 2. The corresponding binary tree is shown in Figure 1.

From these results, it can be easily calculated that the average codeword length is $\bar{l} = 2(0.35 + 0.2) + 3(0.15 + 3 \times 0.1) = 2.45$ and entropy, which stays the same, is equal to $H(X) = 2.4016$. This means that the redundancy of Huffman algorithm in this case is $r = 0.0484$. Although more than one codeword assignment is possible, because there are symbols with identical probabilities, the average codeword length will never change [4]. The average length of the Huffman code is smaller than that for the Shannon algorithm with the same input.

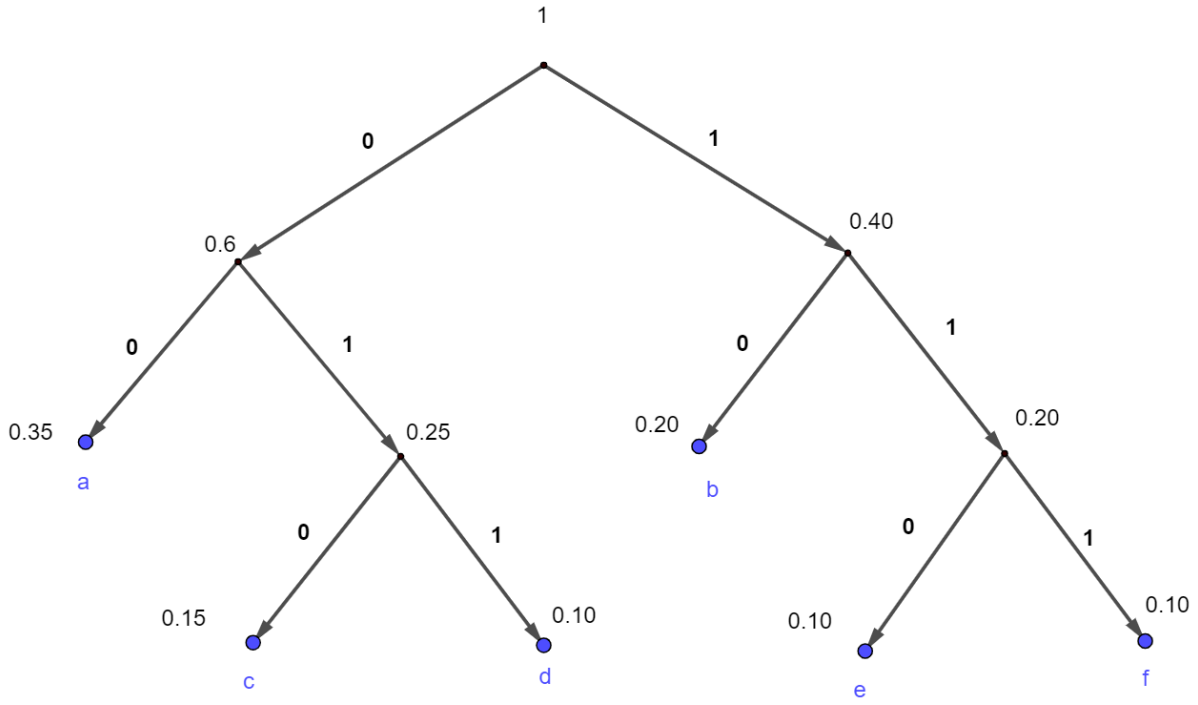

Figure 1. Huffman Binary Tree

In case of symbol-by-symbol coding, codeword lengths cannot be less than 1 bit and the average codeword length is always greater than 1. In order to reduce coding redundancy, a block-lossless coding should be used. As mentioned above, redundancy tends to zero as the block length grows. Any method of symbol-by-symbol can be generalised to block coding by considering a block of symbols as an enlarged symbol. However, the complexity of Huffman coding is proportional to the size of the new alphabet, which grows exponentially with block length. For this reason, considered in the next subsection, arithmetic coding is a generalisation of one modification of the Shannon-Fano-Elias algorithm. The latter has lower complexity than Huffman coding, but larger redundancy.

2.2.3 Arithmetic coding

Arithmetic coding is the generalisation of a Shannon-Fano-Elias coding technique. Arithmetic coding is a low complexity algorithm (it has complexity of order $O(n^2)$, but can be easily implemented with linear complexity that is of order $O(n)$) and can achieve a high compression ratio. The arithmetic coding main operating principle is using symbols probabilities to encode the entire message [6]. Here are the steps that the algorithm performs during the encoding process:

x	Codeword
a	00
b	10
c	010
d	011
e	110
f	111

Table 2. Huffman code codewords and corresponding symbols

1. The algorithm analyses the whole message and retrieves symbol probabilities $p(x_i)$, where x_i is message symbol.
2. The algorithm starts working with a $[0, 1]$ interval that is divided according to symbols probabilities.
3. The algorithm checks the first symbol and moves the interval to the symbol probability interval $(Q(x_i), Q(x_i) + p(x_i))$, where $Q(x_i)$ is the cumulative sum of all probabilities of the symbols preceding symbol x_i, $Q(x_i) = Q(x_{i-1}) + p(x_{i-1})$.
4. In addition, the algorithm computes the variables F and G for each next symbol, where $G \to G \times p(x_i)$ and $F \to F + Q(x_i) \times G$.
5. The algorithm checks next symbol and continues to narrow its interval borders.
6. Upon reaching the last symbol, the algorithm chooses the number $F + G/2$ or middle point of the interval.
7. Codeword length is $l = -\lceil \log_2 G/2 \rceil$
8. Codeword is first l numbers of binary representation of $\lfloor F + G/2 \rfloor_l$

Let six symbols long input sequence **dkekdk** be considered and let probability distribution be: $\{p(d) = 0.3, p(e) = 0.1, p(k) = 0.6\}$. Calculations are summarised in the Table 3.

The length of the codeword is $l = -\lceil \log_2 0.001944/2 \rceil = 11$. The codeword is $c = \lfloor 0.1813296 + 0.001944/2 \rfloor_{11} = \lfloor 0.1823016 \rfloor_{11} = 00101110101$. The entropy is $H(X) = 1.2955$ per symbol, so the redundancy of the code is $r = \bar{l}/n = 11/6 - 1.2955 = 0.5378$. The graphical representation of the arithmetic coding is presented below as Figure 2.

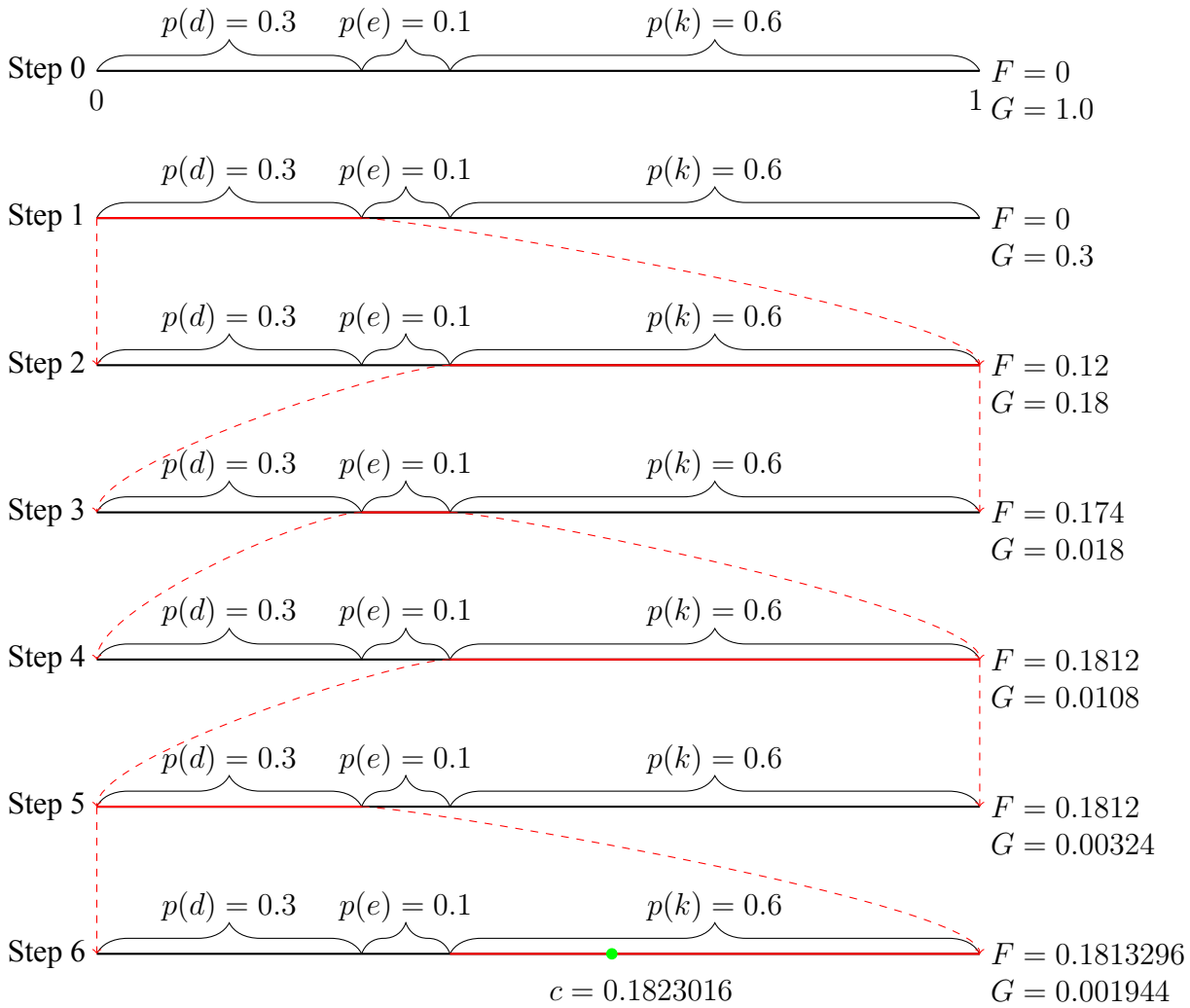

Figure 2. Arithmetic coding graphical representation

Step i	x_i	$p(x_i)$	$Q(x_i)$	F	G
0	–	–	–	0.0000	1.0000
1	d	0.3	0.0	0.0000	0.3000
2	k	0.6	0.4	0.1200	0.1800
3	e	0.1	0.3	0.1740	0.0180
4	k	0.6	0.4	0.1812	0.0108
5	d	0.3	0.0	0.1812	0.00324
6	k	0.6	0.4	0.1813296	0.001944

Table 3. Arithmetic coding calculations

2.2.4 Arithmetic decoding issues

In order to decode a codeword using arithmetic decoding, just as with encoding, the probability distribution on the source symbols is required. It is already an issue by itself because conveying this information together with the codeword itself might be troublesome. The classical arithmetical decoding algorithm looks like the following:

1. The input of the algorithm is: number of symbols n, number representing encoded data $\hat{F}$ and symbols probabilities $p(0), p(1), ..., p(M-1)$ (M is a size of an alphabet).
2. The algorithm additionally initialises S and G variables and cumulative function variable $Q(M) = 1$.
3. For every decoded symbol the following steps must be made:
   (a) The algorithm checks if $S + Q(j) \times G < \hat{F}$, where j is an index of a symbol, starting from the first one.
   (b) If not, then the algorithm takes the next symbol and repeats the previous step.
   (c) If yes, then the algorithm adds the symbol with the index j to the output and increments S by $Q(j) \times G$ and multiplies G by $p(j)$.
4. When all n symbols are decoded, the output message is returned.

As can be seen, the arithmetic coding has some problems. Although computational complexity is proportional to the squared length of the input sequence, the procedure looks completely impractical [6]. The main problems are the following:

- Theoretically, the arithmetic coding algorithm requires infinite computational precision in order to correctly encode/decode the message.
- The coding delay is equal to the length of the input message.

Nothing stops the computer from infinitely encoding an infinite message. Except nowadays computer architecture has computational limitations and that is defined by whether the computer soft- and hardware have 64-bit, 32-bit or 16-bit architecture. Naturally, having arithmetic coding implemented in the 32- or 16-bit architecture will physically limit the computer from making heavily encoded outputs [6]. The solution of the problem above was introduced and implemented by I. H. Witten, R. M. Neal, and J. G. Cleary in their work [9]. Nowadays, arithmetic coding can be implemented on 16-bit and 32-bit computer systems.

There is also a group of algorithms that do not require a knowledge about the probability distribution on the input symbols. They belong to so-called universal coding algorithms and Prediction by Partial Matching (PPM) algorithm is one of these [10]. The idea behind this algorithm is to use arithmetic coding with estimated conditional symbol probabilities. These conditional probabilities are computed based on the ”context”, that is, already encoded symbols preceding a symbol to be encoded. The maximal length of the context can be chosen taking into account complexity issues. As it was mentioned earlier, in this thesis the focus will be on the universal algorithms of lower complexity. They are based on constructing dictionaries by using the text to be compressed.

2.2.5 LZ77 algorithm

The LZ* algorithm family is a set of prefix compression algorithms [4]. The initial algorithm inventors: A. Lempel and J. Ziv, their surnames gave the name to the whole family of algorithms [11]. During the process many modifications of the initial algorithm were made. The first algorithm in the family is LZ77. LZ77 is also sometimes called the *sliding window* compression algorithm because its main feature is the dictionary that stores all possible subsequences of the last W processed symbols, where W is a window size. The dictionary size is predefined and it usually cannot store all of the text [4]. Time complexity of this algorithm varies from $O(n^2)$ (worst-case complexity) to $O(n)$ (best-case complexity). Here is the step-by-step algorithm explanation:

1. The algorithm initialises an empty dictionary W.
2. The algorithm encodes the first symbol.

3. After that, the algorithm starts to encode every next symbol by the following scheme:
   (a) The algorithm takes the next symbol (or sequence) and seeks it within the W dictionary.
   (b) If the subsequence is found, the algorithm sets a flag '1' and encode the distance between the symbol (subsequence) and the corresponding dictionary codeword by rounded up $\log_2 W$. Then the length of the coincidence is encoded by a variable length code.
   (c) If the subsequence is not found, the algorithm sets flag '0' and the new symbol is encoded by rounded up $\log_2 M$, where M is the alphabet size. The symbol is then added to the W dictionary.
4. The algorithm returns the encoded text.

Even though the algorithm novelty and efficiency were high there were some flaws: the W dictionary capacity limit and the fact that because of that the information encoded after the '1' flag that leads to the other subsequence is limited by W dictionary size, thus making it impossible to put links to every part of the processed text. The redundancy of this algorithm depends on the amount of the repeated patterns in the source. If there is none, the algorithm will encode each symbol without any flag attaching. Here is a simple example of LZ77 algorithm. Let the input source is '**parallel**' and the initial alphabet size is 16.

- The first symbol 'p' is encoded as '0' + '0001'.
- The symbol 'a' is encoded as '0' + '0010'.
- The symbol 'r' is encoded as '0' + '0011'.
- Another symbol 'a' is encoded as '1' + '$\log_2 3$' + '10'. Because 'a' was found earlier the '1' flag was encoded together with $\log_2 3$, where 3 stands for the current size of the dictionary and distance '10' (2) to the preceding coincidence.
- The symbol 'l' is encoded as '0' + '0100'.
- The next symbol 'l' is encoded as '1' + '$\log_2 4$' + '1'. Because 'l' was found earlier the '1' flag was encoded together with $\log_2 4$, where 4 stands for the current size of the dictionary and distance '1' (1) to the preceding coincidence.
- The symbol 'e' is encoded as '0' + '0101'.

- The next symbol 'l' is encoded as '1' + '$\log_2 5$' + '10'. Because 'l' was found earlier the '1' flag was encoded together with $\log_2 5$, where 5 stands for the current size of the dictionary and distance '10' (2) to the preceding coincidence.
- Algorithm finishes its work.

The realisation of the LZ77 may differ to fit different purposes, this example was simplified because of the small amount of information encoded. The approximate redundancy for LZ77 can be computed using the following formula: $r = \frac{\log_2 \log_2 W}{\log_2 W}$, where r is redundancy and W is window length [4]. Thus, the redundancy for window with length 8 is $r = \frac{\log_2 \log_2 8}{\log_2 8}$, $r = 0.5283$ bits.

2.2.6 Lempel-Ziv-Welch algorithm (LZW)

The LZ-78 algorithm from the LZ* algorithm family turned out to be more practical than its predecessor LZ-77 (they both have the same time complexity) [12]. For example, it was the first algorithm used for compression in the V40bis standard for public telephone channels [4]. After the T. A. Welch [13] modification (where from the algorithm name got its last letter 'W') the algorithm became truly elegant and effective.

The LZW algorithm builds a dictionary of repeated symbol sequences. There are many different algorithms from the "LZX" family, where 'X' is usually replaced with some other letter or number in order to designate an algorithm modification. Usually the LZW algorithm from the start already has a dictionary that consists of all possible symbols. But in this example 'all possible symbols' are just all unique symbols from the source. Here is a brief explanation of the LZW algorithm operating principle:

1. The algorithm has all source symbols codeword values preassigned in its dictionary.
2. The first symbol is encoded normally as a single one, using the codeword from the dictionary. Then the algorithm begins encoding the text by following scheme:
   (a) The next symbol is obtained from the algorithm dictionary and encoded. Encoding can be performed differently depending on the aim of the archiving (for example using $\log_2$ (dictionary index)) but in this case every binary number codeword has fixed length and the dictionary size increments for every encoded unique symbol or sequence.

(b) A new input is added to the dictionary, which consists of the previous symbol and the current symbol.

3. The algorithm continues to add new inputs to its dictionary while it is still possible.
4. The fully encoded text is returned.

Initially, the dictionary contains single-character symbols with pre-made codewords, but as the algorithm processes text, it adds multi-character sequences, improving compression efficiency. The encoder and decoder form the same dictionary of extra symbols in the processes of encoding and decoding an input [4]. Assuming that the algorithm has a full set of single-character symbols in its dictionary, then it means that the algorithm will add a new unique symbol sequence in its dictionary, giving it its own codeword. Because the decoder does not know about the existence of the first-met symbol sequence, the algorithm is denied from using the last input word from its dictionary [4].

Let us review an example of the LZW algorithm work in practice. Let the input message be **"abracadabra"** and the symbols corresponding codewords are stored as in dictionary from Table 5. The found sequences are stored in the same dictionary for later use as shown in Table 6. All codewords will be 5 bits long each, but the number of bits may vary depending on the length of the message. The algorithm will encode the message as shown in Table 4.

Processed message	Output codeword	Added sequence
abracadabra	00000	-
a**b**racadabra	00001	ab
ab**r**acadabra	00100	br
abr**a**cadabra	00000	ra
abra**c**adabra	00010	ac
abrac**a**dabra	00000	ca
abraca**d**abra	00011	ad
abracad**ab**ra	00101	da
abracadab**ra**	00111	-

Table 4. Demonstration of the LZW algorithm

As the result the algorithm added eight new sequences to its dictionary but used only two of them. This brings us to the problem that not every added sequence is rational to keep in the

Symbol	Codeword
a	00000
b	00001
c	00010
d	00011
r	00100

Table 5. Initial dictionary

Added sequence	Codeword
ab	00101
br	00110
ra	00111
ac	01000
ca	01001
ad	01010
da	01011

Table 6. Dictionary of the new inputs

dictionary as it may not appear in the message again. In addition, when planning this text archiver development, the LZW algorithm looks like an obvious choice, because it is possible to pre-assign all needed character symbols that are presumably the most frequent sequences in one language and significantly speed up the encoding process. With this modification, the LZW algorithm may show the best archive efficiency.

2.3 Compression techniques based on language linguistic features

2.3.1 Text corpora

A critical component in text compression research is the use of text corpora, collections of linguistic data used for analysis. These corpora consist of various textual sources, such as literature, academic papers, newspapers, textbooks, and letters. Testing text compression algorithms without appropriate corpora is impossible, making their selection an essential part of developing archivers.The effectiveness of an algorithm often depends on the data volume used in testing, as was proven in study [2]. In Estonia, linguistic data collection is carried out by institutions such as the Institute of Estonian Language and the Computational Linguistics Research Group at the University of Tartu.

2.4 Algorithms used in previous works

As it was mentioned in the Introduction 1, Akman and his team [2] compared their own algorithm with LZW algorithm and Huffman algorithm and found out that the Huffman coding achieved a stable compression rate of around 31–35%, while the LZW compression varied from 35% to 54%. However, LZW generally required less processing time than Huffman coding. The Akman's algorithm was using a completely different way of encoding text files. By using Turkish

language grammar and syntax rules and block coding, where blocks were usually syllables, great results were achieved: SA algorithm, how they called it, had from 13% up to 40% compression percentage but at the same time losing in compression time spent during the process. The Ševčík and Dvorský in their work [3] were using tree data structure called *Treex* for analysing the text linguistically, then the LZ algorithm was used for encoding the lemmas of the words that carry the main information. The results, however, showed only from 74% to 70% compression ratio. This study aims to use both algorithms to determine which is best suited for Estonian, English and Russian texts compression.

3. Archiver

Archiver is a software that composes archives from the set of files by compressing their size. One of the most popular archivers today are: ZIP, 7-Zip and WinRAR. In addition, ZIP archive software is pre-installed in Microsoft operating systems, such as Windows, while WinRAR and 7-zip must be installed manually from their website. However, modern archivers were developed after years of programming and finding the best solution by combining different compression algorithms and usually having a lot of personalisation options (for example options for archive type, compression ratio, compression speed etc.). Therefore, this thesis algorithm will be compared with the common LZW compression algorithm. The comparison will be conducted by compressing various text files and such indicators as compression ratio and time spent for compression and decompression will be taken into account.

3.1 The Idea

As it was mentioned earlier the main idea was to develop an archiver which will utilise language peculiarities in order to enhance algorithm efficiency and achieve better compression results. As for the programming language the Java was chosen as the most promising candidate due to its object oriented programming features and author's great skills in this language.

3.1.1 The Prototype

Originally, the Huffman coding was seen as the most suitable for this purpose - the archiver prototype was developed which had a pre-made Estonian dictionary. The prototype had a buffer for symbols and if the text had a "word" from the dictionary it was encoded as a one symbol sequence. The rest of the symbols were encoded one by one. The binary tree was composed as the result of the encoding process and this is where the problem occurs: in order to decompress the file the Huffman algorithm binary tree was needed. This means that the whole tree, which consists of all symbols from the text plus all words from the dictionary, should be somehow transmitted or stored in the same file with the encoded message. Allocating memory for storing this (usually big one) data structure was proven to be not expedient. In addition, Huffman algorithm was proven to be efficient only when compressing small files.

3.1.2 Implementation of the LZW archiver

Taking all of these factors into account it was decided to switch the main archiver algorithm to LZW with a few minor modifications:

1. This Java implementation has fixed codeword length of 16 bits or 2 bytes, because the 'Short' Java built-in class is used for storing codewords.
2. Instead of adding a new word into the dictionary by prolonging the processed word by one symbol, it was decided to make it possible to unite two words in one.
3. For this a *sliding window* or buffer was implemented that helps the algorithm to find the coincidences in the sources. Size of the buffer is dynamical and is equal to the longest input in the dictionary.

The operating principle of the LZW algorithm was explained in the "Overview" part of the thesis 2. But in order to make the archiver use the linguistic features of the language several things must be done in advance:

1. The information about the language that the text inside the file is written in must be defined by user and transmitted to the archiver before encoding.
2. The dictionary of the exact language must be composed and added to the archiver code (or in a separate file).

The system cannot define the language by itself, so the user collaboration is necessary. The idea behind the pre-made dictionary will be mentioned later in this thesis.

3.1.3 The Pre-made dictionaries

As it was already mentioned the pre-made dictionaries for each language are required for the functioning archiver. The logical question arises: "Which words should be included in such dictionaries?" The answer is rather complex, because what is added to these dictionaries is not words in most cases but only parts of them. Here is the incomplete list of possible candidates for adding to the dictionaries:

1. The diphthongs and digraphs: combinations of two (rarely three or more) letters that are common and represent in speech one sound. Examples: *'sh', 'th', 'oo', 'wh'*.
2. The most common word endings, such as *'ly', 'ed', 'ing', 'ion'*.
3. Some very common words, like pronouns, prepositions, question words, conjunctions.
4. Common word contractions. For example: *'n't', "s', "re', "m'*.

The dictionaries were usually populated by collecting data from various places (websites, books, researches, etc.) and manually entering the following data structure in code. The developed

archiver has three dictionaries for three languages, which are: English, Estonian and Russian. Estonian was the original object for studying for this thesis, but later it was decided to enlarge the field of study by including English, as the *Lingua Franca* of the modern world, and Russian, as author's native language. Estonian has a complex system of grammatical cases, Russian has a broad range of word endings, and English has a lot of grammatical tenses, which means all of them have some kind of built-in patterns, making all of them perfect candidates for a technique with pre-made dictionaries. The dictionaries itself can be found in the Appendices 4.

3.2 The Archiver manual

Even though this information is available in the open repository of the archiver, it was still decided to duplicate this manual here. The archiver is implemented using the Java programming language and its manual is written in the main Java file (LZW.java). The Java archiver implementation has only a console user interface and a built-in system of folders for all required files. The result of encoding a text file is a binary file (a file with a ".bin" extension) that has all the required information to decode the original text. "src" folder is for Java code files. "corpus_est", "corpus_eng", "corpus_rus" and "corpus_none" are for text files that are about to be archived. "bin_files" is for binary files that the archiver creates as the file with encoded text inside it. "decoded" is for all decoded files. Other folders should not be accessed by a user. The manual has translations into three languages even though the archiver console user interface is available only in English. The instructions of the archiver are the following:

1. Build and run the LZW.java file with the desired Java IDE.
2. Choose mode: 'e' for encoding and 'd' for decoding.
3. For encoding:
    (a) Navigate through the file system, starting from the "corpora" folder.
    (b) Choose the file you want to encode. Files are marked in blue, folders marked in yellow.
    (c) Choose the language the file is written in. est - Estonian, eng - English, rus - Russian and none - the language is not listed.
    (d) After encoding the ".bin" file with the same name will be created and added to the "bin_files" folder in the project. **Note:** The system will tell you the data compression ratio given in % and the time spent in milliseconds.

4. For decoding:

   (a) Navigate through the file system, starting from the ”bin_files” folder.

   (b) Choose the file you want to decode. Files are marked in blue, folders marked in yellow.

   (c) Wait until the file is decoded. All decoded files are added to the ”decoded” folder. Note: The system will tell you the data decompression ratio given in % and the time spent in milliseconds.

As was made clear in the above manual, if the user wants to encode a file, they should add it to a folder with other files written in the same language. The ”none” language option was added for those text files that are written in any other language, which dictionary is not listed in the archiver. The LZW algorithm, however, by its definition will still find repeated patterns and will try to compress the file as much as possible.

3.3 Archiving results

In order to test the archiver compression efficiency it was decided to pick 15 files written in three different languages (five files for each language) and with different sizes. English and Russian file content was copied from Wikisource webpage, which has a large variety of literature in English and Russian. Estonian corpus of literature files was found in open GitHub repository as testing the Estonian files began earlier than English and Russian. After collecting the data the archiver developed as the result of this thesis was given a prototype name ”**LZW-Ling**”. In Figures 3 and 4 show resulting diagrams of the archivers work. Additionally, it was decided to include the comparison with ZIP archiver which result diagram is in Figure 5. 15 blue columns showing the size of each file in kilobytes and orange polygonal chain showing the compression ratio (size of the file after dividing by size of the original file) in percent. In addition, Figure 6 shows the compression ratio and Figure 7 shows the compression time difference between two algorithms (ZIP compression time was not measured). For more details, please refer to Table 7, Table 8 and Table 9 in the Appendices 4.

After thorough analysis of the results the following resume can be formulated:

1. The average compression ratios of the archivers are:

   - LZW-Ling - 36.84%
   - LZW - 37.32%

- ZIP - 33.08%

2. The average compression ratios by languages are (only for LZW-Ling):
    - Estonian - 40.22%
    - English - 40.41%
    - Russian - 29.88%
3. On average, the larger the file, the more efficient the compression process is.
4. Although LZW-Ling modification is always better than LZW, the difference varies from 3% to 0.1%.
5. Both LZW and LZW-Ling showed a better compression ratio on large sized Russian language text files.
6. The average compression times are:
    (a) LZW-Ling - 5356 milliseconds.
    (b) LZW - 5761 milliseconds.

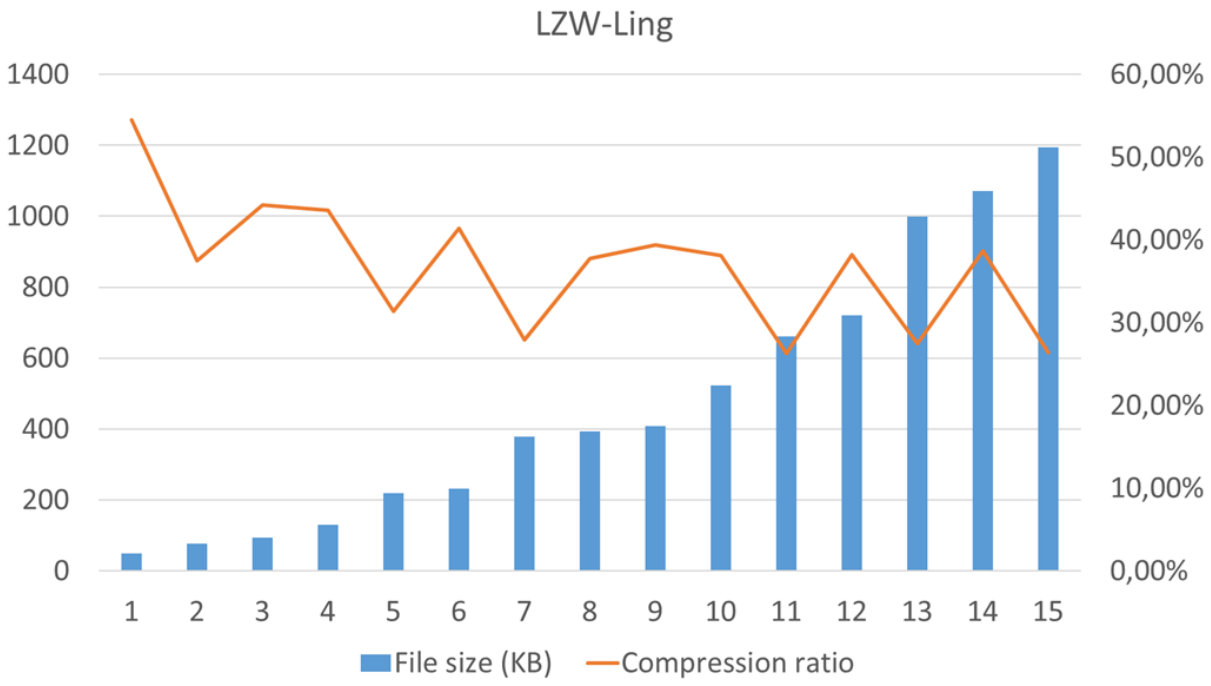

Figure 3. LZW-Ling algorithm archiver results diagram

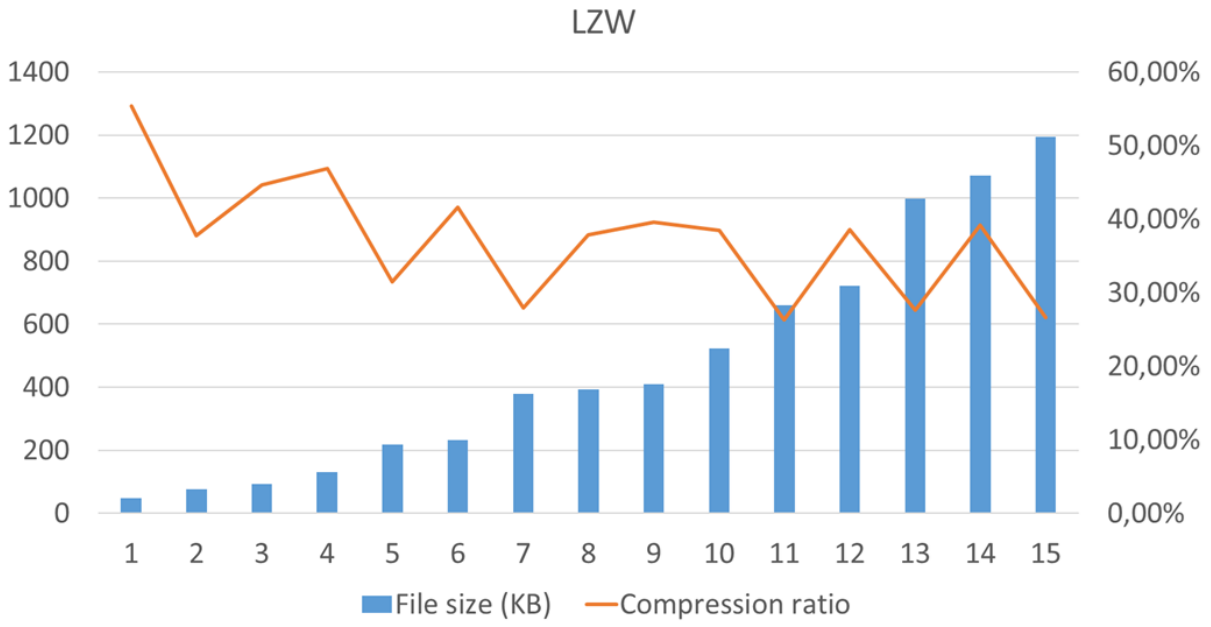

Figure 4. LZW algorithm archiver results diagram

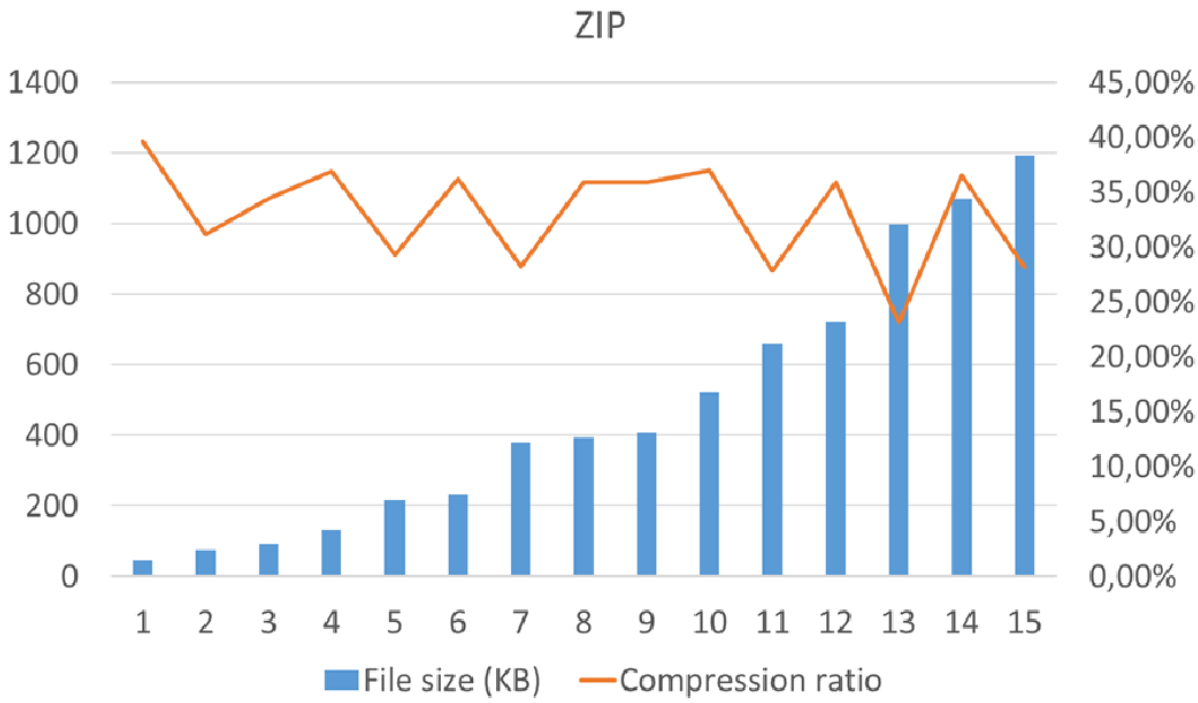

Figure 5. ZIP archiver results diagram

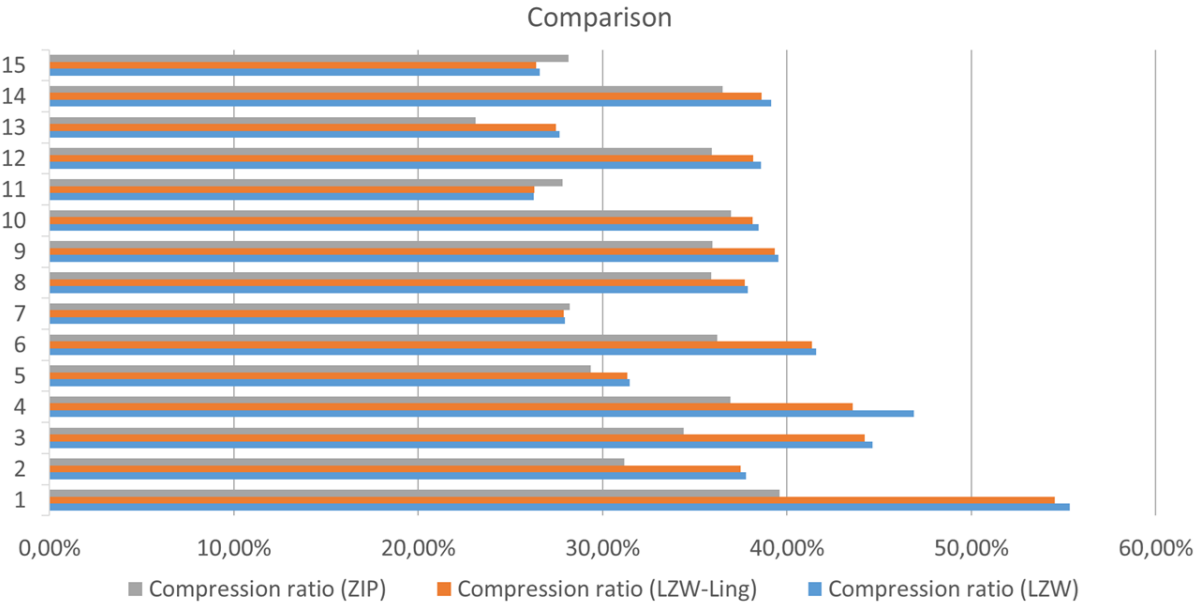

Figure 6. LZW-Ling LZW and ZIP archivers results comparison

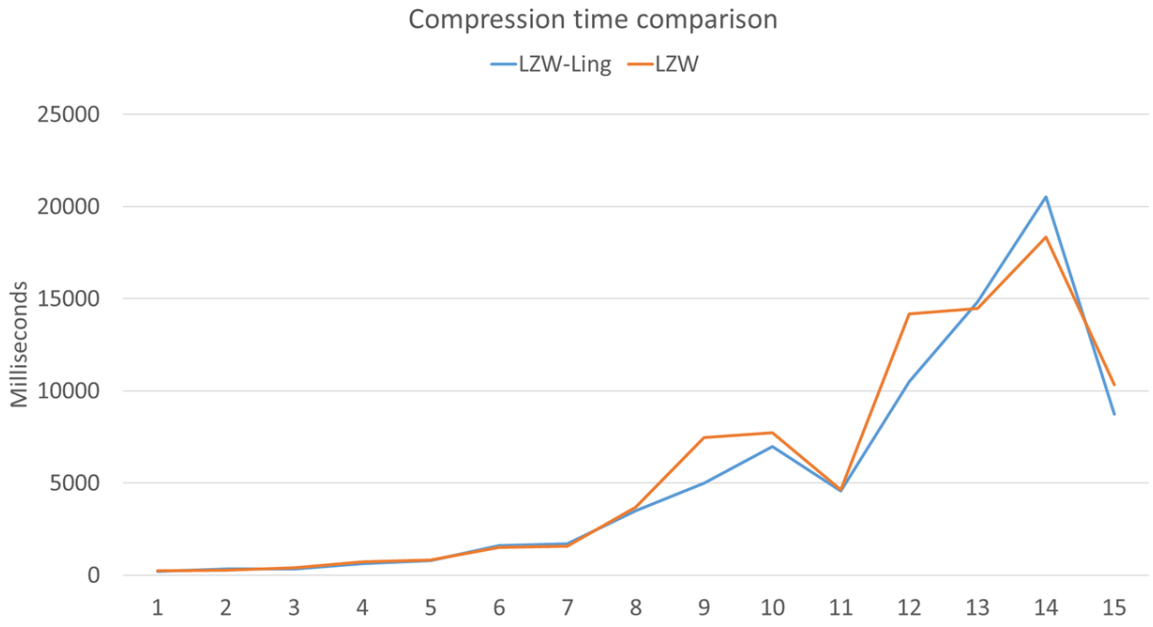

Figure 7. Compression time comparison

4. Conclusion

As the result of this thesis a Java functioning modified LZW algorithm based archiver was developed. The idea of using pre-made dictionaries was proven to be useful and LZW-Ling performed better than modified LZW. Although this experiment was successful, there are still many things to improve. For example, the current version of the archiver simply uses *Short* Java class for storing codewords. This class can hold values from $2^{15} - 1$ to -2^{15}, which severely limits the size of the dictionary. In the future, it is better to implement a code that has a dynamical codeword length depending on file size and the amount of unique symbols in the source. Another problem is the compression time of the archiver. Although the programme was rewritten from scratch multiple times during development for the sake of optimisation, going from several minutes to ten seconds max, there are still some parts that may need better realisation. Pre-made dictionaries could also have been more useful than they are in the current version. It is clear from Figure 6 that LZW-Ling has a small advantage over LZW. It suggests that pre-made dictionaries should be either enlarged or the whole algorithm should be combined with some other elements using linguistical features of the language. The last aspect of the archiver that needs to be upgraded is the algorithm it uses. The original idea was to combine multiple algorithms in one software, but the plan was reviewed in the process. Integrating the use of the symbol probabilities will surely lead to less coding redundancy.

In addition, the GitHub repository [1] was created where the archiver development process can be seen. From the same repository the archiver can be downloaded and used for various purposes, like further development or modification.

On top of the development, a lot of new information was obtained. Some basic concepts and algorithms were known before, but most of the knowledge from the Introduction 1 part needed to be studied from scratch.

[1] https://github.com/ArtemiosTLN/LZW-Ling-Archiver

References

[1] Sayood K. Introduction to data compression (2nd ed.) San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000.

[2] Akman I., Bayindir H., Misra S., Ozleme S., and Akin Z. Lossless text compression technique using syllable based morphology. *International Arab Journal of Information Technology* 8.1 (Jan. 1, 2011). https://research.ebsco.com/linkprocessor/plink?id=895ff104-8717-3cad-80b9-b326328dd819.

[3] Ševčík J. and Dvorský J. Techniques of Czech Language Lossless Text Compression. Lecture notes in computer science. Vol. 9842. Springer, Jan. 1, 2016, pp. 265–276. https://research.ebsco.com/linkprocessor/plink?id=12049878-f51c-37e9-928b-be0cdff90722.

[4] Kudryashov B. Теория информации. Питер, 2018.

[5] Shannon C. E. The mathematical theory of communication. 1963. *M.D. computing : computers in medical practice* 14.4 (July 1, 1997). Place: United States Publisher: Springer-Verlag New York Inc, pp. 306–317. https://research.ebsco.com/linkprocessor/plink?id=1393581a-a098-336c-a937-387d61482ce3.

[6] Bocharova I. Compression for multimedia. Cambridge University Press, 2010.

[7] Awajan A. Multilayer model for Arabic text compression. *International Arab Journal of Information Technology* 8.2 (Apr. 1, 2011), pp. 188–196. https://research.ebsco.com/linkprocessor/plink?id=2e99cab8-368a-3387-bbba-e54fd5de0370.

[8] Huffman D. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE, Proc. IRE* 40.9 (Sept. 1, 1952). Publisher: IEEE, pp. 1098–1101. DOI: 10.1109/JRPROC.1952.273898. https://research.ebsco.com/linkprocessor/plink?id=96f598ea-e68a-3f81-beed-c89280d95685.

[9] Witten I. H., Neal R. M., Cleary J. G., and Sibley E. H. ARITHMETIC CODING FOR DATA COMPRESSION. *Communications of the ACM* 30.6 (June 1, 1987). Publisher: Association for Computing Machinery, pp. 520–540. DOI: 10.1145/214762.214771. https://research.ebsco.com/linkprocessor/plink?id=445f913a-7ab2-3cb8-ba5c-a2bcfc6609c3.

[10] Cleary J. and Witten I. Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Transactions on Communications* 32.4 (1984), pp. 396–402. DOI: 10.1109/TCOM.1984.1096090.

[11] Ziv J. and Lempel A. A universal algorithm for sequential data compression. *Institute of Electrical and Electronics Engineers. Transactions on Information Theory* IT-23 (no. 3

Jan. 1, 1977), pp. 337–343. https://research.ebsco.com/linkprocessor/plink?id=73a8d36d-46fb-3ab9-a185-4817a3b44462.

[12] Ziv J., Lempel A., and Gray R. M. Compression of individual sequences via variable-rate coding. *Institute of Electrical and Electronics Engineers. Transactions on Information Theory* 24 (no. 5 Jan. 1, 1978), pp. 530–536. https://research.ebsco.com/linkprocessor/plink?id=bedd5b59-9c5a-33ea-aa27-4753f7e08616.

[13] Welch T. A Technique for High-Performance Data Compression. *Computer* 17.6 (June 1, 1984). Publisher: IEEE, pp. 8–19. DOI: 10.1109/MC.1984.1659158. https://research.ebsco.com/linkprocessor/plink?id=5642dcfd-55a9-3376-8c40-38a836d6e7c0.

Appendices

File nr.	Archiver	File size (KB)	Compression ratio	Time spent (milliseconds)
1	LZW-Ling (eng)	48	54.52%	210
2	LZW-Ling (rus)	77	37.47%	348
3	LZW-Ling (eng)	93	44.21%	348
4	LZW-Ling (est)	130	43.57%	639
5	LZW-Ling (rus)	218	31.35%	798
6	LZW-Ling (est)	232	41.37%	1614
7	LZW-Ling (rus)	379	27.88%	1718
8	LZW-Ling (eng)	393	37.70%	3489
9	LZW-Ling (est)	409	39.35%	4996
10	LZW-Ling (eng)	522	38.12%	6979
11	LZW-Ling (rus)	661	26.31%	4577
12	LZW-Ling (est)	721	38.16%	10514
13	LZW-Ling (eng)	999	27.48%	14834
14	LZW-Ling (est)	1071	38.64%	20544
15	LZW-Ling (rus)	1194	26.40%	8734

Table 7. LZW-Ling result table

File nr.	Archiver	File size (KB)	Compression ratio	Time spent (milliseconds)
1	LZW	48	55.34%	224
2	LZW	77	37.78%	258
3	LZW	93	44.63%	400
4	LZW	130	46.89%	722
5	LZW	218	31.47%	814
6	LZW	232	41.59%	1523
7	LZW	379	27.97%	1572
8	LZW	393	37.89%	3701
9	LZW	409	39.54%	7464
10	LZW	522	38.46%	7729
11	LZW	661	26.27%	4636
12	LZW	721	38.60%	14179
13	LZW	999	27.65%	14484
14	LZW	1071	39.14%	18362
15	LZW	1194	26.58%	10350

Table 8. LZW result table

```
private final String[] eng = {"ed ", "er ", " the ", " a ", " an ",
            " as ", "ing ", " with ", "th", " if ",
            " and ", " are ", "'re ", " am ", " by ", "he", "in",
            "er", "an", "re", "on", "at", "en", "nd", "ti",
            "es", "or", "te", "of", "ed", " is ", "it", "al", "ar",
            "st", "to", "nt", "oo", " what ", "n't ", "'s",
            "ion ", "ful ", "less ", " which ", " for ", " at ",
            " no ", " to ", " be", " not ", " from ", "ence ",
            "sh", "ch", "ly", "ist", "ll", "ous ", "re", "un", "de",
            " he ", " she ", " I ", " can ", " ha", "'m",
            "pre", " so ", "ee", "oa", "wh", "ea", "ble", "ow", "ou",
            "ie", "ight", "ay", "oi", "ai", "oy", "'re"
            ,"'ve", "ph", "qu"};
```

Figure 8. English pre-made dictionary from LZW-Ling archiver repository

File nr.	Archiver	File size (KB)	Compression ratio
1	ZIP	48	39.58%
2	ZIP	77	31.17%
3	ZIP	93	34.41%
4	ZIP	130	36.92%
5	ZIP	218	29.36%
6	ZIP	232	36.21%
7	ZIP	379	28.23%
8	ZIP	393	35.88%
9	ZIP	409	35.94%
10	ZIP	522	36.97%
11	ZIP	661	27.84%
12	ZIP	721	35.92%
13	ZIP	999	23.12%
14	ZIP	1071	36.51%
15	ZIP	1194	28.14%

Table 9. ZIP result table

```
private final String[] rus = {"нн", "енн", "ённ", "ян", "ый ",
            "ая ", "ое ", "ые ", "ий ", "ие ", "ость ", "ого ",
            "ой ", "ых ", "их ", "ые ", "ую ", "ому", "ым ", "им ",
            "ыми ", "ими ", "а ", "ов ", "ев ", "и ", "е ",
            "ам ", "у ", "ою ", "ами ", "ах ", "ом ", "ях ", "ями ",
            "ям ", "ей ", "ы ", " в", " во", " до", " за",
            " вы", " к", " меж", " на", " не", " ни", " о", " об", " от",
            " по", " под", " про", " с", " у", " без",
            " бес", " вос", " воз", " из", " ис", " раз", " рас", " пре",
            " при", "айш", "ее", "же", "ше", "ть", "аю ",
            "ет ", "ем ", "ют ", "л ", "ли ", "ла ", "ло ", "ся ", "сь ",
            "чи", "ши", " бы ", " я ", " ты ", " мы ",
            " вы ", " он"};
```

Figure 9. Russian pre-made dictionary from LZW-Ling archiver repository

```
private final String[] est = {"sse", "st", "le", "lt", "ks", "ni ",
            "na ", "ta", "ga ", "te", "de", "id ",
            "im ", "em ", "ma", "da", "ev ", "nud ", "tud ",
            "dud ", "takse ", "dakse ", "akse ", "me ", "äe",
            "ti", "di", "si", " ol", "ei", " ja ", "vat ", "n ",
            "b ", "d ", "vad ", " är", "nd", "hk", "tu",
            "ku ", "ne", "se", "oi", "ke ", "kku ", "ku ", "õi", "ai",
            "ld", "va", "mb", "br", "pr", "hv",
            "aa", "ee", "uu", "üü", "ää", "öö", "ii", "oo", "nn", "mm",
            "ll", " see ", "kk", ", et ", "it ",
            "pp", "sid ", "seid ", "ks", "ss", "ea", "au", "ki ",
            "gi ", " kel", " mil", " on ", " sell"};
```

Figure 10. Estonian pre-made dictionary from LZW-Ling archiver repository

License

Non-exclusive licence to reproduce the thesis and make the thesis public I, Artjom Šiškov,

1. grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, my thesis Development of the text archivers using linguistic features of the language, supervised by Irina Bocharova;
2. I grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright;
3. I am aware of the fact that the author retains the rights specified in points 1 and 2;
4. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Artjom Šiškov 15/05/2025