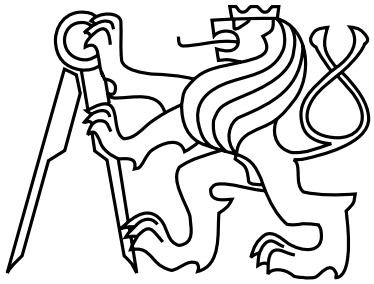
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



Master's Thesis

# Corpus for comparing compression methods and an extension of a ExCom library

*Bc. Jakub Řezníček*

Supervisor: doc. Ing. Jan Holub, Ph.D.

Study Programme: Electrical Engineering and Information Technology

Field of Study: Computer Science and Engineering

May 2010

# Acknowledgements

I would especially like to thank my supervisor, doc. Ing. Jan Holub, Ph.D., for the idea and for his invaluable contributions to this thesis.

I would also like to thank my family and my beloved girlfriend Lucie for their support and encouragement, and Bc. Ondřej Skalička for his quality comments.

# Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act § 60 under Act No. 121/2000, Coll. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Litvínov on May 14, 2010 .....

# Abstract

This thesis deals with the development of a new corpus whose purpose is to evaluate the lossless compression algorithms. We also propose a design of a methodology to maintain its timeliness. The methodology is conceived as a list of steps which are necessary to update a corpus. The efficiency and validity of the corpus, called the Prague Corpus, is verified by thorough experiments using the various compression techniques. This includes the representatives of the statistical and dictionary based methods which were implemented and consequently adapted for the universal library of compression algorithms called ExCom. The second part of the thesis deals with an efficient implementation of all methods which has been researched and described in detail.

# Abstrakt

Tato práce se zabývá vývojem nové korpusu, jehož účelem je porovnání bezeztrátových kompresních algoritmů. Současně předkládáme návrh metodiky pro zachování jeho aktuálnosti. Tato metodika je koncipována jako seznam kroků, které jsou pro aktualizaci nezbytné. Účinnost a platnost korpusu, který byl pojmenován jako Pražský Korpus, je ověřena pomocí podrobných experimentů za využití různých kompresních technik. Mezi tyto techniky patří zástupci statistických a slovníkových metod, které byly implementovány a následně upraveny pro univerzální knihovnu kompresních algoritmů ExCom. Druhá část práce je věnována efektivní implementaci všech metod, které byly detailně prozkoumány a popsány.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 History

History of *data compression* is older than the first need of its usage. Its beginnings date back to early 1840s when *Samuel Morse* presented his famous code for character encoding. Each letter was assigned shorter representation with respect to its occurrence in English texts. The development of *information theory* by *Claude Shannon* 100 years later provided the real basis for data compression [1]. *David Huffman* came up with his invention of lossless variable length encoding. *Huffman coding* was designed as a part of student's final exam, altough Huffman didn't have any proof of its efficiency the result was always better than the best known construction of prefix codes at that time, *Shannon-Fano coding* [2]. The publication of the two lossless *dictionary-based* compression methods, *LZ77* and *LZ78*, in the late 1970s was a significant event for the field of source coding. Since that time many other methods have been presented and data compression is, therefore, very important part of computer science.

## 1.2 Motivation

Despite the increase of storage capacity and continuous improvement of computer networks, the amount of data has grown tremendously in recent years and hence data compression still plays the key role in our daily life. To highlight its importance we can mention an analogy between the real world and the computer's one—before any possibility of storing information (data) to computer files, people had to use (and still use) paper documents and put them into file folders and then store them in filling cabinets, like we similarly do with documents using a computer. Furtunately, we can reduce the computer file size using compression techniques.

As mentioned above, many compression methods exist and they can be divided into several groups. There is no universal method which can *deal with* any type of file and thus to give the best results. For that reason, it is useful to have a common interface or a library which can provide as much different compression methods as possible and choose appropriate one when needed. Such a tool is a result of Filip Šimek's master thesis [3]. The library is called *ExCom* and since its completion, the three compression methods were implemented (*ACB*, *DCA* and *PPM*) as representatives of so called *context methods*. One aim of this work is to extend the ExCom library to provide algorithms of *dictionary-based* and *statistical methods*.

According to the previous paragraph, each method is suitable for different domains of usage. Some can perform better on text files than on images, or on binary files in general. However it is quite important to evaluate particular methods because the compression ratio is not the only parameter to monitor. The running time and algorithm's memory footprint are also significant factors to distinguish different techniques. Data compression *corpora* give us the way to fairly compare several compression schemes.

## 1.3 Main goals

The main contribution of this thesis is to create a new corpus[1] and to design a sophisticated methodology to maintain its timeliness; i.e., to come up with a set of steps/procedures to update and, if necessary, to replace all (or only some) files included in the corpus in the future with a view to keep its features and simplicity to maintain at the same time.

The final corpus should be also tested before considering it as an alternative to existing corpora. Thus, the ExCom library will be extended with the new algorithms to have an adequate number of methods from different categories. It was decided to implement *Shannon-Fano* and *Huffman coding* (both static and adaptive version) as representatives of the *statistical methods*, and *LZ77, LZSS, LZ78 & LZW* from the *dictionary-based methods*. All of here mentioned methods are described in detail in Chapter 2. All implementations should be fast to compress and decompress when there are no limitations in algorithm known. It also must be taken to consider the efficiency, i.e., to output only as much bytes as necessary, e.g., to take advantage of known features or to simplify the header needed for the decoder.

## 1.4 ExCom library

This section briefly outlines the main features and advantages of the ExCom library which has been implemented within the diploma thesis *Data compression library* [3].

### 1.4.1 Purpose

The original idea was brought by *Jan Holub* who stated that the performed research showed a relatively large number of programs used for compression available, but on the other hand a lack of specialized libraries of compression algorithms usable by other applications. Some other issues of the current situation are small amount of implemented methods in these libraries and the type of license. If released under non-free license, the user is not able to modify the source and to experiment with compression algortihms sufficiently. Some methods give better results when combined with each other, therefore availability of more methods would give the user a free hand to try out. Thus, Filip Šimek, the author of ExCom, and Jan Holub decided to design and implement a new library with a common interface for various number of compression schemes flexible enough to co-operate with several applications.

[1] The reasons why to come up with a new corpus will be discussed in detail later in this thesis.

### 1.4.2 Description

The ExCom library is well designed and extensible, which means that only few steps have to be performed to integrate a new compression method. The library has a built-in mechanism used for IO operations available for many types of streams. This includes the standard communication with files stored in the filesystem, data in the memory, and a support of pipes, i.e., output of one compression module becomes an input for another. This unified interface provides some *handy* functions for easy communication with the above mentioned streams, e.g. to access single bits or alternatively to work with blocks of bytes.

The following few lines complete the list of properties and features provided by the ExCom library:

- The library is designed to be a modular system written in C++ programming language and can be used both as static and dynamic library on Unix-like systems
- The concatenation of multiple methods is available for potential experiments[2] and thus to improve the compression results
- It is designed and prepared to be used in multi-threaded programs
- Author of a new implemented method does also not have to meet with the problems regarding a time measure
- It is released as a free software, distributed under *GNU LGPL*[3] version 3

The ExCom library can also be built and used under the Microsoft Windows operating systems using either the MinGW[4] or a Cygwin[5] environment.

## 1.5 Thesis organisation

Chapter 1 outlines the motivation, strategy of work and the main goals of this thesis. It also includes a brief introduction of the ExCom library, its features and advantages.

The second chapter contains the basics and definitions of data compression. The theoretical basis is given to familiarize the reader with some notions and key terms. This chapter also includes a detailed description of the statistical and dictionary-based methods, which are implemented within this work.

The implementation details of all selected compression methods are presented in Chapter 3. This chapter also includes all differences between the original proposed algorithms and new versions.

Chapter 4 is devoted to a new corpus design. The main reasons will be stated why to come up with a new solution. Existing corpora are researched and their pros and cons are discussed. A new corpus and methodology to maintain its timeliness are designed, too. The appropriate public domain representatives are chosen and described.

[2] Using this feature means to create a chain where output of one compression module is provided as input stream for another module.

[3] LGPL = Lesser General Public License.

[4] http://www.mingw.org/, May 2010.

[5] http://www.cygwin.com/, May 2010.

The experimental measurements including performance and parameter testing can be found in Chapter 5.

The last chapter concludes the thesis's main goals and achievements. The reader can also find some comments, ideas and suggestions for future improvement and further research.

# Chapter 2

# Data compression

## 2.1 Introduction

Data compression is a very important branch of *information theory* whose purpose is to minimize the size of data, i.e., to reduce redundancy which is stored in files. In other words there is an effort to represent original input stream differently. We can find redundancy in data because of some reasons, e.g., computer files have their own structure (e.g. to be read and processed easily), thus some information are repeated and some compression schemes are based on this fact. The main goal of the encoding process is that the compressed data still keeps the original information, and hence it can be reconstructed. It is obvious that the *decoder* has to know the output format to be able to realize the decompression. The following section provides a brief introduction to data compression terminology.

## 2.2 Basic concepts and definitions

**Definition 2.1 (Alphabet)**
*Alphabet* is a finite set of symbols, e.g. characters or digits.

**Definition 2.2 (Symbol)**
*A symbol* is an element of an alphabet.

**Definition 2.3 (Source unit)**
*Source unit* is an alphabet symbol or any finite sequence of symbols (words, phrases).

**Definition 2.4 (Code)**
*Code* $K$ is an ordered triplet $K = (S, C, f)$ where

- $S$ is a finite set of source units,
- $C$ is a finite set of codewords,
- $f$ is an injective mapping $S \mapsto C^+$, $\forall s_1, s_2 \in S, s_1 \neq s_2 \Rightarrow f(s_1) \neq f(s_2)$

This can be explained that there are no two different source units from $S$ which are mapped by $f$ onto the same codeword from $C$. Thus, $f$ is an injective mapping. It is a necessary, but not sufficient, condition for unambiguous decoding of each codeword.

**Definition 2.5 (Uniquely decodable code)**
We can say that a code is *uniquely decodable* if its each codeword is recognizable from other codewords. Moreover, all possible *strings* from $C^+$ are uniquely decodable. We denote by $C^+$ the set of all strings with non-zero length containing only symbols from the set $C$.

**Definition 2.6 (Prefix code)**
*Prefix code* is such a set $C^+$ of codewords where no codeword is a prefix of another codeword in the set. Prefix codes are decodable without the need to have a reference to the next codeword following the current decoded codeword. These codes belong to a subset of uniquely decodeable codes. We often use prefix codes with regard to their unique ability when decoding—reading from left to right.

**Definition 2.7 (Compressor and Decompressor)**
*Compressor* and *Decompressor*, sometimes referred to as a *Coder* and a *Decoder*, is a routine or a computer program used for reducing a size of input stream (thus to represent original data in a different way) and for transforming compressed (encoded) data back into its original (decoded) form, respectively.

**Definition 2.8 (Compression ratio)**
*Compression ratio* is a ratio between compressed data size and original (uncompressed) input data size obtained from the following equation:

$$\text{compression ratio} = \frac{\text{compressed data size}}{\text{uncompressed data size}} \tag{2.1}$$

For example, a value of 0.75 signifies that the encoded data occupies 75% of original information size. When the result of equation 2.1 is greater than the value of 1.0, i.e., the compressor produced a larger volume of data than the original size of input stream was, we talk about a *negative compression* [4].

**Definition 2.9 (Corpus)**
*A corpus* is a set of various files, used for evaluating compression methods.

**Definition 2.10 (Lossless compression)**
*Lossless data compression* belongs to a group of algorithms ensuring such an output which becomes the identical data after its decompression. This approach is used when it is necessary to reconstruct encoded data without losing any of original information. The common usage area of these methods is obvious—text files (including source codes), financial data or executable programs.

**Definition 2.11 (Lossy compression)**
*Lossy compression* methods *loses* (or do not use) some information during the encoding process. Therefore, the decoded result of encoder's output is not the same as the original data were. However, it is not undesirable since it can still be used for a particular purpose. We use lossy compression especially in the field of media (audio, video and images) where a loss of some not necessarily needed information is tolerated.

**Definition 2.12 (Symmetrical and asymmetrical compression)**
In the case of *symmetrical compression*, the encoder and decoder use the same technique but in opposite directions. Therefore, the same amount of operations has to be performed (the running time is approximately the same). In contrast, an *asymmetrical compression* method would have either encoding algorithm or the decoding one working significantly slower than the other one.

It is obvious that the usage area of symmetrical methods is where the files are compressed as often as decompressed. When an asymetrical compression method is used, only two possibilities can arise: the compressor performs much faster than the decompressor and *vice versa*. In the former case, we can see its application in data backup because the backed up files are rarely used.

## 2.3 Entropy and redundancy

*Entropy*[1] (or *information entropy*) can be defined as a quantity of information in a source message in the context of *information theory*. The concept of entropy was introduced by Claude Elwood Shannon in his paper *A Mathematical Theory of Communication (1948)* [1]. Let us define it precisely:

Suppose we have $n$ *source units* and and $n$ corresponding *probabilities*:

$$S = \{x_1, x_2, \ldots, x_n\} \tag{2.2}$$
$$P = \{p_1, p_2, \ldots, p_n\} \tag{2.3}$$
$$1 = \sum_{i=1}^{n} p_i \tag{2.4}$$

Since $S$ can contain any symbol from its input set, we also suppose that $S$ can be presented as a *discrete random variable*.

Information content of unit $x_i$ is obtained by:

$$I_{x_i} = -\log_b p_i \tag{2.5}$$

The *entropy* $H$ is defined as an *average information*:

$$H(X) = \sum_{i=1}^{n} p_i I_{x_i} = -\sum_{i=1}^{n} p_i \log_b p_i \tag{2.6}$$

and is measured in *bits* when $b$ (the base of logarithm) is 2. For a value $b = e$ (Euler's number) or $b = 10$ is the unit *nat* or *dit* (abbreviation of decimal digit), respectively.

The entropy function reaches its maximum for $S$ when all source units (symbols) have equal probability. Hence, entropy sometimes refers to an *average uncertainty*, because when processing the input stream it is unpredictable what symbol comes next. The value of entropy also provides a lower bound which can be reached by the best possible lossless compression

[1] Altough we can find the term *entropy* in multiple fields, its name originally comes from the area of thermodynamics. C. E. Shannon used this word thanks to an idea from John von Neumann, a Hungarian American mathematician.

algorithm. Generally, it is not possible to encode a given input stream using less bits (on average) than the entropy $H$ of the corresponding model is [2].

According to the previous definition of entropy (2.6) we can define a term *Redundancy* for a message $X \in S^+$:

$$R(X) = L(X) - H(X) \tag{2.7}$$

where we denote by $L(X)$ the *length* of encoded message $X$ and is defined as follows:

$$L(X) = \sum_{j=1}^{n} d_j \tag{2.8}$$

where $n$ is a number of codewords and $d_j$ is a number of bits of the $j$-th codeword in bits. Therefore, the redundancy is also measured in units of bits.

## 2.4 A classification of methods

Data compression methods differ in many factors, one classification was proposed in the previous section. We divided methods to *lossy* and *lossless* but it is desirable to be more specific. As stated earlier in Chapter 1, there is no universal method which can be used to any type of input data. Thus, it is obvious that many different methods exist. Each of them applies distinct approach to process data with respect to encode its stored information. However, we can always see some similarity. Hence, this section is devoted to a classification of methods from the main perspectives.

In the first place, the compression algorithms can be separated into two basic categories: *statistical methods* and *dictionary-based methods*. We will not deal further with *hybrid methods* category. The methods which belong to this group use principles both of dictionary and statistical techniques. More information can be found in [5]. Let us now describe the rest of groups.

### 2.4.1 Statistical methods

The idea is based on the frequency of occurrence of each symbol found in the input stream. Symbols used more frequently in the source are assigned shorter codes, and, by analogy, the codeword for the infrequent symbol is encoded with a longer bit representation. It is obvious, that these methods use variable-size codes which should be unambiguously recognizable from each other. The solution is self-evident—the usage of *prefix codes*. If the static version[2] of algorithm is used, its disadvantage is that the corresponding probabilities should be known before the encoding begins and thus the input file has to be processed twice. During the first pass the statistical data are collected, processed and prepared, and the second pass is used to replace each symbol by the corresponding codeword afterwards. In this case, the probabilities of symbols are measured. Another possibility is to estimate them. The compression quality depends on the statistical model which is used for encoding. The typical representatives are *Static Huffman coding* [6], *Adaptive Huffman coding* [7, 8, 9, 10], *Shannon-Fano coding* [2], *Arithmetic coding* [11], and *Range encoding*.

[2]Not all statistical methods may have both static and adaptive version.

Despite an easy implementation, Shannon-Fano algorithm does not produce an optimal prefix code in contrast to Huffman coding. However, Huffman coding gives the best results only if all the probabilities are negative powers of two (e.g. 1/2, 1/8), the redundancy is then equal to *zero*. If this condition is not satisfied, the average length of bits per symbol is higher than the optimal minimum set by the entropy $H(S)$ of the source $S$. Huffman coding (both static and adaptive version) will be discussed in more detail in later sections.

Huffman coding is still very popular technique used either separately or combined with some other compression schemes. Some publications (see [12] or [11]) point out the suboptimality of Huffman code which can be limiting in some situations. Therefore, the authors propose to use *arithmetic coding* instead of Huffman codes since arithmetic coding is very nearly optimal in practice thus it can compress data in much better way (at least the resulting code is identical to a Huffman code). The Huffman method always assigns only an integral number of bits to each symbol. When the symbol has its probability of occurrence, say, 0.2 the equation 2.5 gives the result 2.32. Hence, the length of the appropriate codeword is set to 2 or 3 bits. Arithmetic coding tries to overcome this problem by using the single number $n$ $(0.0 \leq n < 1.0)$ representing the whole input stream. The reader may ask why Huffman coding is not superseded by arithmetic coding. The main reason is the encoding speed because of the usage of arithmetic operations, *division* and *multiplication*. We should also mention that arithmetic coding is the patent encumbered method and therefore *range encoding*, a form of arithmetic coding, was proposed. Huffman coding is also easier to understand so it is still widely used.

### 2.4.2 Dictionary-based methods

Unlike statistical compression algorithms, dictionary-based methods do not use the probability distribution of input stream. Instead, *the dictionary* is maintained while the symbols are being input. The encoder tries to find a match between currently processed input data and dictionary items—string patterns (or *phrases*) already seen before. The pointers to this way found items are used as output. The level of compression depends on the number of string repetition in the source. Since the late 1970s when the foundations of these methods were laid, we can present several various techniques. There are different approaches across the dictionary compression spectrum. They differ either in the used data structure representing the dictionary or in the way of substring search. The presentation of *LZ77* concept [13] followed by a publication of *LZ78* [14] resulted into an important milestone having a significant impact on many other derived techniques. One of the aims of this work is to implement some typical methods which will be thoroughly described together with some examples.

### 2.4.3 Other methods

The methods stated here can be considered as so called *context methods*, since the previously processed input is used as context. The context is then used to determine probabilities for the next input symbol. Into the group of context-based methods we can include *BWT* (Burrows-Wheeler transform) [15], *ACB* (Associative coder of Buyanovsky) [16], *DCA* (Data Compression using Antidictionaries) [17], and *PPM* (Prediction by Partial Matching) [18, 19]. The last three methods had been implemented[3] within the Filip Šimek's master thesis [3].

[3] DCA implementation is based on Martin Fiala's version [20] and was adapted to work properly in the ExCom library.

## 2.5 A classification of adaptivity

Further classification can be performed on the compression model which is used by the compression schemes. A model provides the prediction of probability distribution over the input streams [21]. There are three ways of maintaining the model used by both encoder and decoder.

- **Non-adaptive (static) modeling**—the fixed model is available before the compression/decompression. This model stays unchanged during the whole process and it is not set with regard to input data. The advantages are that the encoder and decoder shares the same model and thus it does not have to be attached to compressed data, and that it requires only *one pass* through the data. On the other hand this way used models are well suited to special type of data only, e.g., English texts. It is evident, algorithms using this model would fail (give bad compression results) on any other types.

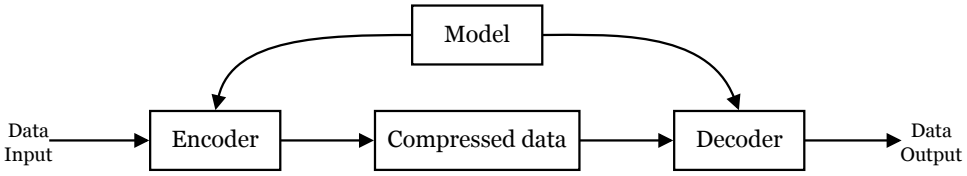



Figure 2.1: Static model

- **Semi-adaptive modeling**—this type of model is similar to a static version but tries to overcome the earlier mentioned issue with collecting statistical data during the first pass. The model is then build and used to encode data during the second pass. Hence, it requires *two passes*, and it tends to slow down the process. The built model has to be transmitted with encoded data to decoder.

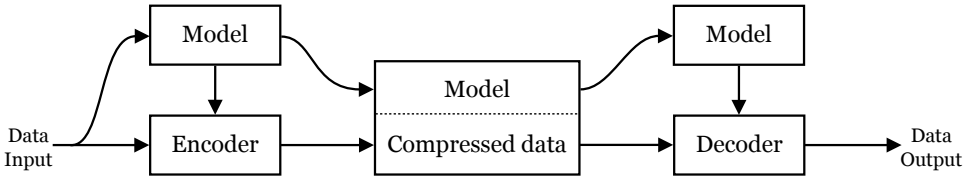



Figure 2.2: Semi-adaptive model

- **Adaptive modeling**—this approach blends the features of both models above. Only one pass is required and the corresponding model is built as the input data are processed, i.e., it changes over time and continuously adapts. The decoder is able to reconstruct the model by the progressive update during decoding, therefore it need not to be attached.

## 2.6 Compression methods implemented in this work

The purpose of this section is to provide an overall description of all methods which are planned to be implemented within this work. All compression algorithms contain an example to illustrate their principle. Three representatives of statistical methods, *Shannon-Fano coding*, *Static Huffman coding* and *Adaptive Huffman coding* are described at first followed by the presentation of typical dictionary-based techniques—*LZ77*, *LZSS*, *LZ78*, and *LZW*.

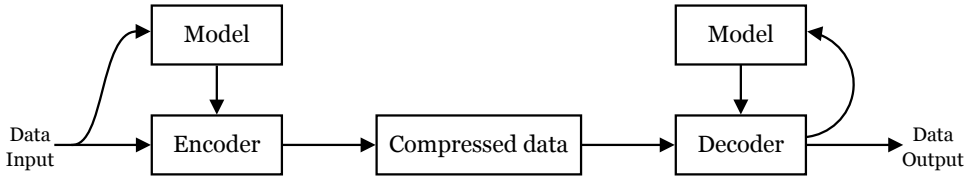



Figure 2.3: Adaptive model

### 2.6.1 Shannon-Fano coding

Shannon-Fano[4] coding is considered to be the first algorithm for constructing binary codes of variable length. This method takes advantage of frequency of occurence of each character from the input alphabet. These frequencies can be either measured (this requires two passes over the source file) or estimated. The idea is based on building the tree starting the process at the root node. Thus, this technique is called a *top-down* approach.

We suppose that initially all symbols become unconnected nodes (tree leaves). A set of given symbols is arranged in non-decreasing order of their occurrence frequencies (or probabilities). Both, non-decreasing and non-increasing principle is possible. We present the former way but the latter approach is analogous. This way prepared set is then divided into two parts with respect to be both segments as *balanced* as possible. In this context, the term "balanced" means that both *halves*[5] are *equal* with regard to the sum of the corresponding frequencies. A new node, as a predecessor of all nodes included in an appropriate part, is created. In the first step, this node is a descendant of the root, in consecutive steps it becomes a child node of the previously created node. This process is applied recursively to both subsets of symbols (nodes) until all leaves are connected. Whenever the current set is divided, the codewords of all symbols in one subset are extended by appending a "0" (1 bit), while a "1" is used in the same way for symbols in the second subset. As a result of this procedure we get a list of prefix codewords for all input symbols. The generalized algorithm using a pseudocode is shown in Figure 2.5.

**Example 2.1** *Consider the following input string S:*

```
S = "BECCDEAEEAEECDEEABBECEEDBCBDECA"
```

An alphabet of this source consists of five distinct symbols $(A, B, C, D, E)$[6]. The frequency of occurrence of each symbol is shown in Table 2.1. Now, we want to obtain the corresponding codewords. As mentioned earlier, we start with a set of unconnected symbols nodes and we will apply the Shanon-Fano algorithm to build a binary tree. The proposed approach is recursive, therefore the left subtree is built at first followed by the construction of the right part of Shannon-Fano tree. For simplicity, let's show a construction of each level of the tree as a separate step.

All four steps needed to build a Shannon-Fano tree are shown in Figure 2.4. Figure 2.4a represents the initial state where only the leaves of the tree are available. This set is then

[4]Shannon-Fano coding is named after Claude Elwood Shannon and an Italian-American computer scientist Robert Mario Fano.

[5]After the splitting, the just created subsets may not necessary contain the same amount of nodes.

[6]We do not include the double quotation marks.

| Symbol | A | D | B | C | E |
|---|---|---|---|---|---|
| **Count** | 4 | 4 | 5 | 6 | 12 |
| **Probabilities** | 4/31 | 4/31 | 5/31 | 6/31 | 12/31 |

Table 2.1: The sorted symbols with their computed probabilities

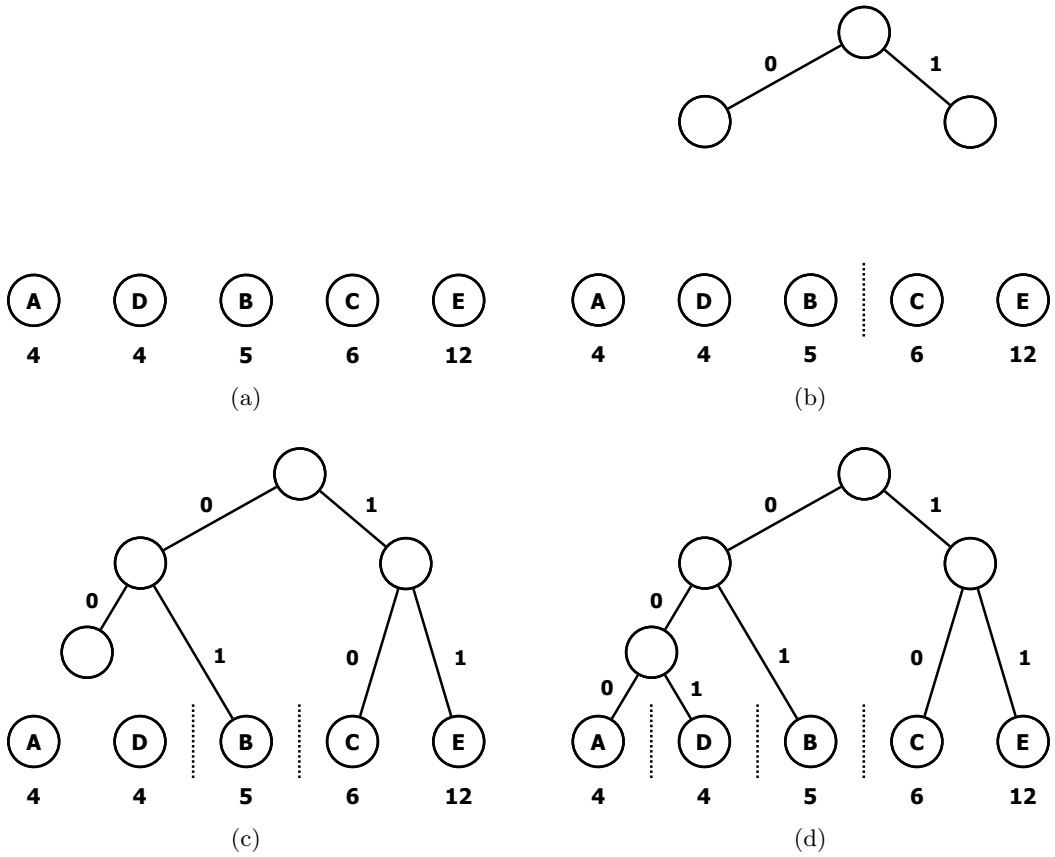



Figure 2.4: Shannon-Fano algorithm in action

divided into two halves. In the next iteration the procedure continues and the previously created subsets are divided again. In each step we always assign the appropriate bit value to the corresponding subset. The vertical dotted lines mark the position where it was decided to make a *split-point* during each step to divide the set into two segments.

The list of divisions during the whole process:

1. As the Figure 2.4b illustrates, the original subset $(A, D, B, C, E)$ is divided into $(A, D, B)$ and $(C, E)$

2. In the next step (Figure 2.4c) the both parts are split into $(A, D)$ and $(B)$, and into $(C)$ and $(E)$ respectively

3. In the last step only two remaining symbol nodes are connected, see Figure 2.4d

As the procedure ends, we get the Table 2.2 of all codewords. To compress the source file we only need to substitute the symbols with the corresponding bit representation. Thus, our example string be encoded the following way:

```
SF(S) = 01|11|10|10|001|11|000|11|11|000...
```

| Symbol | Codeword |
|---|---|
| **A** | 000 |
| **B** | 01 |
| **C** | 10 |
| **D** | 001 |
| **E** | 11 |

Table 2.2: The result of Shannon-Fano algorithm

**Algorithm 1** Shannon-Fano coding algorithm

**Input:** A sorted list of $n$ input symbols $S = \{s_1, s_2, \ldots, s_n\}$ with frequencies of occurrence $F = \{f_1, f_2, \ldots, f_n\}$, where $f_i \leq f_{i+1}, \forall i, 1 \leq i < n$

**Output:** $n$ variable-length codewords $C = \{c(s_i), c(s_i), \ldots, c(s_n)\}$

1: $c(s_i) \leftarrow \varepsilon, \forall i, 1 \leq i \leq n$
2: ShanonFano-Split($S$)
3: **procedure** ShannonFano-Split($S$)
4: **if** $|S| > 1$ **then**
5: split $S$ to $S_1$ and $S_2$ with approximately the same sum of frequencies
6: $c(s_1) \leftarrow c(s_1) + 0, \forall s_1 \in S_1$ ▷ We denote by + the *append* function
7: $c(s_2) \leftarrow c(s_2) + 1, \forall s_2 \in S_2$
8: ShannonFano-Split($S_1$)
9: ShannonFano-Split($S_2$)
10: **end if**
11: **end procedure**

Figure 2.5: Pseudocode of Shannon-Fano algorithm

### 2.6.2 Static Huffman coding

Some characters appear in text more often than other symbols. In English language it is *'E'*, on the other hand *'J'* is an example of symbol we do not *meet* so frequently. Similarly like in the case of Shannon-Fano approach, the Huffman coding tries to take advantage of assigning a shorter representation to those codes for which the probability of occurrence reaches the higher values. Huffman coding shares this concept with the earlier presented Shannon-Fano's version. It only differs in the way how the code tree is built. While Shannon-Fano technique starts construction at the root, Huffman coding proceeds *vice versa*. We call this approach

*bottom-up*. However, this one difference causes the efficiency of compression a lot in behalf of Huffman's algorithm whose description follows. Thus, Shannon–Fano coding does not achieve the best possible code values, so we say the solution is *suboptimal*.

We maintain the set of symbols arranged in the same way like we did in Shannon-Fano version, i.e., the symbols are sorted in non-decreasing order with regard to the count of occurrence of each symbol. Since we work both with single nodes and subtrees, the following situation may occur (it typically does very often). For each item, we hold a weight value which is easily obtained by the sum of weights of all internal nodes. The weight of the leaf is identical to its frequency of occurrence (or probability, then we sum up the probabilities). Thus, the total sum of just created subtree may be greater than the value of some other subtree or node. Therefore, we keep the ordered set during the whole time of constructing the tree, so some changes regarding an order of these subtrees have to be made.

The next procedure is simple. We use two smallest (by sum of weights) items (single nodes or subtrees) to form a new subtree. Thus, a new node (parent of those items) is created and we put this subtree appropriately to a place in the set to keep a sorted sequence. Then we repeat these steps until only one node remains in the set—it is the root of the Huffman tree. After this is done, we traverse the tree from its root to all leaves and simultaneously concatenate the bit values, "0" or "1" depending on the direction. When we reach the leaf, we assign the obtained code to it and continue in traversing. Again, we propose the pseudocode in Figure 2.8 to illustrate Huffman's principle.

Now, let us use the same example as the one in a subsection devoted to Shannon-Fano technique (see 2.1) to point to differences between these two methods.

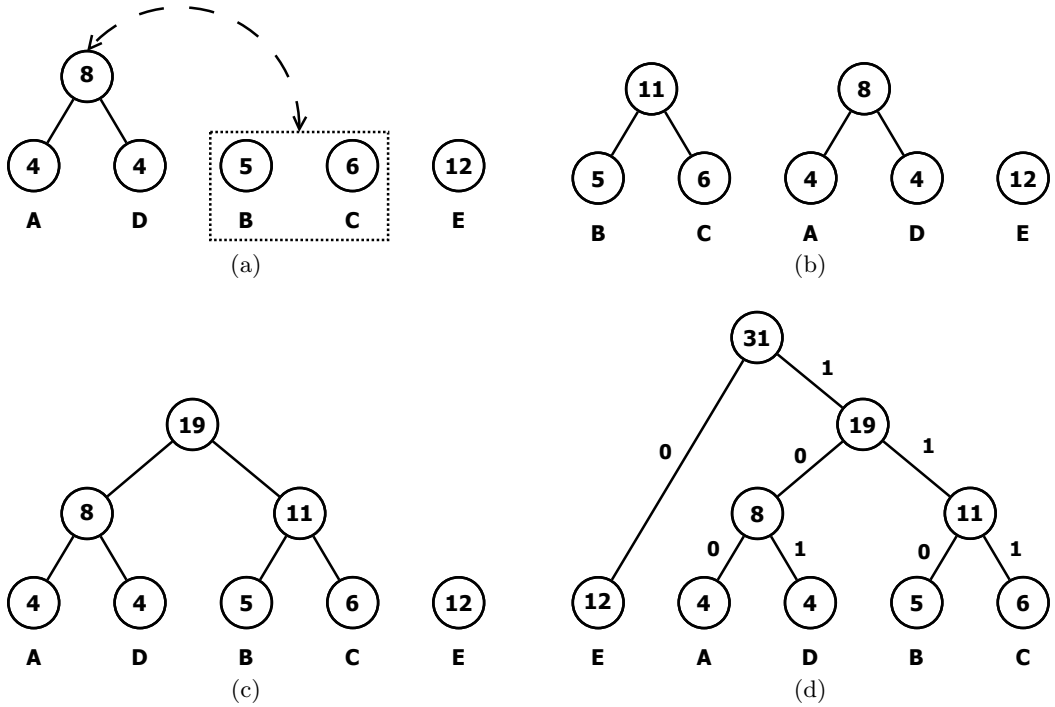



Figure 2.6: The particular steps of building Huffman tree

The process is illustrated by an example in Figure 2.6, the detailed steps follow:

1. At first, the initial set consits only of single unconnected nodes. Figure 2.6a displays the primary step when two nodes are combined to form a new subtree, and a new value (8) is

assigned to it. As we mentioned earlier, we have to keep the set ordered, thus the dashed arrow indicates a new subtree and two other nodes to be swapped. We assume that it is clear to the reader when it is necessary to perform such a step, therefore no more arrows are used in other figures.

2. In the next two figures the algorithm continues in the same manner and two other subtrees are created.

3. Figure 2.6d illustrates the situation when the construction phase is completed, and at the same time the bit values are provided.

In our example, when counting we used the frequencies, as a result of this the root node was assigned a value equal to the length of input stream (31). In the case of using probabilities, the root's value sums to 1. Obviously, each internal node in the tree is assigned the identical value to the sum of its descendant's probabilities.

Once the mapping process (we assign each symbol a new bit representation) is done, we can encode the original file. We scan the input stream and ouput the appropriate codeword of every read symbol.

#### 2.6.2.1 The sibling property

The following interesting property was first proposed in a paper by *Robert G. Gallager* in 1978, see [8]. We traverse the nodes from left to right starting at the lowermost level $l_x$ (the leaves). When the rightmost node is reached, we move up to the level $l_{x-1}$ and apply the same. We repeat this until the root is reached. Thus, we are able to read the value of every node in the Huffman tree. Therefore, we get a sequence of values $V = \{v_1, v_2, \ldots, v_n\}$, where $n$ is a count of all nodes, then:

$$\forall v_i, v_j \in V, v_i \leq v_j (i \neq j)$$

In other words we can list this sequence in order of non-decreasing values, and the tree siblings are side by side. So when we are given a node $n$, its sibling $s_n$ is the node on the same level of the tree to the right. If $n$ is the rightmost node on the level, its sibling is the leftmost one on the level above. Formally, the definition can be stated as follows.

**Definition 2.1 (Sibling property)**
Let $T$ is a binary tree where every node has its sibling (except the root). We say that $T$ has a *sibling property* if all nodes can be listed as a seuquence of non-decreasing values (probabilities or frequencies of occurrence) with the siblings being adjacent in the list.

In Figure 2.7 we can see an example of traversing the Huffman tree which was created earlier. We start at the bottom and continue up to the root. The siblings are the following pairs: {4, 4}, {5, 6}, {8, 11}, and {12, 19}. Furthemore, the pairs {6, 8} and {11, 12} can also be considered to be the sibling, since the first one is always the rightmost node and its sibling is the leftmost one on the higher level.

#### 2.6.2.2 Comparison of Static Huffman coding and Shannon-Fano approach

In the previous parts of this work we demonstrated the function of both Shannon-Fano and Huffman algorithms using the same example. Now, we would like to compare the results

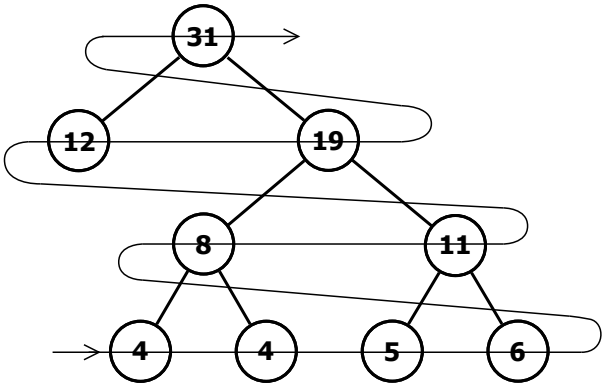



Figure 2.7: The sibling property

and make some conclusions. The Table 2.3 contains the list of all five symbols with their corresponding codewords obtained as the results of application of both methods. We also provide the total number of bits needed to encode the source message.

| | | **Shannon-Fano** | | | **Huffman** | | |
|---|---|---|---|---|---|---|---|
| **Symbol** | **Frequency** | **Codeword** | **Length** | **Total** | **Codeword** | **Length** | **Total** |
| **A** | 4 | 000 | 3 | 12 | 100 | 3 | 12 |
| **B** | 5 | 01 | 2 | 10 | 110 | 3 | 15 |
| **C** | 6 | 10 | 2 | 12 | 111 | 3 | 18 |
| **D** | 4 | 001 | 3 | 12 | 101 | 3 | 12 |
| **E** | 12 | 11 | 2 | 24 | 0 | 1 | 12 |

Table 2.3: The compared results of Shannon-Fano algorithm and Huffman version

When we sum up the both *Total* columns, we get 70, and 69, for Shannon-Fano coding, and Huffman coding, respectively. Even if there are only two symbols, whose codeword length is 3, instead of four symbols in Huffman version, Shannon-Fano technique does not reach the qualities of Huffman coding. While it may seem a negligible, the files used in practice reflect the real values of probability distribution. Hence, the difference may be more significant. However, neither Shannon-Fano's approach nor its successful successor, Huffman coding, does not achieve the optimality in the entropy sense. So, we can calculate the entropy $H$ (2.6) for the source message and compare it with the average number of bits/symbol (BPS) resulting from both methods.

The lower bound of number of bits needed to represent each symbol (on average) is:

$$\begin{aligned} H(X) &= -\sum_{i=1}^{n} p_i \log_b p_i \\ &= -\left(\frac{4}{31}\cdot\log_2\frac{4}{31} + \frac{5}{31}\cdot\log_2\frac{5}{31} + \cdots + \frac{12}{31}\cdot\log_2\frac{12}{31}\right) \\ &\approx 2,176 \end{aligned}$$

While, the values for the methods are:

$$\text{Shannon-Fano:} \quad BPS = \frac{\text{Length of encoded message (in bits)}}{\text{Number of symbols}} = \frac{70}{31} \approx 2.258$$

$$\text{Huffman:} \quad BPS = \cdots = \frac{69}{31} \approx 2.226$$

The results are obvious, as we mentioned above, Huffman coding gives the best results only if all the probabilities are negative powers of two. In this case, the condition is not satisfied. However, it is still better than Shannon-Fano technique, but not as good as for example *Arithmetic coding*

**Algorithm 2** Static Huffman coding algorithm

**Input:** A sorted list of single nodes $L = \{(s_1, p_1, \varepsilon, \varepsilon), \ldots, (s_n, p_n, \varepsilon, \varepsilon)\}$ for symbols $\{s_1, s_2, \ldots, s_n\}$ with probabilities $\{p_1, p_2, \ldots, p_n\}$, where $p_i \leq p_{i+1}, \forall i, 1 \leq i < n$

**Output:** A Huffman tree for $n$ symbols ensuring $n$ distinct prefix codewords

1: **while** $|L| > 1$ **do** ▷ If $|L| = 1$, only the root remained ⇒ the tree is built
2: $l_1 \leftarrow Min(L)$ ▷ Node with the minimal probability value
3: $L \leftarrow L \setminus \{l_1\}$
4: $l_2 \leftarrow Min(L)$
5: $L \leftarrow L \setminus \{l_2\}$
6: $l_{new} \leftarrow (\varepsilon, P(l_1) + P(l_2), l_1, l_2)$
7: $L \leftarrow L \cup \{l_{new}\}$
8: **end while**

Figure 2.8: Pseudocode of Huffman algorithm

### 2.6.3 Dynamic Huffman coding

When the static version of Huffman coding is used, the encoder needs to know the frequencies of occurrence in advance. This causes to read the original data twice to measure them. Another option is to estimate them, but this approach does not give the optimal results because various data contain different number of symbols with different probabilities of occurrence. The later manner is suitable, for example, for English texts. The aim of this method is to pass through the file only once and to build the tree *real-time*. The original concept was independently presented by *Newton Faller* [7], and *Robert Gray Gallager* [8]. Later it was improved by *Donald Ervin Knuth* [9]. Hence, it is sometimes referred to as a *FGK* algorithm. Another version of adaptive Huffman coding is called *Vitter's method*, or *algorithm V*, proposed by *Jeffrey Scott Vitter* [10].

Since the Huffman tree is maintained during the runtime, we say that these codes adapt to the characteristics of the input data. Notice, that the tree may produces different codes as the symbols are processed. The decoder must be synchronized with the encoder, i.e., the build and update the tree in the same way and whenever we interrupt either the decoding or the encoding process, the corresponnding trees should be equal. Furthermore, both algorithms start with an empty tree, in contrast to the static version, where the leaves were available before the encoding process.

#### 2.6.3.1 FGK algorithm

The main idea is based on the *sibling property* we described earlier in this chapter. The tree leaves represent the symbols from the input stream with a weight value which stands for the frequency count. At each state each tree also contains a *zero-node*, whose purpose is to distinguish between the uncompressed symbols (in 8 bit ASCII form) and variable-sized codes, as the results of Huffman traversal. This node must be a part of the tree, thus to be modified during the whole process, too. It is becuase we cannot use any of the variable size codes to achive the differentiation. We can say, it is some kind of the *escape code* of a changeable length. When such a code is read by the decoder, it denotes to read next 8 bits to retrieve an uncompressed byte.

As mentioned above, the output stream consists of either the ASCII symbol, or the variable-sized binary code. It depends on whether the current symbol in the input stream was previously encountered or not. If the former case occurs, the corresponding code for such a symbol is obtained from the tree and output. In the second case, the incoming symbol is directly output.

The Huffman tree needs to be updated after each step to keep the sibling property. If a new symbol occurs, the corresponding node for this symbol together with the new zero-node is created. The previous zero-node becomes a parent of these two nodes, and its weight is incremented by one. Then, we move to its parent and before we increase its weight, too, we check if there is another node with the same value and with the higher index. This node is searched from the current node up to the root, from the left to right. If such a node exists, we need to swap those nodes together with their subtrees. In other words, their parents are exchanged. After this, we can increment the value of frequency count, and continue in the same manner with the parent of a current node until we reach the root. If the symbol, which is already in the tree, is input we just move to the corresponing node and perform the same steps as above. The generalized algorithm, both for encoding and decoding, is proposed by the pseudocode in Figure 2.9.

#### 2.6.3.2 Vitter's method

Vitter's version, sometimes called as $\Lambda$ algorithm, was presented as an improvement to the original method. In each step we swap the nodes only once, in the FGK algorithm this was bounded by $l_c/2$, where $l_c$ is the length of the just added symbol in the previous state of Huffman tree. Vitter also proposed a new numbering mechanism, called *implicit numbering*, where the nodes on the same level of a tree are numbered from the left to right in increasing order. Furthermore, the nodes on one level are numbered with lower numbers than the nodes on the level above. This algorithm also tries to keep the tree at the minimum height, thus to lower the sum of distances from the root to the leaves.

According to [4], the Vitter's method would produce $(S + n)$ bits, while the original FGK algorithm $(2S+n)$ bits. We denote by $S$ the number of bits, if a static Huffman coding with two passes through the file is used. And the result is self-evident, the Vitter's version outperforms the FGK algorithm. However, to implement it with a time efficient factor kept in mind is not trivial.

### 2.6.4 LZ77

*LZ77* method, also known as *LZ1*, is named after *Abraham* ***L****empel* and *Jacob* ***Z****iv* who presented their compression scheme in their article *A Universal Algorithm for Sequential Data*

**Algorithm 3** Adaptive Huffman coding, FGK algorithm

**Input:** The symbol stream $X = \{x_1, \ldots, x_n\}$

```
1:  procedure FGK
2:      Create zero-node
3:      while (input stream X not empty) do
4:          a ← ReadSymbol(X)
5:          if (a not in the tree) then                ▷ The first time a symbol a is seen
6:              code ← GetCode(zero-node)
7:              Output(code)
8:              Output(a)
9:              U ← CreateNode(zero-node, a)           ▷ zero-node and a as the children of U
10:             UpdateTree(U)
11:         else
12:             U ← GetNode(a)
13:             code ← GetCode(U)                      ▷ Code for a node containing the symbol a
14:             Output(code)
15:             UpdateTree(U)
16:         end if
17:     end while
18: end procedure
19:
20: procedure UpdateTree(U)
21:     while (U ≠ root) do
22:         if (∃U_1, U_1.weight = U.weight & U_1.order > U.order) then
23:             SwapNodes(U, U_1)
24:         end if
25:         U.weight ← U.weight + 1
26:         U ← U.parent
27:     end while
28:     U.weight ← U.weight + 1                        ▷ Update the root's weight
29: end procedure
```

Figure 2.9: Pseudocode of adaptive Huffman coding

*Compression (1977)* [13]. Generally, it is based on references to the repeated strings, which were previously seen scanning the input stream.

With the publication of this paper, many other methods based on this idea were presented, thus we class them as the members of so called *LZ77 family of methods*. They all belong to a group of *Dictionary-based methods*. In this case, an external dictionary containing all possible symbol strings is not assumed. Instead of it, this technique uses the principle of so called *Sliding Window* which is divided into two consecutive parts. The left part contains only encoded data and we call it the *Search buffer*. The part on the right side is called *Look-ahead buffer* containing data which are not encoded yet. The purpose of the Search buffer is to provide a dictionary which is used to search in previously compressed data. Search buffer is scanned backwards (from right to left) with a view to find a match for the first symbol in the Look-ahead buffer. After a successful matching, the encoder then tries to match as many symbols (following the first one) as possible (with a direction from left to right) to find the longest exact copy of

actually compressed data in a Look-ahead buffer. The encoder repeats the same procedure until the end of the Search buffer for the reasons of finding eventually longer match. Data are then encoded as a triplet $(i, j, s)$, where:

- $i$ is a position of the first symbol match in the Search buffer
- $j$ stands for a length of the best match (number of used symbols)
- $s$ is the first symbol, which could not be encoded

In other words, the algorithm's aim is to find the longest prefix in not encoded part which would be the same as some other sequence of symbols found in the Search buffer. The prefix must start anywhere in the range ⟨1, length_of_the_search_buffer⟩, and may contains the symbols from the Look-ahead buffer, i.e., we can cross the *imaginary line* representing the border between these two buffers.

When searching for a match, we alwas start with a single symbol only. Then, two possibilities may occur:

1. The symbol $s$ is not found in a dictionary, then the output triplet would look like $(0, 0, s)$. Hence, the Search buffer is designed as an array (or a *Circular queue*), whose indices always begin at 1 to distinguish whether the match was found or not.
2. The symbol $s$ is found at at least one position, thus the encoder tries to match as many symbols following the symbol $s$ as possible.

The sliding window is then shifted to the right $(j + 1)$ positions. The process is repeated until the Look-ahead is empty and all input symbols are successfully encoded.

**Example 2.2** *Use LZ77 algorithm to encode the following input message $M$:*

$$M = miss\_pippi\_in\_mississippi$$

The size $S$ of the Search buffer is set to $S = 6$ and for the Look-ahead buffer suppose the length of 4 symbols.

The whole process is shown in Figure 2.10. The Search buffer is is displayed on the left followed by the Look-ahead buffer. On their sides, we can see the symbols shifted outside the bounds of the Search buffer (e.g. `..pi`), or the symbols which were not read, yet, respectively (e.g. `ss..`). In the rightmost column we can see all triplets. Now, we will go through the steps to make it obvious.

1. In the first step we have loaded four symbols `miss` into the Look-ahead buffer. No symbol had been encoded so far, thus the Search buffer is empty and there is no reason to search for the first symbol `m`. Instead of it we output the triplet (0,0,"`m`"), and shift the Sliding window to the right.

2. – 3. Repeat the same as in the step 1.

4. Now, we try to find a match for symbol `s`. We start at index = 1 and we are immediately successful. However, we continue to the end of the Search buffer. No other same symbol is matched, therefore we output a token (1,1,"␣"), while the symbol ␣ is next to `s`.

5. – 6. The same approach like above.

7. We scan the Search buffer looking for a match for the first symbol `p` in the Look-ahead buffer. We match the 2 `p`'s in the encoded part of the Sliding window. One at position 1, and the other in a distance 3 from the beginning. We continue in the searching of `pi`. The comparison with the first `p` would fail, thus we move to the left and we match it at position 3. We are unable to match any other longer substring, so in this case the token is following: (3,2,"␣").

8. During this step the prefix `i` (no longer prefix cannot be matched) is found at positions 2 and 5. Therefore, we can encode it either as (2,1,"`n`") or (5,1,"`n`"), respectively. The decision does not affect the result, however the encoder chooses the former possibility since the position value ($i = 2$) is less than 5 and thus can be encoded using less bits.

9. – 12. We continue in the same manner.

13. – 14 We perform the last search for symbol `p` and encode the triplet (1,1,"`i`"). No more symbols remain, we are done with the encoding process.

| | | 6 | 5 | 4 | 3 | 2 | 1 | Look-ahead buffer | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | | | | | | | | m | i | s | s | ␣p.. | (0,0,"m") |
| 2. | | | | | | | m | i | s | s | ␣ | pi.. | (0,0,"i") |
| 3. | | | | | | m | i | s | s | ␣ | p | ip.. | (0,0,"s") |
| 4. | | | | | m | i | s | s | ␣ | p | i | pp.. | (1,1,"␣") |
| 5. | | | m | i | s | s | ␣ | p | i | p | p | i␣.. | (0,0,"p") |
| 6. | | m | i | s | s | ␣ | p | i | p | p | i | ␣i.. | (5,1,"p") |
| 7. | mi | s | s | ␣ | p | i | p | p | i | ␣ | i | n␣.. | (3,2,"␣") |
| 8. | s␣ | p | i | p | p | i | ␣ | i | n | ␣ | m | is.. | (2,1,"n") |
| 9. | ..pi | p | p | i | ␣ | i | n | ␣ | m | i | s | si.. | (3,1,"m") |
| 10. | ..pp | i | ␣ | i | n | ␣ | m | i | s | s | i | ss.. | (4,1,"s") |
| 11. | ..i␣ | i | n | ␣ | m | i | s | s | i | s | s | ip.. | (1,1,"i") |
| 12. | ..in | ␣ | m | i | s | s | i | s | s | i | p | pi | (3,3,"p") |
| 13. | ..is | s | i | s | s | i | p | p | i | | | | (1,1,"i") |
| 14. | ..si | s | s | i | p | p | i | | | | | | |

Search buffer

Figure 2.10: Example of LZ77 Sliding window

As we mentioned, the encoder outputs only the triplets consisting of 3 items. We represent each of them using the different number of bits. The choose of length cannot be done arbitrary. The size of the *position* $i$ (or index) in the Search buffer $S$ is determined by $\lceil \log_2 |S| \rceil$. In practice, we use Search buffers 1,024–8,192 bytes long, so the bit size is typically somwhere between the values 11–13. The formula for the *length* field is similar, we only have to substract 1 from the size of the Look-ahead buffer $L$. This is because the third field of the token is a symbol following the longest match and we cannot extend beyond the bounds of the Look-ahead since those data are not read yet, or there are no more data in the input stream. So we evaluate it as follows, $\lceil \log_2(|L| - 1) \rceil$. In contrast to the former case, we use the Look-ahed buffer, whose size is several times less than the practical values of the Search-buffer length. It is typically only a few tens of bytes long, thus we need only the few bits. The *symbol* item is typically represented as on byte (8 bits), but we can generally say the size is set to $\lceil \log_2 |A| \rceil$, where we denote by $A$ the input alphabet. In practice, the token can occupy approximately 3 bytes (12, 4, and 8 bits, respectively).

We can establish the length for a token $(i, j, s)$ with regard to our example 2.2:

- The index field $|i| = \lceil \log_2 |S| \rceil = \lceil \log_2 6 \rceil = 3$
- The length field $|j| = \lceil \log_2(|L| - 1) \rceil = \lceil \log_2(4 - 1) \rceil = 2$
- We suppose that all 256 possible bytes are used, thus we set $s$ to the value of 8

Therefore, the size of one token is $3 + 2 + 8 = 13$ bits. We have encoded 13 tokens in total. It is evident, that the length of compressed stream is $13 \cdot 13 = 169$ bits. The size of original message was $25 \cdot 8 = 200$, where the value 25 stands for the number of all symbols with 8-bit representation. The compression ratio is 169 / 200 = 0.845. As we can see, the size of the Search buffer should be set to a power of two. In the case of the Look-ahead buffer, the size can be set to a power of two + 1, since we use the character at the last position for the symbol field in a token.

#### 2.6.4.1 Decompression algorithm

LZ77 based schemes are asymetric, since the encoding process takes more time than the decoding. We also maintain the Sliding window of the same size, but the decompression is much simpler. We do not have to perform the matching procedure as in the case of compression. The decompression can be described in several steps.

The decoder:

- reads a token,
- copies the corresponding substring (a match in the buffer),
- concatenates it with the symbol field,
- output data,
- and shifts the buffer.

### 2.6.5 LZSS

LZSS is a lossless compression scheme and an efficient derivate of LZ77. This algorithm was invented by *James A. Storer* and *Thomas G. Szymanski* in 1982, and their considerations were presented in the article *Data Compression via Textual Substitution* [22]. Their aim was to overcome some drawbacks of LZ77. Now, we will go through its features and improvements.

- At first, LZSS differs in implementation details. The Look-ahead buffer is implemented as a circular queue. The circular does not necessary need to be any sophisticated data structure, a basic linear array can be used instead. We only have to maintain two pointers—one of them indicates the *start* position where we can delete the items in the queue, and the other one is an *end* index used for appending the symbols.
- Since the searching of the long matches is slow, the dictionary (the Search buffer) is held in a binary search tree. The basic concept remains the same, when we shift the window $n$ positions to the right, we delete $n$ nodes (words) and the same number of nodes is again inserted. The nodes are compared in lexicographic order, i.e., for example the string `compression` precedes the string `data`. More solutions to solve the speed performance will be discussed later in this section.

**Algorithm 4** LZ77 encoding algorithm

**Input:** The symbol stream $X = \{x_1, \ldots, x_n\}$, *sBufferLen*, *lBufferLen* of type integer

**Output:** A sequence of triplets in $(i, j, a)$ form, where $i$ is an index to *sBuffer*, $j$ is the length of the longest match, $a$ stands for a symbol

**Structures:** SearchBuffer *sBuffer* of size *sBufferLen*, LookAheadBuffer *lBuffer* of size $lBufferLen$, SlidingWindow *sWindow*

1: $sWindow \leftarrow sBuffer \cup lBuffer$
2: **while** (*lBuffer* not empty) **do**
3: Find the longest match for *lBuffer* in the *sWindow*
4: ▷ Pointer to this match must start in the *sBuffer*, *lBufferLen-1* is its max. value
5: $i \leftarrow$ {Pointer to the match}
6: $j \leftarrow$ {Length of the match}
7: ShiftToTheRight(*sWindow*, $j$) ▷ To have the next symbol $a$ at the first position
8: $a \leftarrow$ GetFirstSymbol(*lBuffer*)
9: Output($(i, j, a)$)
10: ShiftToTheRight(*sWindow*, 1)
11: **end while**

Figure 2.11: Pseudocode of LZ77 encoding process

However, the most fundamental difference between these two techniques is in the usage and form of tokens. While LZ77 maintains only one token type, LZSS outputs two variants—original data (i.e. the literal) are directly sent to an output without any encoding, or the *(offset, length)* pair is used. Therefore, we have to use another extra one-bit flag to distinguish between them. The reasons why this approach is used are described now.

LZ77 overcomes the *no-match problem*[7] by outputting a token in the following form: $(0, 0, \mathrm{s})$, i.e., only one single byte is output without any reference to the Search buffer. In the previous section we mentioned, that in practice the typical token can be about 24 bits long. But, only 8 bits are needed for storing a character. For this reason, the dictionary reference can be actually longer than the original string the encoder was trying to substitute. Thus, no compression occurs, instead of it we produce the redundant information. The situation does not change even if the symbol is found, while still only two bytes are encoded. Hence, we establish a new value for the minimal length.

To point up the mentioned improvements we propose an Example 2.3 and compare the results with the LZ77 output.

**Example 2.3** *Encode the following input message $M$ using the LZSS technique:*

$$M = aaaaaabaaaaabbbabbbbbabbabb$$

The sizes of both buffers remains the same, i.e., it is 6 for the Search buffer, and the value is 4 in the case of the Look-ahead buffer. Since we can make use of the whole size of the Look-ahead buffer, its bit representation increase by 1 to the total 3 bits. Hence, the size of the token is 6 bits.

[7] This term stands for a situation when no current data from the Look-ahead buffer were found in the dictionary.

The principle of the Sliding window was thoroughly described for the LZ77 Example 2.2, thus we only provide the sequence of tokens. The literals are shown in ⟨0,"s"⟩ form, and the *(offset, length)* pair as ⟨1,(i,j)⟩. The complete output stream is as follows. Notice, that the angle and round brackets are not output:

⟨0,"a"⟩, ⟨1,(1,4)⟩, ⟨0,"a"⟩, ⟨0,"b"⟩, ⟨1,(5,4)⟩, ⟨0,"a"⟩, ⟨0,"b"⟩, ⟨0,"b"⟩, ⟨0,"b"⟩, ⟨1,(4,4)⟩, ⟨1,(6,4)⟩, ⟨1,(3,4)⟩

We have seven literals which occupy $7 \cdot 8 = 56$ bits. In the case of the *(offset, length)* pair it is $5 \cdot 6 = 30$ bits. We have to add 12 bits to distinguish between 12 tokens. This gives us 98 bits in total.

When we try to encode the message $M$ with LZ77 variant using the same parameters, the output would be:

(0,0,"a"), (1,3,"a"), (1,1,"b"), (4,3,"a"), (6,2,"b"), (1,1,"a"), (4,3,"b"), (6,3,"b"), (3,2,"b")

Since we know that it is 13 bits needed to represent one of these tokens, we get $9 \cdot 13 = 117$ bits, in other words this way compressed message is 19 bits longer.

**Algorithm 5** LZSS encoding algorithm

**Input:** The symbol stream $X = \{x_1, \ldots, x_n\}$, *sBufferLen*, *lBufferLen*, *minLen* of type integer
**Output:** A sequence of tokens in either $(i, j)$, or in $(a)$ form where $i$ is an index to *sBuffer*, $j$ is the length of the longest match, $a$ stands for a single symbol
**Structures:** SearchBuffer *sBuffer* of size *sBufferLen*, LookAheadBuffer *lBuffer* of size *lBufferLen*, SlidingWindow *sWindow*

```
1:  sWindow ← sBuffer ∪ lBuffer
2:  while (lBuffer not empty) do
3:      Find the longest match for lBuffer in the sWindow
4:              ▷ Pointer to this match must start in the sBuffer, lBufferLen is its max. value
5:      i ← {Pointer to the match}
6:      j ← {Length of the match}
7:      if j ≥minLen then
8:          OutputPair(i, j)                          ▷ This also outputs a bit of value 1
9:          ShiftToTheRight(sWindow, j)
10:     else
11:         a ← GetFirstSymbol(lBuffer)
12:         OutputLiteral(a)                          ▷ This also outputs a bit of value 0
13:         ShiftToTheRight(sWindow, 1)
14:     end if
15: end while
```

Figure 2.12: Pseudocode of LZSS encoding process

#### 2.6.5.1 Techniques to improve the speed performance

The most significant drawback of LZ77 method and its derivates is the way how the matches are found in the dictionary. The original concept proposed the usage of a brute force sequential search (comparing symbol by symbol). This approach has a critical impact on the encoding time. *Timothy Bell* and *David Kulp* [23] have come up with several methods which can be used to speed up the compression as much as possible. Now, we try to breifly outline their basic

principle and description.

- **Knuth-Morris-Pratt algorithm**—this algorithm is generally used when we have a pattern and a long text to search in. It takes advantage of the observation when a mismatch during the search happens. In simple terms, the pattern contains enough information to determine where a position of the next match can be. Therefore, it still keeps the information during the text scan. The complexity is in $\mathcal{O}(n+m)$ instead of the worst case complexity $\mathcal{O}(nm)$ in the case of naive algorithm, where $n$ is the text length, and $m$ the length of the pattern. The pattern needs to be preprocessed first.

- **Linked lists**—the search process always starts with the first symbol in the Look-ahead buffer, and then in the case of successful match, the adjacent symbol to the first one is taken with respect to find a longer match until the whole dictionary is searched. The linked lists give us the opportunity to improve this a lot—we can maintain the linked list for every possible byte (256 possibilities). Each of this lists would contain indices to the Search buffer. As the Sliding window is shifted we must dynamically remove, and at the same time insert the new indices.

- **Hash tables**—we can also maintain the hash table using the string of $k$ symbols long as the hash key. However, the size of this table can be very large, so some sophisticated algorithms has to be applied, see [24] for more information about this.

- **Splay trees**—a splay tree is a special variant of a self-adjusting binary search tree, where its items accessed in recent time can be quickly reached again. It provides good performance for operations such as insertion or removal in $\mathcal{O}(\log n)$ amortized time[8]. This idea of this binary tree was firstly presented in 1985 by *Daniel Dominic Sleator* and *Robert Endre Tarjan*, see [25].

More techniques, such as *Tries*, *Suffix Trees*, or some other string search methods (e.g. *Boyer-Moore algorithm*) can be used when implementing LZ77, LZSS, or their derivates.

### 2.6.6 LZ78

It took only one year for *Abraham Lempel* and *Jacob Ziv* to introduce a new data compression scheme—LZ78, which is sometimes referred to as *LZ2*. The principle, which was proposed in the *Compression of Individual Sequence Via Variable-Rate Coding* paper [14], is completely different. This approach does not use the Sliding window, instead of it maintains the *table of all phrases*. In other words, we have a dictionary with all previously processed strings as its items. In contrast to LZ77 (and other Sliding window based methods), the algorithm does not delete any item, therefore we can take advantage of the strings encoded in the distinct past. However, this reflects negatively on the size of the dictionary, which tends to grow rapidly. The size depends only on the memory which is currently available. Unlike LZ77, the output token is in $(i, s)$ form, thus, it consists only of two fields. The first one is an index to the dictionary representing the *prefix*[9]. The *symbol* is denoted by $s$. We do not need the *length* field. As we will see later, the length of the phrase is determined based on the value in the dictionary it is referred to. Whenever we output the token, a new item is added. This clarifies why the dictionary size increases so fast.

The encoding algoritm can be described in the few steps:

[8] The algorithms which takes amortized time guarantee the average time taken per operation.
[9] The meaning of this term in this context will be described shortly.

- The dictioanary is initialized as empty, and so the prefix is.
- Encoder reads the first symbol from the input stream (it alaways reads one character at a time).
- If the (prefix+symbol) is found in the dictionary, we extend the current character (to tell the truth, we change the value of the pointer referencing the prefix in the dictionary). We repeat this until the prefix (formed by the concatenating of the read symbols) is not present in the dictionary. Then we output $(i, s)$. where $i$ is the longest prefix found and $s$ the current symbol from the input source. This phrase (a pointer to the prefix + symbol) is added into the dictionary. Theoretically, there is no phrase-length maximum value. We are only limited by the memory size.
- The situation, when the phrase is not found, typically happens in the first step (and obviously in the others), since the dictionary is empty. Hence, the token would look like $(\varepsilon, s)$, where $\varepsilon$ indicates the *null* prefix for symbol $s$. In the output stream, the $\varepsilon$ sign can be represented as 0 (zero), thus the next indices would be 1, 2, ... This gives us the opportunity to encode the index field using the variable number of bits. We start with 1 bit only (0, 1 indices), then increase by one to 2 (2, 3), then to 3 for representing the indices in the range $\langle 4, 7 \rangle$, etc.

The formalized pseudocode of this algortihm is shown in Figure 2.13.

An interesting observation can be made after reading the previous list of steps. The dictionary always contains the substrings created by removing the character at the last position of another phrase. In other words, when a phrase {`compression`} is present in the dictionary, we are able to find the following phrases, too:

{`compressio`}, {`compressi`}, {`compress`}, ..., {`co`}, {`c`}

In practice, the dictionary is usually implemented as a *trie*[10]. A trie (or prefix tree) is a multi-way tree data structure commonly used for storing strings over an alphabet. Its principle is based on the idea, where the strings with the same prefix have (share) the same parent node. The root is assigned the empty string. Unlike the binary search trees, every node can have more than two children. In our case, this can be up to 256 descendants. The main advantages over the classic binary search trees are the speed performance when looking up the strings, and the space required for the structure.

The decoder works in a similar way like the encoder does. During the decoding process the dictionary is built with regard to the read tokens. However, it is simpler and faster than the encoding.

As we mentioned earlier, the dictionary size may fill up and exceeds its limit. This situation can occur very often, but in the case of very small input data stream, this does not happen. So the encoder and the decoder should solve this somehow to overcome it. The list of the most used methods follows.

- Keep the dictionary in its current status (freeze it). We can still use a static dictionary to encode incoming strings, however we cannot add the new phrases into it
- Delete the entire dictionary and set it to its initialized (empty) status. Thus, the output stream is divided into several independent blocks with their own dictionary. This approach

[10] From the *retrieval* word.

is alike LZ77 method—the future symbols take more advantage of the new data rather than of the old ones

- We can delete some dictionary entries which were not used as often recently. This solutions brings also some issues—we do not know how to find out how many and which items to delete, we also need to keep the prefix property
- Another strategy offers to freeze the dictionary and monitor the compression ratio. When it gets worse, we start with the empty dictionary. This is used in the UNIX compress utility

**Algorithm 6** LZ78 encoding algorithm

**Input:** The symbol stream $X = \{x_1, \ldots, x_n\}$
**Output:** A sequence of tokens in $(i, a)$ form where $i$ is an index to the *dictionary*, $a$ stands for a single symbol
**Structures:** The Dictionary *dictionary*

```
1:  dictionary ← ε, prefix ← ε, i ← 1
2:  while (input stream X not empty) do
3:      symbol ← x_i
4:      if ((prefix + symbol) is present in the dictionary) then
5:          prefix ← prefix + symbol
6:      else
7:          if (prefix is empty) then
8:              prefixIndex ← 0
9:          else
10:             prefixIndex ← GetPrefixIndex(prefix)
11:         end if
12:         Output(prefixIndex,symbol)
13:         Insert(i, prefix+symbol)
14:         i ← i + 1
15:         prefix ← ε
16:     end if
17: end while
18: if (prefix is not empty) then
19:     prefixIndex ← GetPrefixIndex(prefix)
20:     Output(prefixIndex)                    ▷ Only the index is output now
21: end if
```

Figure 2.13: Pseudocode of LZ78 encoding process

**Example 2.4** *Use LZ78 algorithm to encode the following input message $M$:*

$$M = miss\_pippi\_in\_mississippi$$

**Solution:** We will read character by character, and consecutively build the dictionary. Now, let us describe the encoding steps thoroughly.

- Initially, the dictionary is empty. The prefix is set to *null*

- The first symbol is `m`, it is certainly not in the dictionary, thus the (0,"`m`") pair is output and the phrase `m` is added into the dictionary. The same repeats with two next symbols—`i` and `s`. These situations are pictured in Figures 2.14a, 2.14b and 2.14c
- Then the symbol `s` is read, which was encoded during the third step. Thus, encoder finds it in the dictionary and inputs the next symbol ␣. In this case, the phrase `s`␣ is not present. The token (3,"␣") is output, since an index of `s` is 3. Like in the previous steps, we add this phrase into the dictionary and read the next chracter
- The next characters are processed in the same manner. Now, we will describe only the two last steps
- The symbol `i` is input. Since the second step we know it is in the dictionary, therefore `p` is read as the next symbol. The phrase `ip` is also in the dictionary, because it was previously seen in *pippi*. However, the next phrase `ipp` is not found in the dictionary. It is output as (6,"`p`")
- The reading of the last symbol `i` indicates both the successful finding in the dictionary (index 2) and end-of-file, since the input stream does not have more symbols (bytes) to be processed. Thererefore, we output (2,$), where we denote by $ EOF. Or just the value of 2 can be output—it depends on how the encoder and decoder are designed. The solution is demonostrated by the line 18 in the pseudocode describing the function of LZ78
- Table 2.4 shows all 15 steps in encoding the message $M$ from the 2.4 example. The complete graphical representation of the dictionary as a trie is displayed in Figure 2.15

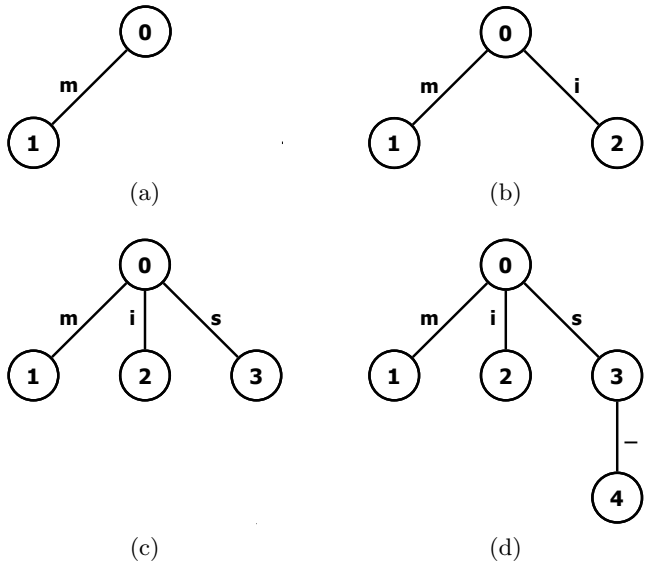



Figure 2.14: The first four steps of building the LZ78 dictionary

### 2.6.7 LZW

LZW is the last significant representant of the dictionary-based methods presented in this work. It is a very popular variant of LZ78 because it is simple to understand the basic principle, and, in fact, easy to implement. The method was developed by *Terry Welch* in 1984 for the purpose of improving its predecessor, see [26] and [27]. It became the first widely used compression

| Index | Phrase | Token | Index | Phrase | Token |
|---|---|---|---|---|---|
| 0 | $\varepsilon$ | | 8 | ␣ | (0,“␣”) |
| 1 | m | (0,“m”) | 9 | in | (2,“n”) |
| 2 | i | (0,“i”) | 10 | ␣m | (8,“m”) |
| 3 | s | (0,“s”) | 11 | is | (2,“s”) |
| 4 | s␣ | (3,“␣”) | 12 | si | (3,“i”) |
| 5 | p | (0,“p”) | 13 | ss | (3,“s”) |
| 6 | ip | (2,“p”) | 14 | ipp | (6,“p”) |
| 7 | pi | (5,“i”) | 15 | i(EOF) | (2,EOF) |

Table 2.4: All encoding steps in the LZ78 example

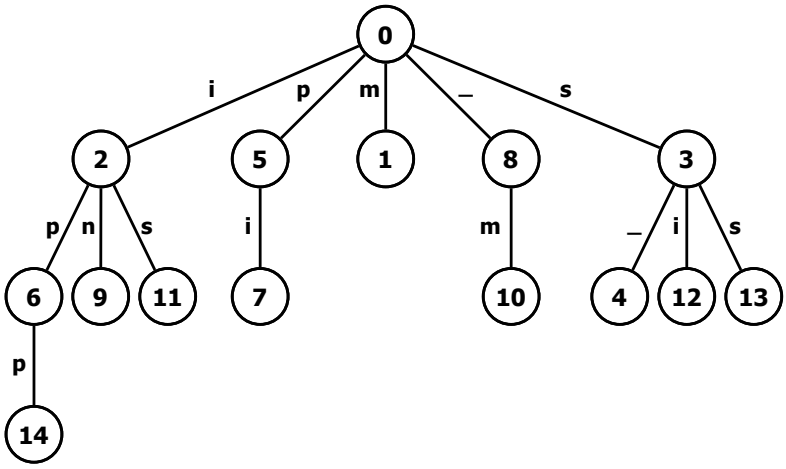



Figure 2.15: The final LZ78 dictionary tree

technique, but it does not necessary give the optimal results—sometimes this method is referred to be slow to adapt to the input data. It is the most suitable to use this algorithm for encoding the files that contain lots of repetitive data. LZW founds its place as a technique used in `GIF` in 1987. We can find it to be optionally used in `TIFF` and `PDF` files (some versions).

We will describe the encoding idea and explain some its features. This compression scheme tries to eliminate the symbol field of LZ78 token. Hence, we output only the indices addressing the dictionary entries. The dictionary is, in contrast to LZ78, initialized to the entire set of symbols of input alphabet. In practice, at the beginning of the encoding process the number of these entries equals to 256. This property ensures that first symbol (and obviously all the others) is always found in the dictionary, and it also explains the output token form, which consists of only one field.

During the whole process, encoder maintains the dictionary as the some kind of *table of strings*, where the items at positions $\langle 0, 255\rangle$ are occupied with the single characters, and the subsequent positions represent the substrings of length at least 2. We read character by character with concatenating them into a string $S$. As long as the encoder searches for this string and is successful, it is expanded with more and more symbols. Once, it can occur that the string $Sx$ ($x$ is currently the last read symbol) is not found in the dictionary. Then, the index of $S$ is output, and the phrase $Sx$ is put into the first unused position. The string $S$ is also set to the symbol $x$.

The original concept proposed the usage of the fixed-length tokens. The dictionary was

planned to contain up to 4096 items. Hence, the tokens would be represented using 12 bits. However, this approach can be improved by the application of variable-sized codes. As we already know, the dictionary is commonly initialized with 256 values with 8-bit representation (0–255). Thus, the next index (256) needs 9 bits to be encoded. The next value, when we have to increment the required number of bits, is 512, 1024 and so on. In other words, as the encoding progresses, we increment the index which is used to add the new phrases, and this index is also output as a token. So, we can start with 9 bits and gradually increase the number of bits.

The situation, when the dictionary fills up, also happens and is not overcome in this case. Thus, it needs to be solved. Fortunately, we can use the same approaches as we described above for LZ78 method.

Again, we propose the example to illustrate the algorithm and its pseudocode in Figure 2.16, like we did before when introducing the compression methods to the reader.

**Algorithm 7** LZW encoding algorithm

**Input:** The symbol stream $X = \{x_1, \ldots, x_n\}$, alphabet $\Sigma$ of $k$ symbols
**Output:** A sequence of tokens in $(i)$ form where $i$ is an index to the *dictionary*
**Structures:** The Dictionary *dictionary*

```
1:  INSERT(dictionary, a_i), a_i ∈ Σ, ∀i, 1 ≤ i ≤ k          ▷ Dictionary initialization
2:  string ← READNEXTBYTE(X)
3:  while (input stream X not empty) do
4:      symbol ← READNEXTBYTE(X)                              ▷ Returns x_i
5:      if ((string+symbol) is found in the dictionary) then
6:          string ← string + symbol
7:      else
8:          stringIndex ← GETSTRINGINDEX(string)
9:          OUTPUT(stringIndex)
10:         INSERT(dictionary, string+symbol)
11:         string ← symbol
12:     end if
13: end while
14: stringIndex ← GETSTRINGINDEX(string)
15: OUTPUT(stringIndex)
```

Figure 2.16: Pseudocode of LZW compression method

**Example 2.5** *Encode the following input message $M$ using the LZW compression method:*

$$M = ababaaaababababca$$

**Solution:** We initialize the dictionary to contain all possible symbols from the input alphabet. For simplicity, suppose the alphabet containing only 3 symbols $\Sigma = \{a, b, c\}$. The initialized dictionary prepared for the first step is shown in Figure 2.17. We also propose the final (Figure 2.18) form of the LZW dictionary in this example with the list of output codes, and the following list summarizes all the steps of the encoding process. We denote by $S$ the *string*, and by $x$ the currently read *symbol* like it is used in the pseudocode 2.16.

1. We start with the initialized dictionary and by assigning the first values to $S = a$, and to $x = b$, respectively. The phrase $ab$ is obviously not found, thus we output the index of $S$ (1) and add the phrase into the dictionary (node 4). Then we initialize the *string* $S$ by assigning the value of $x$ to it, therefore $S = b$

2. The next symbol in the message $M$ is $a$, we search for the phrase $ba$ and it is not present in the dictionary, too. We do the same process like in the preivous step. A node 5 is added, and the *string* $S = a$

3. We input the next character and obtain the phrase $ab$, which was added in the first step. Therefore, we read the next one to concatenate it into the new phrase. However, the phrase $aba$ is not found the dictionary. We output the index of the phrase $ab$, and prepare the *string* for the next iteration ($S = a$)

4. We read the following symbol and try to find the phrase $aa$. We are not successful now, so we output the index 1, add the phrase $aa$ into the dictionary and assign a new value to $S$

5. The next symbol $a$ is read, we find the current phrase $aa$ in the dictionary at the position 7, which is also output, because we are not able to find the phrase $aaa$ which was obtained by inputting the next character. Instead of, it is added into the dictionary and $S$ is initialized to the symbol $a$

6. In this step, we are able to read two symbols and thus get the phrase $aba$ found at the position 6. We save the phrase $abab$ in the next available entry (9)

7. The concatenation of string $S = b$ and the next symbol in the stream $x = a$ results into the phrase $ba$, which is found at the position 5

8. Now, we encounter the first occurrence of the symbol $c$, so it is evident that we cannot find the current phrase $bc$ in our dictionary

9. The next symbol is $a$, we add the phrase $ca$ into the dictionary and we abort the reading, since we do not have any symbol remaining in the stream. However, we still have the value in the symbol $s = a$, which needs to be also output (step 10)

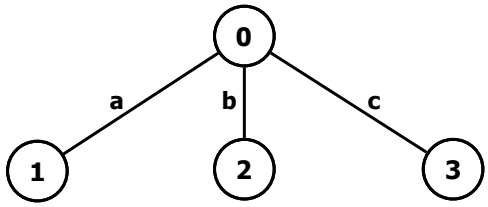
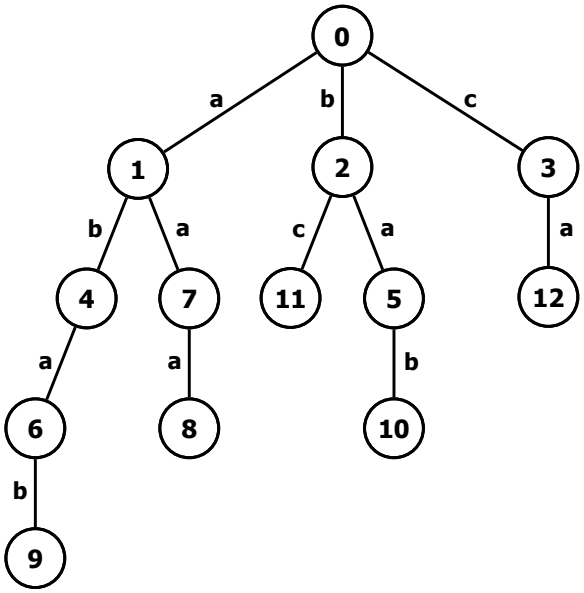



Figure 2.17: The LZW dictionary after its initialization

## 2.7 Summary

The purpose of this chapter was to provide a comprehensive view of data compression. We began with some preliminary terminology and basic notions, which we needed later for understanding the algorithms, which were introduced with regard to give an adequate overview of two main groups of the data compression schemes—statistical and dictionary-based methods.

Each of the methods was extended by an example illustrating its function. Since all of the presented methods are planned to be implemented within this thesis, the thorough description was provided to mention all interesting features. Chapter 3 deals with the implementation details.




| Step | Code | New index |
|---|---|---|
| 1 | 1 | 4 |
| 2 | 2 | 5 |
| 3 | 4 | 6 |
| 4 | 1 | 7 |
| 5 | 7 | 8 |
| 6 | 6 | 9 |
| 7 | 5 | 10 |
| 8 | 2 | 11 |
| 9 | 3 | 12 |
| 10 | 1 | 13 |

Figure 2.18: The final form of LZW dictionary tree with an additional table

# Chapter 3

# Implementation

This chapter covers the implementation details of all proposed compression methods described in Chapter 2.

## 3.1 Implementation details

As we mentioned in Section 1.4, the ExCom library is written in C++ programming language and thus the new methods should also be. Implementations of all compression algorithms are freely available in many different languages, including C++, and therefore they can be easily adapted to the ExCom library. However, we decided to implement them from scratch with a view to ensure the similar coding style. Furthermore, some methods (e.g. LZ77, LZSS) share the same idea and hence they can use the same data structures.

Similarly, as Filip Šimek decided, all source codes were written without using any Integrated Development Environment (IDE). The methods were integrated into the ExCom library according to the *Appendix F: Implementing new compression methods* in [3], however, some steps had to be modified, thus we provide some updated steps in Appendix F.

The Doxygen[1] documentation can be generated using the scripts provided in the ExCom library, based on the comments in the source codes according to the coding standards.

## 3.2 Supporting classes

This section briefly describes the classes which are used by some compression methods implemented within this work.

### 3.2.1 Universal codes

Sometimes it is useful to represent integer number in a different form which is suitable for our purpose. One of the solutions to this can be the Huffman coding proposed in 2.6. However, the usage of Huffman codes can be inconvenient because we need to know in advance the probability of occurrence of each symbol which appears in the input stream. Hence, the universal codes ensure the fixed codeword sets, i.e., the mapping of one integer always results into the same binary codeword. Small integers should be represented using less bits, while the large ones

[1] http://www.doxygen.org/, May 2010.

by more bits. Furthermore, the codes are also the prexif codes which simplifies the process of decoding. Hence, the `UniversalCodes` class was implemented to provide such codes.

**Definition 3.1 (Universal code)**
*Universal code* is a code satisfying the following formula:

$$L_{avg} \leq c_1 H_{avg} + c_2 \tag{3.1}$$

where $L_{avg}$ is the average length of given codeword, $H$ represents the average entropy (2.3), and $c_1$, $c_2$ are constants. We say, that the code is *asymptotically optimal*, when $c_1 = 1$.

The typical representatives for integer encoding are the *Elias codes* invented by *Peter Elias* [28]. He presented five different codes—$\gamma, \gamma', \delta, \omega, \omega'$. These codes can be obtained by calling the following functions, whose parameter is an unsigned integer and they return a sequence of bits as a string.

- `getGamma` for $\gamma$ code
- `getGammaPrime` for $\gamma'$ code
- `getDelta` for $\delta$ code
- `getOmega` for $\omega$ code
- `getOmegaPrime` for $\omega'$ code

Moreover, this class provides the usage of the *unary codes*, where the code for a given positive integer $n$ is defined as a sequence of $n-1$ ones (1) followed by a single zero (0), or *vice versa*, e.g., the number $n = 5$ can be represented either as `11110` or as `00001`. The length of such a code is $n$. We denote by $\alpha$ the unary code, thus the corresponding functions are named `getAlpha0` and `getAlpha1`. The output of the former function would be as `11110`, while the latter one would produce for the same number $n$ `00001` output.

Above this, the standard functions for binary-to-decimal, and decimal-to-binary conversions are available. The function `getBeta` (we denote by $\beta$ the binary numbers) can be called either with one or two paremeters. In the case of a single parameter, the returned value is without the leading zeros, on the other hand we use two parameters to controll the length of encoded number. Therefore:

`getBeta(15)` would produce `1111`
`getBeta(15, 6)` would produce `001111`

*Fibonacci codes* (see [29]) has been already implemented by Filip Šimek [3].

### 3.2.2 Sliding window

As we already know from the description of LZ77 and LZSS (2.6.4), the Sliding window is divided into two parts, the Look-ahead buffer and the Search buffer. These methods differ only in minor details. Hence, it is reasonable to have these buffers implemented only once, do not repeat the code, and thus to share them between these two dictionary-based methods. Furthermore, some more similar methods may be added in the future, so it is convenient to have it prepared.

The Look-ahead buffer is implemented as a **vector** template from C++ STL. The elements (in our case, of type `unsigned char`) can be inserted one by one, or as a block of bytes, which is preferable because at first we have to read more data from input stream file and then use them to fill the Look-ahead buffer, and it is obvious that reading more bytes at once is faster. The `remove` function removes $n$ buffer's elements from its beginning, the parameter $n$ corresponds to the number of positions the Sliding window is shifted to the right after each step of encoding.

The Search-buffer is implemented as a circular queue. And as we mentioned earlier in Chapter 2, it is a normal array with indices pointering to its beginning and to an end. Therefore, we can easily simulate the shifting. To obtain these indices, the `getBounds` function is provided.

## 3.3 Statistical methods

### 3.3.1 Shannon-fano coding

As we mentioned earlier, the binary tree is built to obtain the code for each symbol in the input stream. Therefore, the `ShannonFanoTree` class serves as the base of this algorithm. The leaves (nodes representing the symbol) are stored in a **map** (STL template), where a symbol as such is used as a key value to access the particular node. The leaves are also stored in another structure together with the rest of inner nodes. This approach was used for the reasons to keep the references to the leaves even if the the tree is built, thus to easily retrieve the codes. All nodes are implemented using the `ShannonFanoNode` class, where each node has a pointer to its children and to its parent. In the case of the leaf, the node also contains the frequency of occurrence and the symbol itself.

This method does not use the estimated probabilities, on the contrary the values are measured during the first pass through the file. As the encoder reads the symbols, the function `addSymbol` is used which automatically counts the frequency values. When the first pass is completed, the nodes are sorted in ascending order considering the symbol's occurrence, and the `buildTree` function is called to recursively generate the tree. The split point in each iteration is gained by the separation of both parts to be nearly the same with regard to a sum of frequencies. Since we use only the frequencies, not the probabilities, we do not have to use a division operation (symbol_freq / sum_of_freqs) and thus to make it faster.

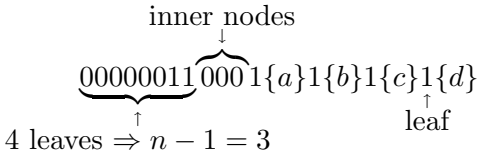
To decompress the file properly, it is necessary for the decoder to know the codes assigned to all processed symbols. We have several alternatives how to save the requiered information. David Salomon in [4] proposed two options how to write the probabilities or frequencies, as side information, on the compressed stream:

1. The values of frequencies can be written as integers, and the probabilities can be output as scaled integers. This approach results into few more hundred bytes to be output.

2. The second way is to write the variable size codes. The disadvantage is evident, the lengths of codes may vary, thus it is not easy to process.

However, we decided to output the Huffman tree without the the frequencies or probabilities, since it is enough to know only the codes of input symbols. We will try to explain this option. We have several processes to visit each node in the tree, we call it *tree traversal*. This includes postorder, inorder, or preorder traversal. The last one mentioned is the way we choosed to save the tree. We start in the root, then traverse the left subtree and continue with the right subtree traversal. When the inner node is reached, we output 0. In the case of the

leaf, a bit of value 1 is output followed by the 8 bit value representing the symbol. However, we still need to be aware of the number of leaves to know where the tree ends, and the normal compressed part begins. Therefore, the first 8 bits in the output stream represent the number of leaves (symbols). Because we have $2^8$ possible symbols in the input alphabet, and the number 256 needs 9 bits to be represented, we store the number in $n-1$ form. So we save one more bit. The whole process is simulated in Figure 3.1. The bit stream representing the Huffman Tree would be as follows:



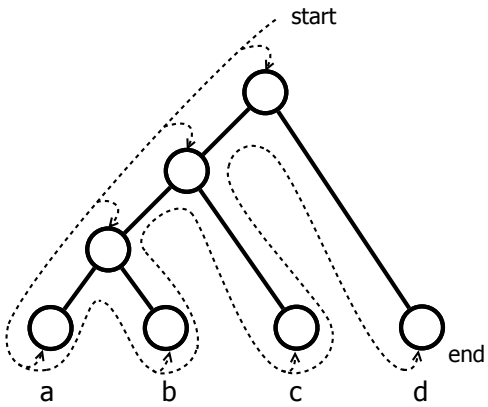




Figure 3.1: Preorder traversal of the Huffman tree

To obtain the traversed tree, we call the `getTreeAsString` function, which returns the bit sequence as a string. We denote by $\{d\}$ the 8 bit representation of a symbol. Every tree, with at least 2 leaves, needs $[(2 \cdot n_L - 1) + n_L \cdot 8]$ bits to be stored this way, where $n_L$ is the number of leaves. In the case of a tree containing only one symbol, the number of bits would be 10—1 bit for the root, 1+8 for the symbol. To control, if the file is properly decompressed, the compressed file starts with the fixed 32 bit header containing the original file size.

The compression then continues with going through the input stream again and each character is replaced with the corresponding bit representation until the encoder reaches the end of the file.

When decompressing, we use the same formula as above to find out the required number of bits to reconstruct the tree. For this purpose, the function `reconstructTree` is provided. The input stream is then read bit by bit until we match some symbol, which is output. To find a match, we start at the root and according to the current bit we go either to the left subtree or to the right one, until some leaf is reached.

### 3.3.2 Static Huffman coding

The Huffman coding is implemented in very similar manner as the Shannon-Fano coding was. It only differs in the way the tree is built. As we already know, the tree is constructed in a *bottom-up* approach. Hence, we start with the set of unattached leaf nodes with the measured frequencies assigned. The we repeat the process of joining two nodes with the lowest frequencies

until only one node remains, the root. The parent of these nodes is put back into the proper place in the set. To achieve this, it is ideal to use the *priority queue*. Such a container is available in STL templates, we only have to implement the `Compare` operator to define the comparision of the nodes.

The rest of the process is almost the same. We also ouput the original size of the file, and traverse tree Huffman tree as in the Shannon-Fano implementation.

### 3.3.3 Dymamic Huffman coding

Both the encoder and the decoder maintain and update the Huffman tree in the same way. Hence, the tree representation does not have to be stored in the output stream, since it is constructed as the decoder passes through the file. After a check, whether the symbol was already encountered, the `updateTree` function is called with the index of the last node processed as its paremeter. This function traverses up the tree until it reaches the root node. If the violation of sibling property is detected, the `swapNodes` function is called to reorganize the current tree. This function is called with two arguments—indices of nodes to be swapped.

## 3.4 Dictionary methods

### 3.4.1 LZ77 & LZSS

The *dictionary* was introduced in Chapter 2, and its implementation earlier in this chapter. We decided to abandon the idea to implement the original concept of dictionary searching, because a strict sequential search was noticeably slow for practical use. To ensure the fast search in the dictionary, we maintain a special list for each symbol. The list values correspond to the indices in the Search buffer, where the appropriate symbol can be found. We update these lists always whenever the Sliding window is shifted, i.e., some symbols are inserted into the Search buffer and the same amount of symbols disappears. For this purpose, an associative array, the **map** from STL containers, was used. The key value is a symbol itself, and is associated with the list of indices. In this case, the **deque** (**d**ouble-**e**nded **que**ue) was chosen, since we can access its elements through random access iterators, and insert and remove them in an efficient way. Furthermore, the elements are kept ordered. This container is named as `sBufferIndices` and its role is illustrated in Figure 3.2.

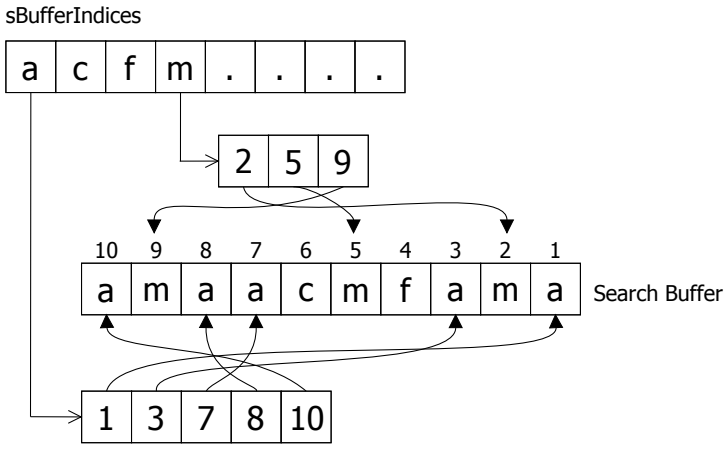



Figure 3.2: The usage of sBufferIndices

We go through the indices for the purpose to find the longest match, and thus to visit only the positions where the symbol really occurs. So, for example the symbol `m`, as shown in Figure 3.2, needs to be searched at three positions only instead of 10, which is the size of the Search buffer. In pratice, the size is multiple size bigger, so the speedup is more significant. Notice, that the search process can be aborted when the length limit for the match is reached. In the case of LZ77, this limit is typically set to the size of the Look-ahead buffer - 1.

Since the Search buffer is implemented as a cirucalr queue, the index of the recently added item may be for example 8, while we have much more items in it. Therefore, the values of indices are the real positions in the buffer as the symbols were input. In other words, every symbol in the Search buffer keeps its index value for the whole time it is present in the Search buffer. Hence, before the index is used in a triplet, it must be mapped to an alternative value corresponding to the original concept—the number of positions from the right.

The LZ77 variant always outputs a triplet of a fixed length, which depends on the lengths of both buffers. The approach of LZSS is different, the encoder makes a decision, what should be output, on the basis of the `minLength` parameter. When a match of string is shorter than this threshold, the single bit of value 1 is output followed by the ASCII representation of the first symbol in the Look-ahead buffer. We did some experiments with the Search buffer of size 4096 bytes, and with the Look-ahead buffer capable to contain 15 bytes. Therefore, the total size of one pair would be $12 + 4 = 16$ bits[2]. Since the single byte is encoded with 8 bits, a pair for a phrase of length 2 makes no compression. Hence, `minLength` was set to 3. Furthermore, we can make the best of using this parameter, i.e., the length field in a token can be extended up to 18, because before we output a token, we substract the minimal length from the real length of the longest found match. When decompressing, we add the same value to retrieve the original value. Thus, we still use only 4 bits and the compression ratio may be improved.

### 3.4.2 LZ78 & LZW

Both algorithms were thoroughly described in Chapter 2. However, the most remarkable part is the way we search the phrases in the dictionary. Salomon in [4] proposed a *hashing process* to do this. The original versions of both, LZ78 and LZW, were implemented using this technique. However, we found this approach to be slow. Therefore, we decided to do a research with regard to find a better way. *Juha Nieminen* on his website ([30]) comes with an article *An efficient LZW implementation*. We consider the both techniques to be interesting, thus we present the searching using a hashing process and Nieminen's modified version, too. They both suppose the tree nodes to be stored in an array (or in a similar data structure).

#### 3.4.2.1 Dictionary search using a hash function

According to the examples 2.4 and 2.5, in each encoding step we try to find a phrase, consisting of either a *prefix* or a *string*, and a new incoming symbol, in the dictionary. Such a dictionary entry can be represented with three fields—*prefixIndex*, *index*, and *symbol*. The *prefixIndex* pointers to a parent string, *index* is an address of a new phrase (a dictionary entry as itself), and a single character is stored in a *symbol* field. Notice, that the *index* and a real position in an array may vary. This situation may occur due to the collisions produced by the hashing function. We will explain this using the following steps which are necessary to use this technique.

Whenever we have a string $S$ and a current symbol $x$, we use the hash function to obtain an index to array. Such a function can be implemented in several ways, while the ideal hash

[2] Obviously, we need to output one more bit to distinguish between two types of tokens in the case of LZSS.

function would produce a minimal number of collisions which slow the search process down. One of the options is for example this formula:

```
((stringIndex << 19) | (currentSymbol << 7)) % DICTIONARY_SIZE
```

where `<<` is *bitwise left shift*, `|` is *bitwise OR*, and `%` stands for *modulo operation*. This function produces values in the range from 0 to (DICTIONARY_SIZE[3] - 1) because only at these positions we can save the dictionary phrases. The previously encountered string is represented with `stringIndex` as an integer value. There are three possibilities which may occur after the encoder hashes the string $S$ and a current symbol $x$:

- The node at the obtained index is unused, which means that the phrase is not in the dictionary, thus we save it
- The dictionary entry is used, and at the same time a string $S$ is equal to the string at *prefixIndex* field and also the *symbol* field is identical to a current symbol. Therefore, the phrase was found in the dicitonary, the string $S$ is extended by a concatenation $S \cup x$ and the next symbol is read
- The similar situation as in the previous option happens, but the node fields do not correspond to the current phrase values—it signifies a collision. We continue in the search by incrementing the obtained index until we found the first unused entry or we find a match for the phrase

#### 3.4.2.2 Dictionary search using a binary tree

In this case we maintain a structure consisting of 5 fields for each node in a non-balanced tree.

- **prefix index**—a pointer to a prefix string of a dictionary phrase
- **symbol**—a single character
- **first**—a link to the first phrase using this string as prefix
- **left**—a link to the next phrase using the same prefix as this one, and whose symbol is smaller (comparing ASCII values)
- **right**—a link to the next phrase using the same prefix as this one, and whose symbol is larger (comparing ASCII values)

The search process is simple then. Suppose we have a string $S$ and a symbol $x$. We find a node for $S$[4] and look at the *first* field. If it is empty, the phrase $S \cup x$ is not in the dictionary, thus we add it. If the field is occupied with a pointer to the next node, there is at least one phrase, whose prefix is $S$. Thus, we move to this node and according to the value of $x$ we follow a link in either a *left* or in a *right* field. We continue, until we are able to find the next node.

To make it clear, we propose an example of a simplified LZW dictionary in Figure 3.3. In addition to the initialized entries (bytes of values 0–255), the dictionary contains the following phrases: *ea*, *eb*, *es*, and *fc*. Figure 3.4 demonstrates the same dictionary using the special structure for tree nodes. Notice, that for simplicity we present only a part of the dictionary,

[3] Typically this value is between 1,024 and 8,192.

[4] It is easy since we keep the string $S$ as an integer value.

and that the indices of phrases are arbitrary. The items in each node are in the following order: *position in the dictionary*, *prefix index*, *symbol*, *first*, *left*, *right*.

The symbols $d$, $e$, $f$ are single characters, thus their *prefix index* is set to `null`. In Figure 3.3 this situation is illustrated with a link to the root node. The symbol $d$ is not prefix of any other string in the dictionary, thus its fields *first*, *left* and *right* are set to `null`. Then we have a phrase $fc$, so its *first* field pointers to the node at position 320. Suppose the string $S = e$ and the symbol $x = s$. Thus, we start at position 101, since it is a position of $S$ (ASCII value of $e$). The value in the field *first* is set to 270. We move to this node and compare the value of *symbol* with symbol $x$. The symbols are different, and $x$ is bigger than the *symbol* field in sense of ASCII values. Hence, we follow the node, whose index is in the *right* field. It is a dictionary entry 360, and both symbols are equal now. The process for the phrase $ea$ is analogous.

We add the items sequentially in contrast to the previous approach with hashing. Hence, we can take full advantage of the variable-sized codes as it was described in 2.6.7.

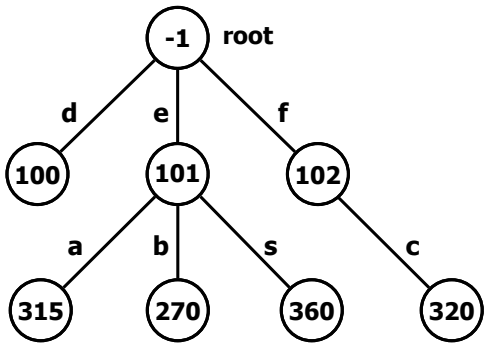



Figure 3.3: Example of LZW dictionary

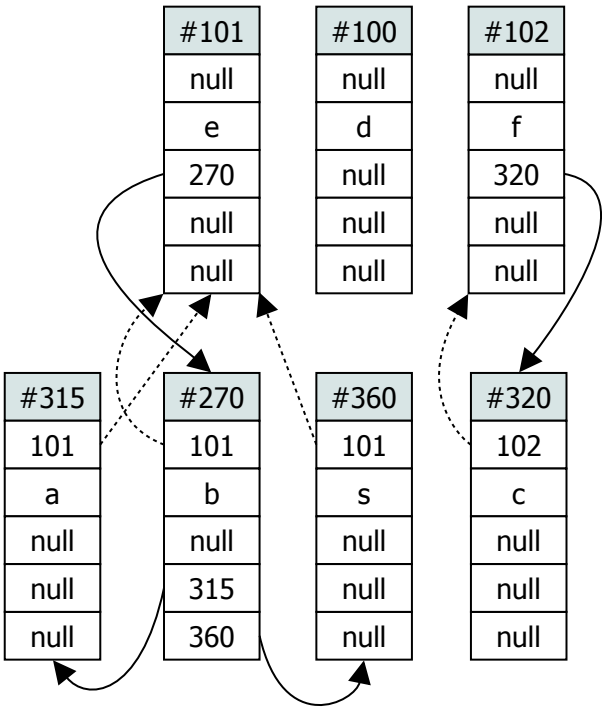



Figure 3.4: Illustration of tree nodes with the corresponding relations

## 3.5 Additional notes

For more information of how to use the compression methods, see Appendix F. All implemented methods and the related files are stored in the following locations:

| | |
|---|---|
| **Shanon-Fano coding** | `lib/method/sfano` |
| **Static Huffman coding** | `lib/method/shuff` |
| **Dynamic Huffman coding** | `lib/method/dhuff` |
| **LZ77** | `lib/method/lz77` |
| **LZSS** | `lib/method/lzss` |
| **LZ78** | `lib/method/lz78` |
| **LZW** | `lib/method/lzw` |

Furthermore, the classes for the Search buffer, Lookahead buffer, and the Universal codes are located in:

| | |
|---|---|
| **Search buffer** | `lib/method/lz_common` |
| **Lookahead buffer** | `lib/method/lz_common` |
| **Universal codes** | `lib/method/universal_codes` |

# Chapter 4

# A new corpus

In this chapter we describe the development of the new corpus and the design of methodology to maintain its timeliness. We begin with a detailed description of the existing solutions—we will mention their features and give an overview of the pros and cons. Based on the reasearch we will try to come up with our solution with a tendency to overcome some drawbacks. But, in fact, it is not our intention to create the new corpus which would still be used in next two decades. For that reason, we mainly focus on the methodology consisting of the set of steps verifying the up-to-date status of the corpus, and in case of need to provide a tool how to easily update the files included in the corpus.

## 4.1 General purpose

Data compression methods differ a lot. We have several factors indicating a quality of an algorithm to monitor. Some can perform faster than other algorithms, on the other hand, their compression ratios may be worse. It depends on the situation when space efficiency is (or is not) an issue which method should we use. In the case of asymmetrical methods, the encoding time of some file is not the same as the time required for its decompression. Hence, it is very important to have *something* which can be easily used to objectively evaluate the performance of various compression schemes. Corpora—collections of files used as a benchmark for comparing lossless compression methods—offer a solution to this problem. However, it is not trivial to select the appropriate file candidates which would behave like an average representantive of each group of file types. We use the term *average* because, in our opinion, the algorithm should give the quality results when compressing files likely to occur. So far, we have only two main corpora used by authors of papers dealing with the field of data compression. Several others exist, though, they are not so frequent. Each corpus comes with its own features and the area of usage. The purpose of the following section is to cover the most of them and make some conclusions.

## 4.2 Existing corpora

### 4.2.1 Calgary Corpus

The Calgary Corpus, which was founded by *Ian Witten*, *Tim Bell* and *John Cleary* at University of Calgary (Alberta, Canada) in 1987, is the most referenced corpus and has become some kind of standard for comparing the lossless data compression methods. However, it was firstly

published in their paper *Modeling For Text Compression* in 1989 [31], and it was also included in a book [21]. It consists of 14 files (9 different types), but there are also 4 other files which are not included in the corpus anymore, becuase these files were not evaluated in the paper and in the book. Since it used to be the only one data set officially available, it was frequently used in the 1990's. In 1997, the corpus was replaced by the Canterbury Corpus. All the files, excluding the 4 mentioned files (paper3, paper4, paper5 and paper6), are described with their sizes in Table 4.1. This collection was made to cover the typical files used in that time.

| File | Size [Bytes] | Description | Type |
|---|---|---|---|
| bib | 111,261 | Collection of 725 bibliographic references | Text |
| book1 | 768,771 | Hardy: *Far from the madding crowd* (fiction book) | Text |
| book2 | 610,856 | Witten: *Principles of computer speech* (non-fiction) | Text |
| geo | 102,400 | Geophysical seismic data | Data |
| news | 377,109 | A batch of unedited news articles | Text |
| obj1 | 21,504 | Executable file for VAX: compilation of progp | Binary |
| obj2 | 246,814 | Executable file for Apple Macintosh: "Knowledge support system" program | Binary |
| paper1 | 53,161 | Witten, Neal and Cleary: *Arithmetic coding for data compression* (technical paper) | Text |
| paper2 | 82,199 | Witten: *Computer (in)security* (technical paper) | Text |
| pic | 513,216 | A bit-map black and white picture number 5 from the CCITT Facsimile test files (page of textbook) | Image |
| progc | 39,611 | C source code: unix utility `compress` version 4.0 | Source |
| progl | 71,646 | Lisp source code: system software | Source |
| progp | 49,379 | Pascal source code: program to evaluate compression performance of PPM | Source |
| trans | 93,695 | Transcript of a terminal session | Text |
| Total | 3,141,622 | | |

Table 4.1: The Calgary Corpus files

### 4.2.2 Canterbury Corpus

The Canterbury Corpus was published in 1997 by *Ross Arnold* and *Tim Bell* [32] with regard to overcome the issues connected with the Calgary Corpus. The authors of the Canterbury Corpus mention that the new compression methods were too much tuned to the Calgary collection and therefore achieving only the small improvements. At the same time, the files were chosen without any elaboration so the results may not reflect the real qualities of the algorithms. Furthermore, the corpus was getting old and with the rise of Internet the computer files had changed. Hence, the authors presented a technique for selecting the files which were included in their corpus.

The development started with a collection consisting of almost 800 files of different types and sizes. With that, the files were divided into the groups according to their type (e.g. executables, HTML), and each file in the group was compressed with several compression algorithms (e.g. compress, gzip) with a purpose to obtain a scatter plot, where the original file size and its compressed size in bytes were monitored. Then, the linear regression line was fitted to the collected data, whereas the representative of each group was chosen upon the lowest distance from the line. The *best* file was stated this way, however they admit that file

can deviate from the average one. This process resulted into 11 distinct files of a total size 2,810,784 bytes, see Table 4.2. When you look at the file *pic* in the Calgary Corpus, and at the file *ptt5*, respectively, and compare their sizes, description and the data type, you would notice that these files are identical[1].

The corpus collection was also designed to satisfy the following conditions. The files should be all public domain, and the total size should not be bigger than necessary to ensure the seamless distribution. Furthermore, the corpus should be available (published on Internet for everyone) and useful in sense of that the compression methods should give the similar results on both corpus files and the other ones.

When we compare the sizes of both mentioned corpora, we see a drawback of the newer data set which is smaller than its predecessor, while the files sizes increased a few times. This issue was partially solved with publishing of the *large Canterbury Corpus* which included three more files whose total size was bigger than the sum of both the Calgary and the Canterbury Corpus together. The files are listed in Table 4.3. *Sebastian Deorowicz* in his dissertation thesis [33] also points to the file kennedy.xls, which seems to be controversial to him because of its specific structure causing the very different results of various compression schemes.

| **File** | **Size [Bytes]** | **Description** | **Type** |
|---|---|---|---|
| alice29.txt | 152,089 | Carroll: *Alice's Adventures in Wonderland* | Text |
| asyoulik.txt | 125,179 | Shakespeare: *As You Like It* | Text |
| cp.html | 24,603 | Compression Pointers | HTML |
| fields.c | 11,150 | C source code | Source |
| grammar.lsp | 3,721 | Lisp source code | Source |
| kennedy.xls | 1,029,744 | Excel spreadsheet | Binary |
| lcet10.txt | 426,754 | *Workshop on electronic texts*, edited by James Daly | Text |
| plrabn12.txt | 481,861 | Milton: *Paradise Lost* | Text |
| ptt5 | 513,216 | A bit-map black and white picture number 5 from the CCITT Facsimile test files (page of textbook) | Image |
| sum | 38,240 | Executable object code for SPARC | Binary |
| xargs.1 | 4,227 | A Unix manual page | Text |
| Total | 2,810,784 | | |

Table 4.2: The Canterbury Corpus files

| **File** | **Size [Bytes]** | **Description** | **Type** |
|---|---|---|---|
| e.coli | 4,638,690 | Complete genome of the *Escherichia coli* bacterium | Binary |
| bible | 4,047,392 | The King James version of the bible | Text |
| world192.txt | 2,473,400 | The CIA world fact book | Text |
| Total | 11,159,482 | | |

Table 4.3: The Large Canterbury Corpus files

[1]The files were compared with `diff` command and no differences between them were found.

### 4.2.3 Silesia Corpus

The Silesia Corpus was proposed within the dissertation thesis of *Sebastian Deorowicz* at Silesian University of Technology (Gliwice, Poland) in 2003 [33]. Author mainly concentrated on the disadvantages of existing corpora. These include the lack of large files because nowadays we need to store more and more data, a representation of English texts only whereas more different languages is used, absence of medical images, and the lack of data (typically databases) tend to grow rapidly over time.

Deorowicz collected 12 various files of a total size more than 200 MB, a more detailed description of these files is presented in Table 4.4. In our opinion, the main disadvantages of the corpus are the mentioned size and that the files were chosen without any sophisticated method denoting that the files are appropriate candidates for evaluating of compression methods. Even if we have the high-speed access to the Internet, we should stay with an idea of the authors of Canterbury Corpus to keep the data set easily available. Furthermore, we do not find useful to concatenate a number of files even if they are of the same type. We also think, that it is necessary to perform at least some basic tests before considering the files to be convenient.

| File | Size [Bytes] | Description | Type |
|---|---|---|---|
| dickens | 10,192,446 | A concatenation of 14 novels by Charles Dickens | Text |
| mozilla | 51,220,480 | Tarred directory of installed web browser Mozilla | Binary |
| mr | 9,970,564 | A magnetic resonance medical picture of a head | Image |
| nci | 33,553,445 | The chemical database of structures | Database |
| ooffice | 6,152,192 | A dynamic-link library (DLL) of Open Office | Binary |
| osdb | 10,085,684 | An Open Source Database Benchmark (MySQL) | Database |
| reymont | 6,627,202 | Reymont: *Chłopi* | PDF |
| samba | 21,606,400 | Tarred source code of an open source Samba project | Source |
| sao | 7,251,944 | Data of sky objects | Binary |
| webster | 41,458,703 | The 1913 Webster Unabridged Dictionary | HTML |
| xml | 5,345,280 | A concatenation of 21 XML files | HTML |
| x-ray | 8,474,240 | An X-ray medical picture of child's hand. | Image |
| Total | 211,938,580 | | |

Table 4.4: The Silesia Corpus files

### 4.2.4 Other corpora

To partially complete the list of existing corpora, we should also mention the following less important collections:

- **The Lukas Corpus**—this corpus consists of 4 parts used to evaluate the algorithms in the medical imaging field. The 401 files (total size = 212,764,094 bytes) were collected by *Professor Dr. Rainer K ster* from the University of Duesseldorf (Germany)
- **The Protein Corpus**—set of 4 protein files (total size = 7,154,401 bytes) used in the article *Protein is incompressible* by *Craig Nevill-Manning* and *Ian Witten* in 1999 [34]
- **The Artificial Corpus**—this corpus includes 4 files (total size = 300,001 bytes) representing the worst possible example, e.g., a file containg only a letter 'a', repeated 100,000 times

- **The Miscellaneous Corpus**—there is only one file in this corpus (total size = 1,000,000 bytes) containing the first million digits of $\pi$
- **Ekushey-Khul Corpus**—a new corpus for evaluation of Bengali text compression schemes, see [35] for more information

## 4.3 Data file classes analysis

This section deals with the classification of computer files with regard to divide them into appropriate groups. Each group should be represented in the planned corpus with at least on file. These files should be selected carefully to reflect the commonly used data types.

### 4.3.1 Text and binary files

In fact, all computer files can be considered to be *binary* since they are stored as a collection of *0*s and *1*s. However, some of them can be classified as *text* files. These simple data files consist of a series of lines containing only unformatted text[2], which is readable to a human. By defintion, the lines must be terminated. To represent an end of line, the special characters *carriage-return* (CR), *line-feed* (LF), or their combination CR+LF is used. A typicall text file is stored in *ASCII* format, with each character assigned the standard code. The standard ASCII set uses only 128 different symbols represented with 7 bits. Though, some larger characters sets exist with regard to support non-English characters from various alphabets. One of these extensions is *UTF-8*, which is backward compatible with ASCII, so the codes for latin alphabet remain the same. It is achived by using from 1 to 4 bytes to represent a single characeter, while the characters equal to or below a value 127 (ASCII) are represented with 1 byte.

Beside the readable documents, some other types belong between the common text files. This includes the web pages written in *HTML*, and the *source codes* of different programming languages consisting of instructions, statements and declarations in human-readable format. In contrast to plain text files, the structure of source codes is more specific because of their purpose, which is to be converted by a compiler or interpreter to a binary form—*object code*. One of the disadvantages is a low entropy which causes that the text files need more space to be stored than necessary.

A binary file is such a computer file which may contain any type of data, including text, images or audio. The binary files can be defined as a sequence of bytes (256 possible values), whose order determines the file purpose and how it is treated. Most of computer files are of binary types since it allows more complex structure than the plain text files.

### 4.3.2 Commonly used data files

There are many various fields where we can use the computers. Each area brings the different computer files of different purposes, however the idea of reducing their sizes remains. Therefore, to evaluate a compression method and make some conclusions about its effectiveness, we should include into the corpus as many as possible of files to cover up the whole range of *file formats*.

**Definition 4.1 (File format)**
*File format* specifies how the digital information is stored and organized within a file.

[2]No fonts or emphasis can be used in plain text files. We can achive the relative formatting by using empty lines between paragraphs or by inserting the additional space between words.

The format of file is often determined based on its extension[3]. If the file extension is not provided, the file can be recognized upon the reading its file header containing some important information typically located at the beginning and is either in a binary or in plain text format.

Based on results from the research, we created several groups and sub-groups that should be enough to include the most common file types and to meet the requirements in the future.

- **AUDIO**—the audio files are either compressed or uncompressed, while the methods can be both lossless and lossy. For the purpose of our corpus, we decided to include only lossless audio files in their uncompressed form. Such formats are for example `WAV` and `AIFF` files. The compressed versions of both exist, but we consider to be reasonable to use only the basic version to evaluate the algorithms.
- **BINARY**—this category contains several other subgroups, i.e., *executables*, *object codes* or *libraries*.
- **DATABASE**—the files belonging here can be in binary or in text form, however their common feature is that they contain different types of data (i.e. numbers, text, images). Hence it is suitable to use them.
- **DOCUMENTS**—we regard the documents as the files containing plain or formatted text. Additionaly, they can contain some other multimedia, i.e., images as in the case of `DOC` or `PDF` files.
- **GRAPHICS**—the images are typically compressed. However, some fields do not allow the usage of lossy compression methods. This is characteristic for *medical images*, where some important pixels, for example indicating cancer, must remain unchanged. Therefore, we should choose either the images with no or with lossless compression. The images can be divided to *raster*, *vector*, and *3D* categories.
- **MARKUP LANGUAGES**—the markup languages annotate the text in a specific way to process it easily later. The well-known examples are `HTML` and `XML`. We consider this group to be so specific, and thus not to insert the representatives to the *scripts* category.
- **SCRIPTS**—this group involves the *source codes* and *scripts* of various programming languages.
- **SPREADSHEETS**—the data arranged in cells of one or several tables.

We decided not to include the *video* files, since they are rarely used as uncompressed, which is evident for the size they occupy. Furthermore, some other categories were obtained during the research, i.e., *scientific* (math, chemistry, biology), *GPS*, or *video games* data (game engines). However, they can be put into some of above mentioned groups.

## 4.4 The design of methodology

This section thoroughly describes all the steps which were necessary to obtain the set of files that form the new corpus. The following procedure is a concrete version of the planned methodology, which is generalized in Appendix A. In the case of need for updating the corpus, the reader should go through that formalized steps.

[3] A short sequence of letters (usually 3 letters) appended to a file name.

### 4.4.1 Obtaining data and further preprocessing

All of here mentioned corpora contain very similar types of data, and as we stated earlier, a new corpus should include various data. Hence, we strictly follow the file categories proposed in the previous section. The second task was to get as many files as possible. Notice, that all files should be in the public domain, or their licence should allow the file to be freely distributed. Otherwise, the corpus would not be available for others to use.

Some data types (i.e. HTML, source codes) can be easily found in many different forms. On the other hand, for example, the free databases are not so frequent. We focused on governmental and non-governmental organizations providing data in many formats placed in the public domain. Then, we tried to find projects specializing in maintaining the collaborative databases of certain data, such as audio. We also explored some national and international statistics offices with large data sets. All sources and some other useful tips how to find the needed files are listed in Appendix D.

As a result of this, we have collected approximately 1,250 distinct files of total size more than 2 GB. The file sizes ranged from few bytes to tens of megabytes. Some categories (i.e. source codes) contain small or average sized files, in contrast to this the audio or graphics categories contain rather the large ones. In the case of documents and text files we tried to cover more language families to avoid the representation of English texts only, as it happend during a creation of the Canterbury corpus. Hence, the text candidates we written, for example, in Czech, German, Swedish, Italian, or Gaelic.

All files were classified into eight proposed categories, furthermore they were put into subcategories according to their extension, i.e., Documents—`TXT`, `DOC`, ..., Scripts—`CPP`, `JAVA`, .... Since the `PDF` files are internally compressed with the Deflate algorithm, we decided to uncompress them before using them in experiments. For this purpose, we used a handy tool `pdftk`[4] for `PDF` files manipulation. This was done due to find out an alternative compression scheme for PDF files providing better results. The candidates were then compressed with three different command-line compression tools provided in Linux:

- `bzip2`[5]—it uses the Burrows-Wheeler block sorting text compression algorithm together with the Huffman coding.
- `gzip`[6]—this program uses LZ77 dictionary method.
- `lzma`[7]—the principle of this tool is based on the modified LZ method called Lempel-Ziv-Markov chain-algorithm.

All tools were used with their default settings, thus no parameters to achieve the better results or to speed the encoding process up were set. To manipulate with such a large number of files, we created several supporting bash scripts to process the files in a batch mode.

### 4.4.2 Post-processing of results

Our aim was to find the files, which would behave as the average ones over the test data set. Therefore, we used a similar technique as the authors of the Canterbury Corpus presented. We

[4] `http://www.pdftk.com/`, May 2010.
[5] `http://www.bzip.org/`, May 2010.
[6] `http://www.gzip.org/`, May 2010.
[7] `http://tukaani.org/lzma/`, May 2010.

monitored only the compressed size, thus we had four values for each file available—its size before compression and the values obtained from three tools. For each subcategory, consisting of at least 15 files, a scatter plot was generated with the set of file sizes before compression on $x$-axis, and with the corresponding size values after compression on $y$-axis. Since the characteristics of files in appropriate subcategories were similar, the linear regression line could be fitted to the plot. Then, we established the file which was closest to the line—this was calculated as the square of the distance below or above the straight line. This was repeated for each of three tools, so we obtained three representative files for each subcategory. However, we wanted to keep the number of corpus files manageable, hence we found useful to include only one of these files into a corpus. To decide which candidate is the most suitable, we simply compared their compression ratios with regard to the lowest value. The example of scatter plot is shown in Figure 4.1, where each point represents one file.

Furthermore, to take advantage of all files (even of those whose number of representation is less than 15), we constructed the same plot for each category. In other words, we merged all subcategories within one group and constructed another scatter plot. Thereby, we gained few more files for a corpus.

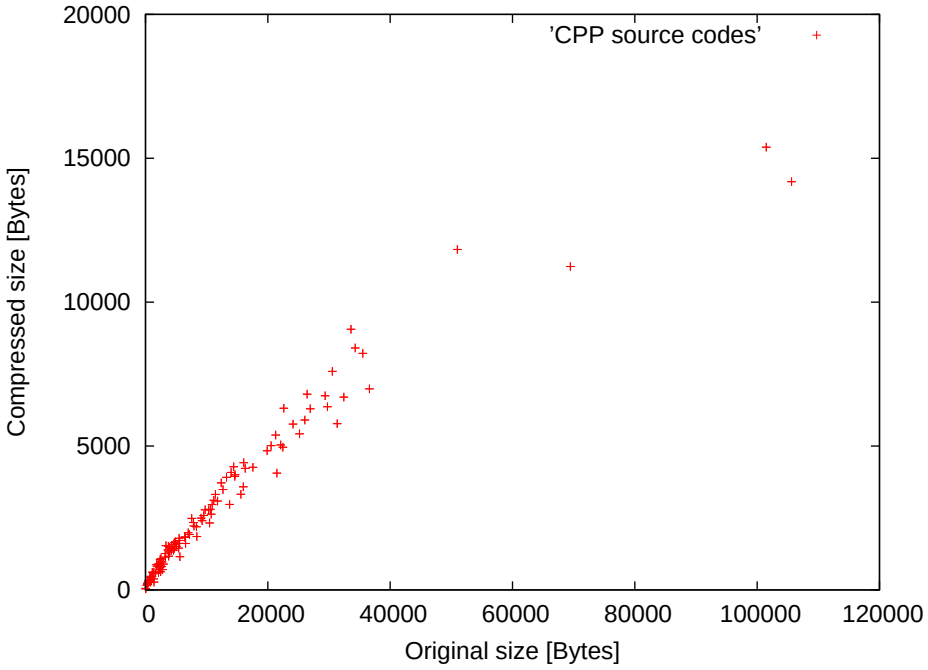



Figure 4.1: Scatter plot for CPP source codes, compressed with `lzma` tool

## 4.5 Corpus files

Based on the analysis and experiments in the previous sections we propose the *Prague Corpus* (Table 4.5) consisting of 30 files of a total size of around 60 MB. Some formats are represented with two files, however, there are 23 various data types classified in 8 groups. As you can see, some files may contain similar data, i.e., files *cyprus* and *hungary*. However, this was not intended, we tried to obtain data from many sources to achieve a variable collection. We wanted to strictly follow the results of experiments, therefore this occurred during the development of the set of files. The file size does not reflect an avarage length of the compressed file. This is more discussed in Chapter 5.

| File | Size [Bytes] | Description | Type |
|---|---|---|---|
| firewrks | 1,440,054 | Sound of fireworks | Audio |
| thunder | 3,172,048 | Sound of thunder | Audio |
| drkonqi | 111,056 | KDE crash handler | Binary |
| libc06 | 48,120 | A dynamic-link library | Binary |
| mirror | 90,968 | A part of the software package | Binary |
| abbot | 349,055 | Part of interior design application | Binary |
| gtkprint | 37,560 | A shared object | Binary |
| wnvcrdt | 328,550 | A database file | Database |
| w01vett | 1,381,141 | A database file | Database |
| emission | 2,498,560 | Waterbase emissions data | Database |
| bovary | 2,202,291 | Gustave Flaubert: *Madame Bovary*, in German | Documents |
| modern | 388,909 | Axel Lundegård, Ernst Ahlgren: *Modern. En berättelse*, in Swedish | Documents |
| ultima | 1,073,079 | Mack Reynolds: *Ultima Thule*, in English | Documents |
| lusiadas | 625,664 | Luís Vaz de Camões: *Os Lusíadas*, in Portuguese | Documents |
| venus | 13,432,142 | Ultraviolet image of Venus' clouds | Graphics |
| nightsht | 14,751,763 | A photo of a city at night | Graphics |
| flower | 10,287,665 | A photo of a flower | Graphics |
| corilis | 1,262,483 | CORILIS land cover data | Graphics |
| cyprus | 555,986 | Air Quality Monitoring in Cyprus | Markup languages |
| hungary | 3,705,107 | Air Quality Monitoring in Hungary | Markup languages |
| compress | 111,646 | Wikipedia page about data compression | Markup languages |
| lzfindmt | 22,922 | C source code from a file archiver | Scripts |
| render | 15,984 | C++ source code from an action game | Scripts |
| handler | 11,873 | Java source code from the GPS tracking system | Scripts |
| usstate | 8,251 | Java source code from the GPS tracking system | Scripts |
| collapse | 2,871 | JavaScript source code from the project management framework | Scripts |
| xmlevent | 7,542 | PHP source code from the calenedar generator | Scripts |
| mailflder | 43,732 | Python source code from the ECM framework | Scripts |
| age | 137,216 | Age structure in the world | Spreadsheets |
| higrowth | 129,536 | Financial calculations | Spreadsheets |
| Total | 58,233,774 | | |

Table 4.5: The Prague Corpus files

## 4.6 Summary

This chapter was devoted to the development of a new corpus. We started with a research of existing corpora, which resulted into a design of the Prague Corpus, which should be easily updated, hence the methodology, how to proceed, is presented in Appendix A. We also found suitable to create a report whose purpose would be to summarize all important steps during the update. Therefore, the we come up with a template for such a report, see Appendix B.

# Chapter 5

# Experimental measurements

The purpose of this chapter is to discuss the results of all performed measurements and experiments using the testing application which is provided as default by the ExCom library. This application is very simple and is controlled via the command line parameters. One of its advantages is the built-in mechanism to measure the time consumed by each compression method. The mechanism uses the `clock_gettime` function representing the time with nanosecond precision. However, the time result is returned in microseconds by the ExCom library.

All compression methods were checked with *Valgrind*[1]. Valgrind is a complex tool used for a program analysis. It is capable of profiling, memory leaks detection and debugging. No erroneous behaviour and no ememory leaks were detected, thus all allocated resources were released when exiting. To prove the correctness of the encoding and the decoding process of our methods, we compared the decompressed file with the original one using the `diff` command.

## 5.1 Experimental details

The testing platform included the Intel® Core™2 Duo Mobile Processor P8600 with 2.40 GHz processor (64-bit architecture) and 3 MB L2 cache. The platform had 4 GB of main memory available. The operating system was Ubuntu 9.04 (Jaunty) with `2.6.28-18-generic` linux kernel version and the ExCom library and all implemented methods were compiled with `gcc 4.3.3`. To ensure the reliable results we switched off the *dynamic frequency scaling* feature, which adjustes the CPU speed during the time.

All files were compressed and decompressed 30 times by each of the methods to avoid the wrong results caused by the slowdown of system processes. However, we tried to stop all unnecessary applications and services. The lowest value of obtained compression and decompression times was taken for further processing.

The purpose of all proposed experiments is to prove the validity of the Prague Corpus and to point to some drawbacks of the Canterbury Corpus which seems to be obsolote. Moreover, the performance of our methods is no less important. For these reasons we performed all experiments on both Canterbury and the Prague Corpus. The compression and decompression times, and the achieved compression ratios are monitored for all methods, including the existing ones in the ExCom library, i.e., ACB, DCA and PPM.

Regarding to a relatively large number of files in the Prague Corpus (30), we decided to divide it into two parts. The *Set A* contains the representatives from *Audio*, *Binary*, *Database*,

[1] http://valgrind.org/, May 2010.

and *Graphics* groups. The *Set B* consists of files from *Documents*, *Markup languages*, *Scripts* and from *Spreadsheets*. We found this distribution to be reasonable, since the files in both sets share some features in common. Notice, that the results are discussed and described together.

## 5.2 Performance experiments

One of the aims of this work was to implement the new algorithms from the statistical and dictionary methods. The approach of both groups differ a lot, so we decided to provide the tests separately. We start with the Canterbury Corpus, then the results from the Prague Corpus testing will be presented. We will summarize the results at the end of this chapter.

### 5.2.1 Canterbury Corpus

Table 5.1 contains both compression and decompression times of all implemented statistical methods. As we can see, the Shannon-Fano coding is approximately three times faster than Static Huffman coding as regards the compression time. This can be caused due to the way the binary tree is built. When we compare the decompression times of these algorithms, the values are almost identical. The explanation is self-evident, both techniques use the same approach to reconstruct the tree and to obtain the codewords of input alphabet. This approach is useful with respect to a substitution of both methods when decompressing. In other words, a file encoded with Shanon-Fano coding can be decoded with the Static Huffman coding and *vice versa*. The Dynamic Huffman coding was proposed to overcome the issue of passing through the files twice as it is in the case of Shannon-Fano and Static Huffman coding. However, these results show that it is slower than the others. We will focus on the adaptive version of Huffman codes later in this chapter. The Figure 5.1 illustrates the measured values.

| | Compression time [$\mu s$] | | | Decompression time [$\mu s$] | | |
|---|---|---|---|---|---|---|
| File | SFano | SHuff | DHuff | SFano | SHuff | DHuff |
| alice29.txt | 56,493 | 149,733 | 312,498 | 21,139 | 33,074 | 317,133 |
| asyoulik.txt | 47,872 | 128,634 | 272,034 | 18,025 | 18,177 | 276,150 |
| cp.html | 9,930 | 27,181 | 61,635 | 3,890 | 3,900 | 62,537 |
| fields.c | 4,815 | 12,095 | 28,634 | 1,770 | 1,762 | 29,197 |
| grammar.lsp | 1,878 | 4,051 | 9,146 | 611 | 604 | 9,283 |
| kennedy.xls | 353,877 | 866,568 | 1,850,597 | 117,201 | 117,955 | 1,871,776 |
| lcet10.txt | 158,296 | 425,200 | 892,278 | 60,037 | 60,039 | 905,944 |
| plrabn12.txt | 177,316 | 471,620 | 968,729 | 66,236 | 66,309 | 989,280 |
| ptt5 | 138,228 | 242,907 | 405,418 | 28,239 | 28,921 | 394,261 |
| sum | 17,667 | 43,821 | 119,398 | 9,718 | 9,737 | 118,183 |
| xargs.1 | 2,089 | 7,467 | 10,867 | 1,125 | 1,130 | 17,385 |

Table 5.1: Compression and decompression times of the statistical methods on the Canterbury files

The compression and decompression times for the dictionary-based methods are listed in Table 5.2. At first glance, it is evident that LZ78 and LZW performs much faster than LZ77 and LZSS. It is caused due to the efficient technique described earlier in Chapter 3. Moreover, the data show that the LZ77 method and its derivate LZSS are asymmetrical methods, since we do not have to perform the longest match search during the decompression. Therefore, the

decoding process needs several times less amount of time than the encoding. Furthermore, the LZ78 and LZW compressed and decommpressed all files from the Canterbury Corpus in less than 100 ms. The comparison of four dictionary methods is shown in Figure 5.2 and 5.3 respectively.

The compression ratios of all implemented methods are presented in Table 5.3. The lowest (best) value in each row is highlighted with a grey background. Except for some values, LZSS methods gives the best results.

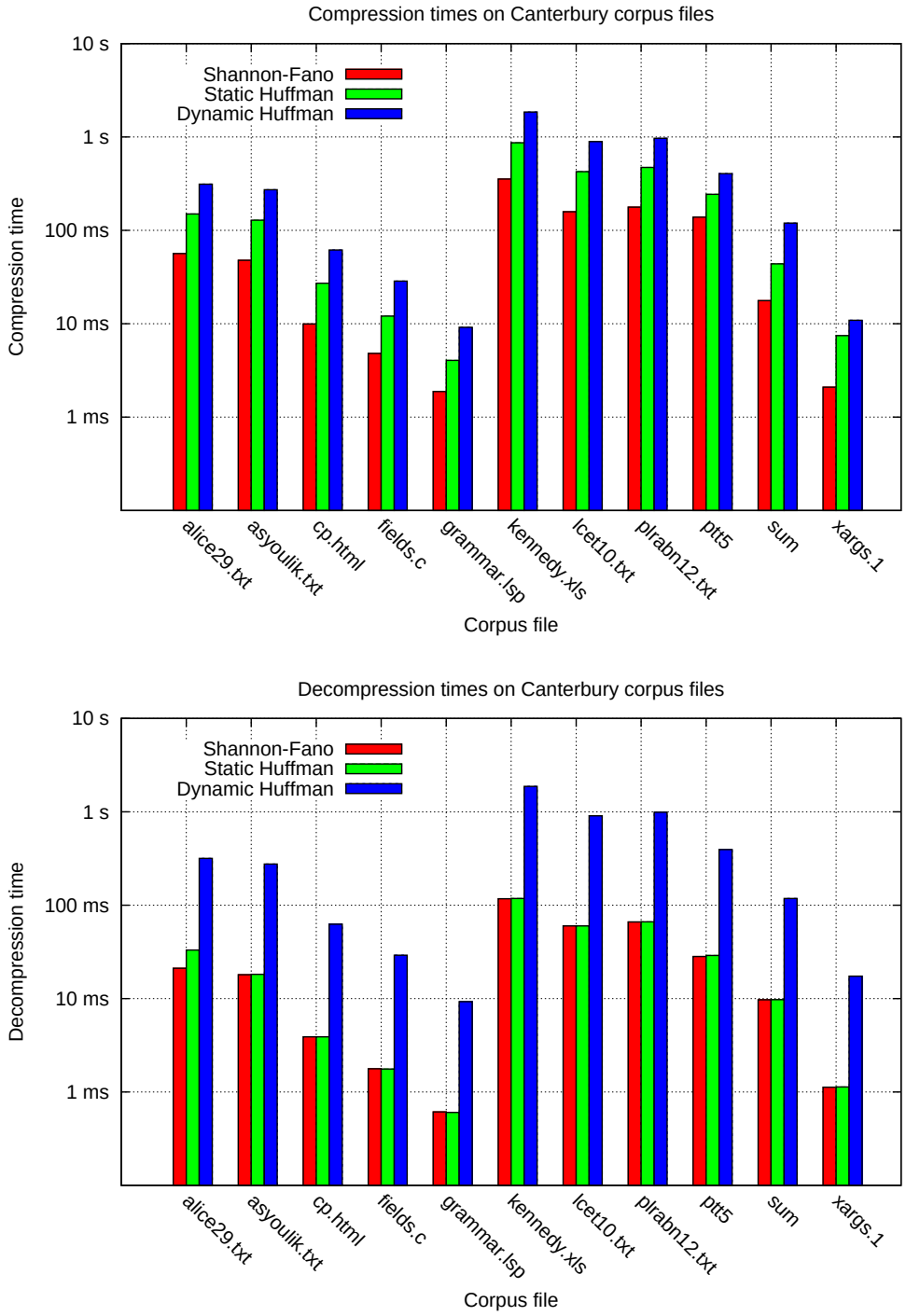



Figure 5.1: Compression and decompression times of the statistical methods

| | Compression time [$\mu s$] | | | | Decompression time [$\mu s$] | | | |
|---|---|---|---|---|---|---|---|---|
| File | LZ77 | LZSS | LZ78 | LZW | LZ77 | LZSS | LZ78 | LZW |
| alice29.txt | 221,572 | 350,888 | 7,516 | 7,107 | 14,724 | 26,223 | 3,253 | 5,446 |
| asyoulik.txt | 179,451 | 179,730 | 6,393 | 9,013 | 18,936 | 13,732 | 2,705 | 4,490 |
| cp.html | 17,816 | 38,672 | 1,915 | 1,739 | 3,769 | 2,449 | 559 | 943 |
| fields.c | 8,773 | 9,518 | 904 | 783 | 1,741 | 1,162 | 282 | 430 |
| grammar.lsp | 2,424 | 3,003 | 308 | 266 | 570 | 376 | 108 | 167 |
| kennedy.xls | 1,377,398 | 1,194,604 | 37,230 | 32,433 | 101,955 | 105,832 | 14,116 | 28,411 |
| lcet10.txt | 569,531 | 618,602 | 20,760 | 18,793 | 41,405 | 46,633 | 9,700 | 15,020 |
| plrabn12.txt | 819,687 | 782,825 | 23,836 | 22,023 | 46,287 | 53,871 | 16,305 | 16,926 |
| ptt5 | 578,781 | 6,173,581 | 10,482 | 10,036 | 51,382 | 53,939 | 10,423 | 10,906 |
| sum | 86,156 | 57,303 | 1,859 | 1,643 | 5,803 | 3,869 | 1,301 | 1,435 |
| xargs.1 | 4,260 | 3,082 | 255 | 326 | 647 | 426 | 192 | 199 |

Table 5.2: Compression and decompression times of the dictionary methods on the Canterbury files

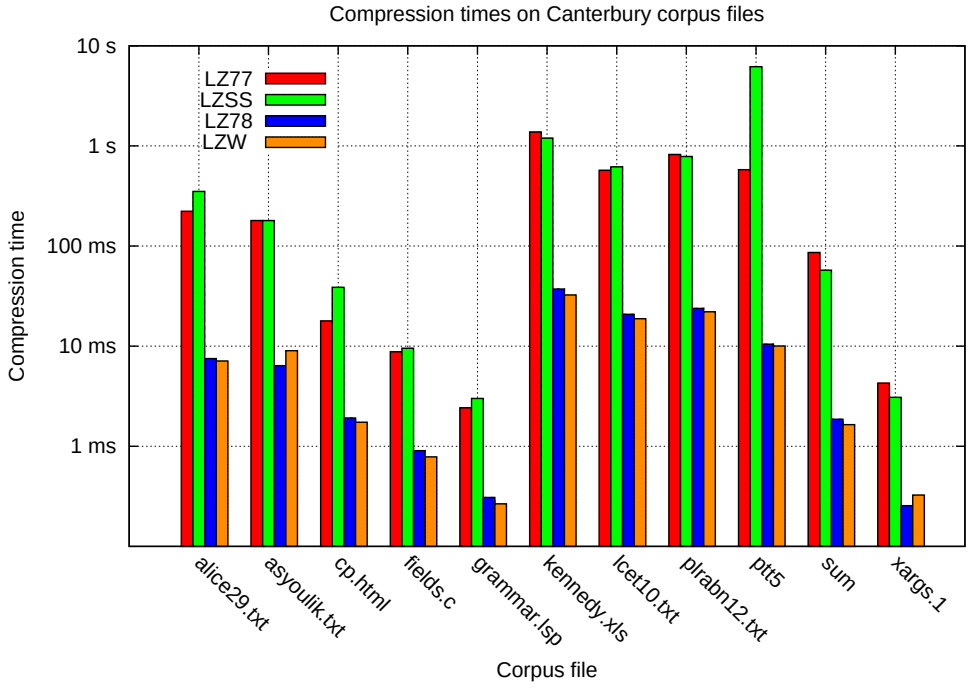



Figure 5.2: Compression times of the dictionary methods

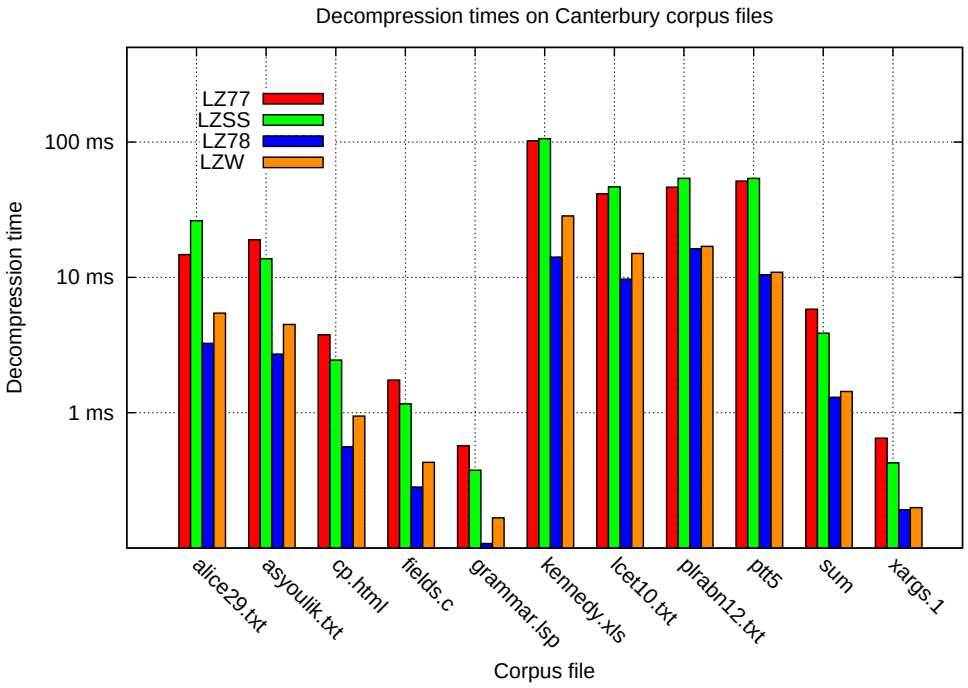



Figure 5.3: Decompression times of the dictionary methods

| File | SFano | SHuff | DHuff | LZ77 | LZSS | LZ78 | LZW |
|---|---|---|---|---|---|---|---|
| alice29.txt | 0.578937 | 0.577202 | 0.577346 | 0.560968 | 0.481849 | 0.613470 | 0.539309 |
| asyoulik.txt | 0.607786 | 0.606300 | 0.606476 | 0.595979 | 0.524505 | 0.644046 | 0.574569 |
| cp.html | 0.663862 | 0.662968 | 0.663496 | 0.518595 | 0.449376 | 0.605658 | 0.560663 |
| fields.c | 0.646009 | 0.640717 | 0.641704 | 0.428879 | 0.350942 | 0.578655 | 0.476502 |
| grammar.lsp | 0.611395 | 0.610051 | 0.609514 | 0.512765 | 0.416823 | 0.615157 | 0.568127 |
| kennedy.xls | 0.451497 | 0.449487 | 0.449699 | 0.274627 | 0.328858 | 0.241596 | 0.279121 |
| lcet10.txt | 0.588712 | 0.587397 | 0.587477 | 0.549471 | 0.469306 | 0.621806 | 0.544928 |
| plrabn12.txt | 0.572601 | 0.572138 | 0.572217 | 0.614835 | 0.545439 | 0.626928 | 0.560732 |
| ptt5 | 0.208228 | 0.208012 | 0.208027 | 0.277825 | 0.220414 | 0.148244 | 0.136831 |
| sum | 0.681485 | 0.679106 | 0.679916 | 0.557558 | 0.479001 | 0.586428 | 0.560617 |
| xargs.1 | 0.638987 | 0.638514 | 0.638987 | 0.608943 | 0.505559 | 0.694819 | 0.635912 |

Table 5.3: Compression ratios of all implemented methods on the Canterbury files

### 5.2.2 Prague Corpus

In this part of text we provide a discussion on the results from the Prague Corpus experiments and compare them with those from the Canterbury Corpus. Notice, that we present only some results because of large amount of data obtained during the testing. We refer to the rest of tables and appropriate charts stated in detail in Appendix E.

The Tables E.1 and E.2 contain the measured data, i.e., compression time, decompression time, and compression ratio, obtained from the testing of the statistical methods on the Prague Corpus file set. As in the case of the Canterbury files, the Shannon-Fano coding is the fastest one of these methods. However, its compression ratio is the worst among them, even if the values are very similar. Therefore, the compression ratio is not the deciding factor to evaluate

these algorithms. More interesting is the compression time of the Dynamic Huffman coding, which is significantly slow and thus not so efficient for the practical use. We assume this is caused due to the manipulation of the Huffman tree whenever a new symbol is encountered. This method should be thoroughly explored to overcome this undesirable drawback.

The comparison of the dictionary-based methods, as in the tables 5.4 and 5.5, gives us notably better results than the same comparison using the files from the Canterbury Corpus. The most remarkable is the *negative compression* in some cases, i.e., the compressed file occupies more space than its original. All statistical compression algorithms implemented within this work could always push down the file size even if only a little bit. The dictionary methods were not so successful. Hence, we can see that it is very important to include various file types in the corpus to detect such a negative feature.

| | Compression time [$\mu s$] | | | | Decompression time [$\mu s$] | | | |
|---|---|---|---|---|---|---|---|---|
| File | LZ77 | LZSS | LZ78 | LZW | LZ77 | LZSS | LZ78 | LZW |
| firewrks | 1,444,637 | 2,459,853 | 159,891 | 154,589 | 113,370 | 51,281 | 35,997 | 43,652 |
| thunder | 14,717,676 | 16,477,387 | 319,860 | 278,200 | 276,661 | 225,022 | 105,527 | 84,527 |
| drkonqi | 220,052 | 209,870 | 8,056 | 4,410 | 17,140 | 17,875 | 3,627 | 2,530 |
| libc06 | 97,242 | 88,620 | 2,587 | 3,811 | 4,865 | 4,815 | 977 | 1,191 |
| mirror | 100,998 | 195,467 | 4,584 | 5,937 | 13,885 | 14,141 | 3,053 | 3,374 |
| abbot | 198,826 | 280,020 | 43,045 | 22,683 | 42,688 | 13,812 | 8,689 | 16,583 |
| gtkprint | 91,728 | 222,101 | 1,555 | 1,411 | 3,699 | 3,860 | 1,212 | 844 |
| wnvcrdt | 434,599 | 2,586,361 | 10,498 | 6,088 | 33,231 | 35,286 | 6,780 | 4,434 |
| w01vett | 2,016,299 | 16,220,495 | 29,894 | 41,431 | 140,176 | 149,333 | 29,310 | 19,299 |
| emission | 1,326,482 | 6,462,507 | 82,383 | 71,495 | 247,370 | 262,791 | 67,426 | 71,842 |
| venus | 22,732,635 | 27,255,211 | 888,905 | 761,653 | 1,157,293 | 841,062 | 316,203 | 368,174 |
| nightsht | 14,922,267 | 23,802,595 | 1,146,905 | 1,007,606 | 1,172,412 | 598,307 | 371,783 | 456,131 |
| flower | 10,789,581 | 14,970,909 | 710,657 | 521,726 | 938,865 | 796,183 | 242,101 | 274,269 |
| corilis | 706,728 | 1,054,591 | 72,209 | 58,019 | 115,511 | 144,034 | 42,647 | 47,945 |

Table 5.4: The measured data of the dictionary methods on the files from the Prague Corpus (set A)

With regard to the compression times, it is similar as before. LZ78 and LZW are faster than two other dictionary techniques. Moreover, these methods are capable to achieve both the encoding and the decoding speed about 10 MB/s. Unfortunately, their compression ratios are not as good as the ratios of LZSS. However, we can increase the dictionary size. By default, the size 4,096 is used. The implementation allows to use the dictionary containing from 1 to 65,536 items. Section 5.3 is devoted to the experiments to find an optimal value so that the compression/decompression time would remain reasonable.

The following notion helps us to support our hypothesis, that the lack of different data types decreases the quality of a corpus. When we compare two files with a similar size of a different type, we can observe the distinct values regarding the compression time and the compression ratio. For example, notice the files `firewrks` and `w01vet`, whose sizes are about 1.5 MB. LZSS needs more than 16 seconds with pushing down the file `w01vet` to approximately 16% of its original size, while in the case of the file `firewrks` it is about 2 seconds to compress it with a negative compression ratio as a result. One may notice that the better results require more time to achieve such a compression. It may not necessary be the truth. To prove this, we propose the comparison of another two files, `thunder` and `hungary`, where the situation is opposite to the first example. This was not so obvious when using the files from the Canterbury files, hence the Prague Corpus seems to be preferable to be used for an evaluation of algorithms.

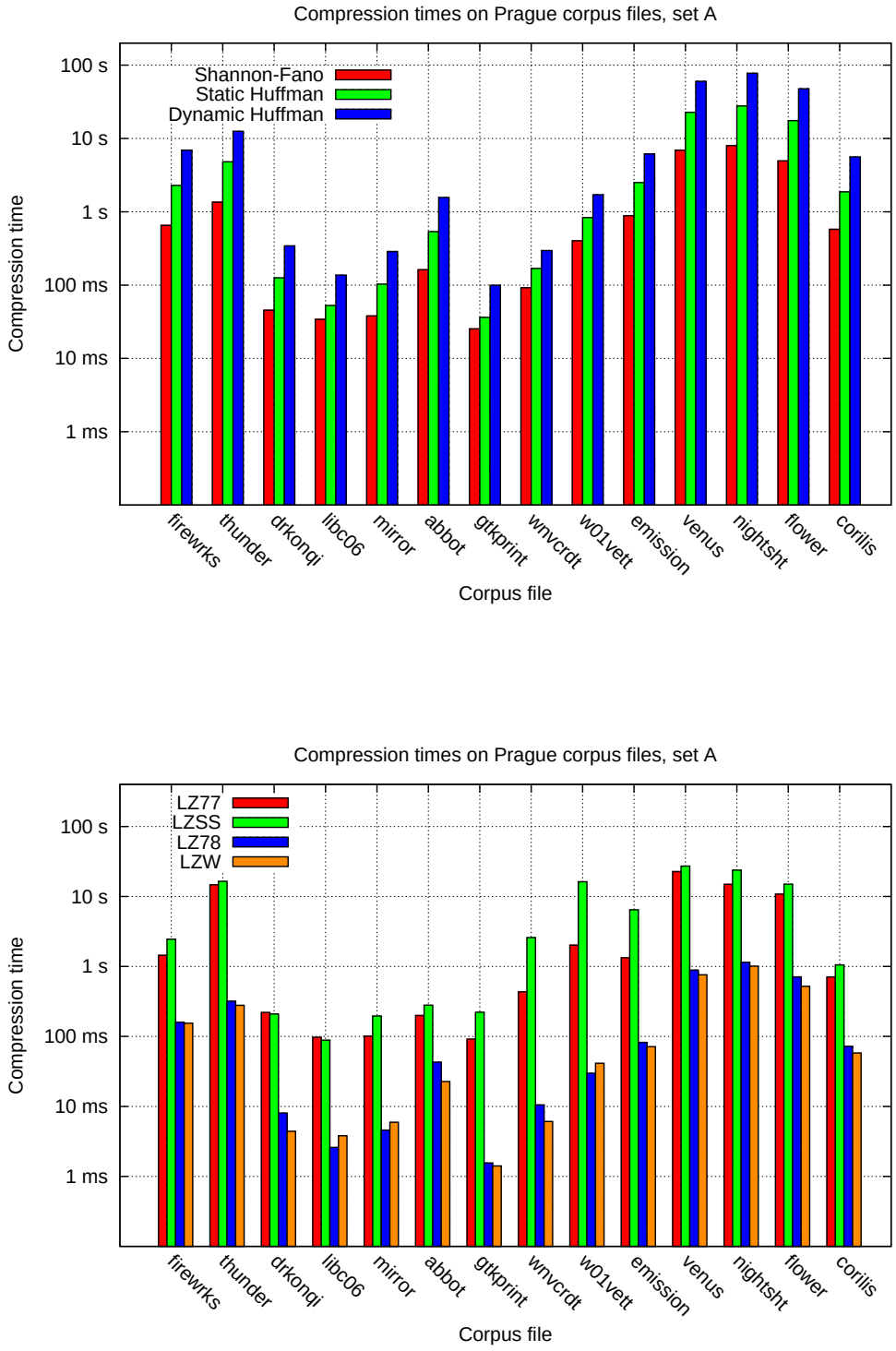



Figure 5.4: Compression times of all methods on the files from the Prague Corpus (set A)

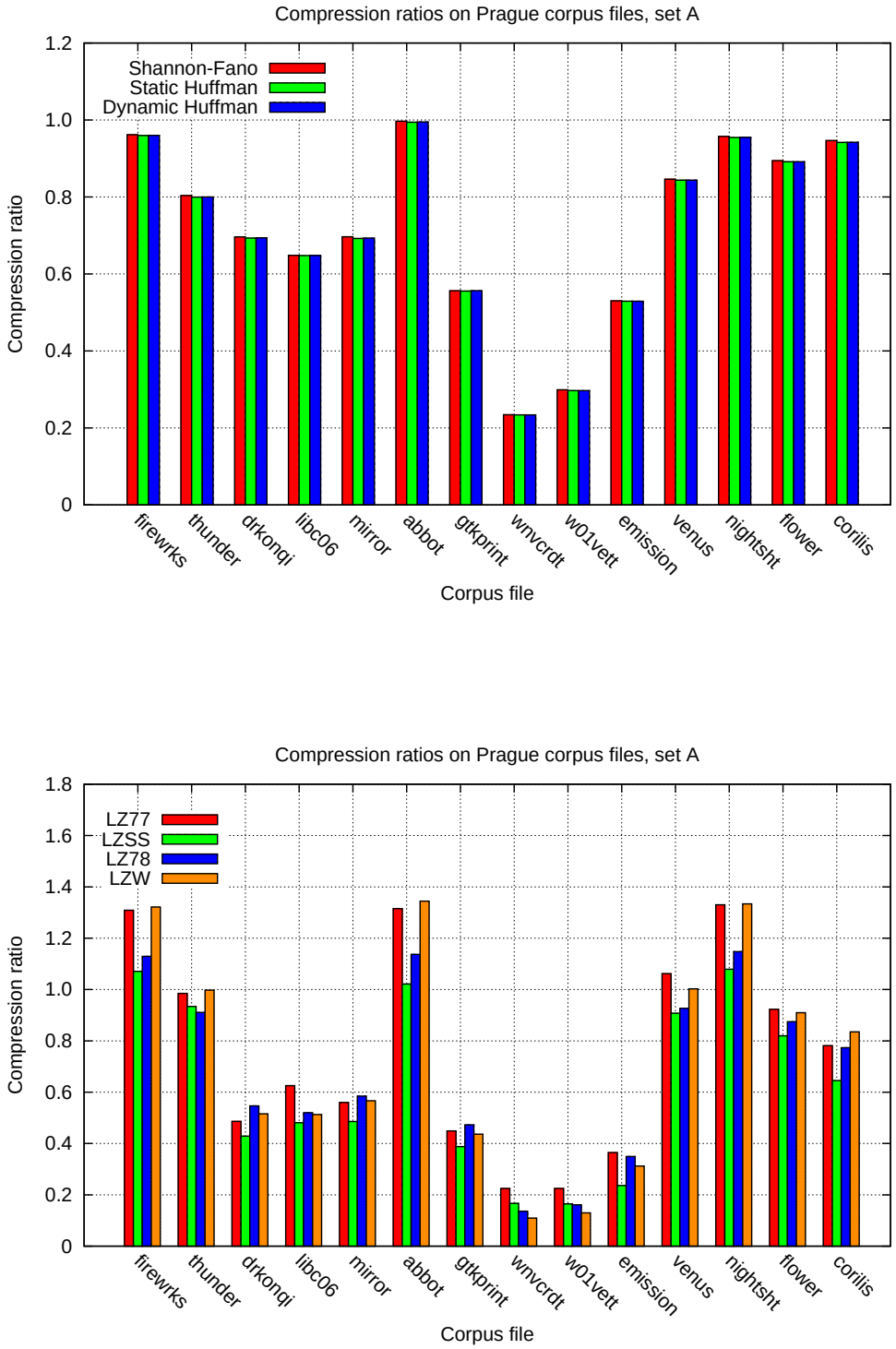



Figure 5.5: Compression ratios of the all implemented methods on the files from the Prague Corpus (set A)

| | Compression ratio | | | |
|---|---|---|---|---|
| File | LZ77 | LZSS | LZ78 | LZW |
| firewrks | 1.308778 | 1.070836 | 1.129115 | 1.321735 |
| thunder | 0.984881 | 0.933816 | 0.910682 | 0.997486 |
| drkonqi | 0.485890 | 0.428658 | 0.546841 | 0.515434 |
| libc06 | 0.625623 | 0.481027 | 0.519929 | 0.512884 |
| mirror | 0.559845 | 0.485522 | 0.584689 | 0.566672 |
| abbot | 1.315220 | 1.021644 | 1.136867 | 1.344467 |
| gtkprint | 0.449201 | 0.387913 | 0.473003 | 0.436209 |
| wnvcrdt | 0.224906 | 0.166888 | 0.136095 | 0.109280 |
| w01vett | 0.224504 | 0.164712 | 0.161192 | 0.129905 |
| emission | 0.364453 | 0.236672 | 0.349156 | 0.311961 |
| venus | 1.061799 | 0.907768 | 0.927089 | 1.002743 |
| nightsht | 1.329959 | 1.078796 | 1.147760 | 1.333555 |
| flower | 0.923105 | 0.819711 | 0.874569 | 0.909836 |
| corilis | 0.781170 | 0.645019 | 0.773533 | 0.834714 |

Table 5.5: The compression ratios of the dictionary methods on the files from the Prague Corpus (set A), the lowest values are highlighted

## 5.3 Parameter experiments

This purpose of this section is to determine a suitable size of the Search buffer for LZSS compression, and to find the optimal size of the dictionary in the case of LZ78 or LZW. The file `bovary` has been chosen to be tested. The compression times and ratios were monitored, the results for LZSS method are in Table 5.6.

| SB size | Time [$\mu s$] | Ratio | SB size | Time [$\mu s$] | Ratio |
|---|---|---|---|---|---|
| 32 | 704,708 | 0.779058 | 1,024 | 1,193,962 | 0.489636 |
| 64 | 731,964 | 0.696117 | 2,048 | 1,846,309 | 0.467970 |
| 128 | 776,461 | 0.637712 | 4,096 | 2,816,608 | 0.450844 |
| 256 | 838,815 | 0.561457 | 8,192 | 4,698,748 | 0.426162 |
| 512 | 963,315 | 0.516395 | 16,384 | 7,305,987 | 0.401667 |

Table 5.6: The compression times and ratios depending on the LZSS Search buffer size

As we can see, the compression ratio improves with the size of the Search buffer. When it contains up to 65,536 characters, the compression ratio is approximately 2× better than lowest tested value, i.e., 32. Considering the larger values of obtained times, we assume that the function has an exponential growth. We decided to set the value 4,096 as a default size of the Search buffer for LZSS and LZ77. Altough, the higher values achieve better results, the compression times grows fastly up and we want to keep it reasonable.

Next, we tested the LZ78 method and the results, listed in Table 5.7, are relatively remarkable. The compression times remained allmost the same during the whole testing. Moreover, the values were getting lower as the size of the dictionary increased. This can be explained due to the frequent initialization of the dictionary, because it fills up quickly with a small capacity. Therefore, the larger values of the dictionary can be set to get very fast superior results.

The appropriate charts for both teste methods are in figures 5.7 and 5.6. Notice, that the

vertical axes for compression times are in logarithm scale. To outline the course of a function, the appropriate curves were added to the graphs. We do not present a graph for the compression times of LZW method, since the values are almost constant.

| SB size | Time [$\mu s$] | Ratio | SB size | Time [$\mu s$] | Ratio |
|---|---|---|---|---|---|
| 32 | 140,359 | 0.986584 | 1,024 | 102,993 | 0.621857 |
| 64 | 129,385 | 0.881677 | 2,048 | 102,990 | 0.585530 |
| 128 | 118,564 | 0.791346 | 4,096 | 101,957 | 0.550717 |
| 256 | 110,592 | 0.720025 | 8,192 | 101,714 | 0.518583 |
| 512 | 140,001 | 0.665956 | 16,384 | 103,558 | 0.489578 |

Table 5.7: The compression times and ratios depending on the LZ78 dictionary size

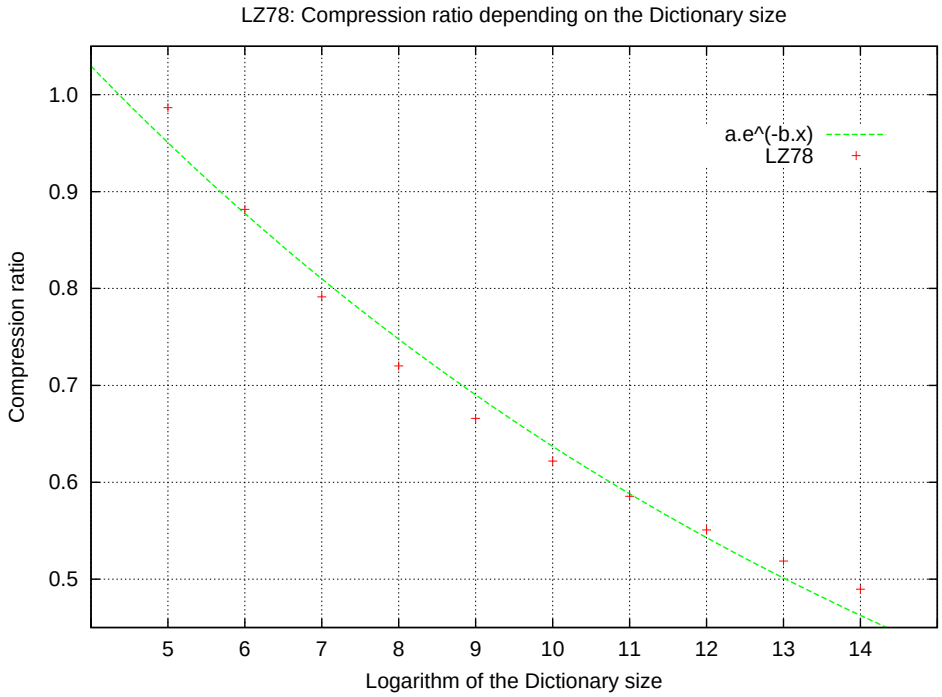



Figure 5.6: Compression ratio of LZ78 depeneding on the dictionary size

## 5.4 Existing methods

This section provides the discussed experiment results for the context methods, which were already implemented within the master thesis by Filip Šimek, [3]. The algoritms were tested on the whole Prague Corpus, however, we were not able to complete the compression of two largest files, `venus` and `nightsht`, using the DCA method. The computer notably slowed down whenever we tried to execute the encoding with a tendency to freeze. Therefore, we gave up to compress them after several attempts. The measured data for the set A are presented in Table 5.8 and in Table 5.9, respectively.

ACB algorithm was significantly the slowest in all performed cases. This drawback was already explained by Filip Šimek as an ineffective implementation of used algorithms. Based on the results from [3] we decided to set the Search buffer size of ACB to 4,096 to avoid the

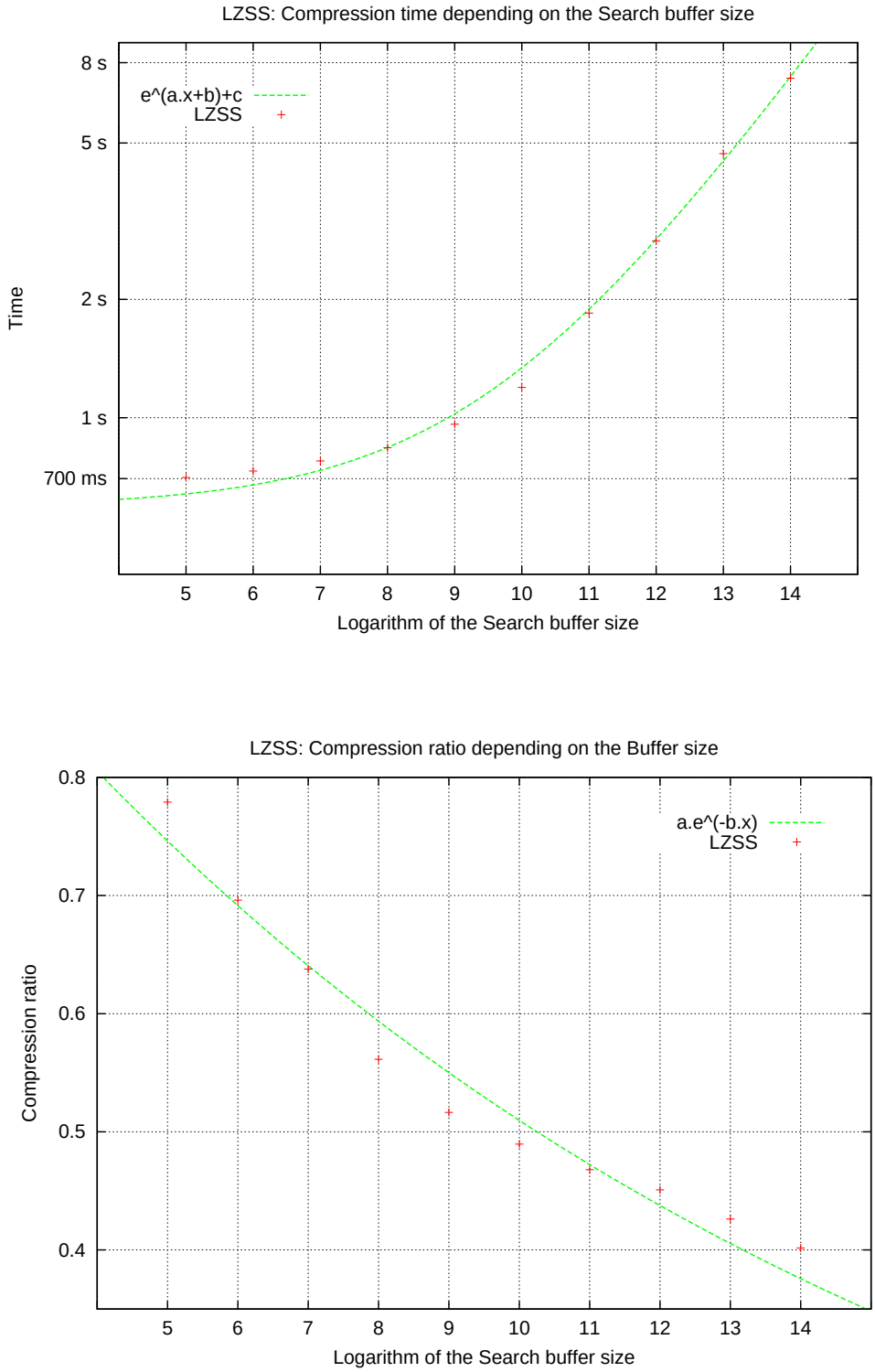



Figure 5.7: Compression time and ratio depending on the logarithm of the LZSS Search buffer size

| | Compression time [$\mu s$] | | | Decompression time [$\mu s$] | | |
|---|---|---|---|---|---|---|
| File | ACB | DCA | PPM | ACB | DCA | PPM |
| bovary | 86,688,206 | 1,206,903 | 322,380 | 78,295,563 | 1,198,491 | 336,170 |
| modern | 16,240,293 | 268,837 | 101,548 | 13,784,635 | 278,612 | 111,307 |
| ultima | 41,314,370 | 2,536,075 | 510,722 | 33,601,722 | 2,707,698 | 597,208 |
| lusiadas | 87,709,386 | 475,977 | 117,535 | 70,172,690 | 470,079 | 131,385 |
| cyprus | 18,353,726 | 141,976 | 21,698 | 15,964,467 | 111,761 | 22,303 |
| hungary | 127,497,124 | 800,071 | 130,146 | 112,613,360 | 1,727,312 | 132,936 |
| compress | 4,503,601 | 81,745 | 19,258 | 4,353,701 | 82,026 | 13,131 |
| lzfindmt | 870,163 | 20,496 | 4,461 | 853,287 | 25,615 | 4,865 |
| render | 580,546 | 18,098 | 3,545 | 557,137 | 22,486 | 3,864 |
| handler | 521,844 | 14,041 | 2,686 | 513,276 | 18,600 | 2,952 |
| usstate | 372,838 | 11,154 | 1,954 | 357,706 | 14,245 | 2203 |
| collapse | 78,075 | 8,911 | 1,065 | 75,802 | 8,711 | 1,191 |
| xmlevent | 331,157 | 15,056 | 1,992 | 328,439 | 14,361 | 2,211 |
| mailflder | 1,670,053 | 49,786 | 8,242 | 1,591,313 | 33,097 | 8,951 |
| age | 44,005,647 | 309,690 | 79,751 | 33,637,556 | 168,983 | 86,947 |
| higrowth | 4,964,372 | 247,383 | 57,404 | 4,520,433 | 210,545 | 40,766 |

Table 5.8: The measured data of the context methods on the files from the Prague Corpus (set B)

negative compression. However, this situation occurred in one case when compressing the file `ultima`.

In contrast to ACB, PPM method performed very well in all cases, while it is tens of times faster than ACB and approximately 4× faster than DCA. If we compare the compression ratios of DCA as a sum of the compressed file and its corresponding file with exceptions to the antidictionary with the ratios of PPM, the latter one is always better. Nevertheless, the values of DCA compression ratios can be improved since the coding used for the exceptions is poor.

## 5.5 Summary

This chapter was devoted to en evaluation of all compression methods, which are currently implemented in the ExCom library. Furthermore, the second aim was to verify the effectiveness and the real usage of the developed Prague Corpus. Some significant findings were discovered, such as that the files from the Canterbury Corpus are too small and including a few data types only. Presently, many new formats exist and the sizes of files increased a lot. Some algorithms, whose results from the Canterbury Corpus were good, failed in several cases with regard to their encoding speed or the negative compression.

To conclude the implemented methods, PPM gives the best results on all files from the Prague Corpus, the second one is LZSS. The encoding speeds of both algorithms are very good. On the other hand, the perfomance of ACB and DCA is not satisfactory. Furthermore, the Dynamic Huffman coding should also be improved to achieve lower compression and decompression times.

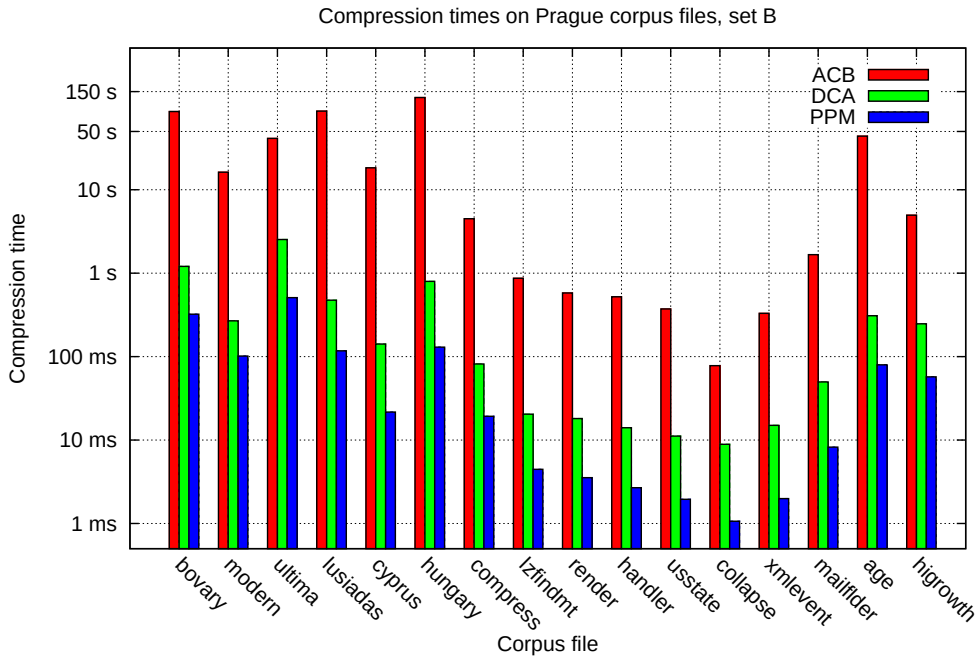



Figure 5.8: Compression times of the context methods on the files from the Prague Corpus (set B)

| | Compression ratio | | | |
|---|---|---|---|---|
| File | ACB | DCA | DCA+*exc* | PPM |
| bovary | 0.703750 | 0.319436 | 0.633733 | 0.223464 |
| modern | 0.883291 | 0.355708 | 0.653094 | 0.276376 |
| ultima | 1.250290 | 0.208723 | 3.519402 | 0.599823 |
| lusiadas | 0.624126 | 0.471518 | 1.099133 | 0.256622 |
| cyprus | 0.234954 | 0.107497 | 0.144473 | 0.046812 |
| hungary | 0.247117 | 0.124309 | 0.133957 | 0.042301 |
| compress | 0.483009 | 0.140910 | 0.585941 | 0.157641 |
| lzfindmt | 0.480019 | 0.115173 | 0.569409 | 0.176294 |
| render | 0.508258 | 0.141329 | 0.775838 | 0.209647 |
| handler | 0.487072 | 0.134760 | 0.881833 | 0.207277 |
| usstate | 0.453036 | 0.154527 | 0.956369 | 0.221791 |
| collapse | 0.679554 | 0.087774 | 1.492860 | 0.340648 |
| xmlevent | 0.538451 | 0.094802 | 1.000928 | 0.237868 |
| mailfldr | 0.501852 | 0.234634 | 0.693840 | 0.180051 |
| age | 0.755991 | 0.395063 | 1.671154 | 0.369899 |
| higrowth | 0.715963 | 0.321802 | 1.479704 | 0.343132 |

Table 5.9: The compression ratios of the context methods on the files from the Prague Corpus (set B)

# Chapter 6

# Conclusion

This work deals with a description of lossless compression algorithms. After a brief introduction to data compression terminology, the representatives of the statistical and dictionary-based methods were presented. Each of the methods was thoroughly researched to discover all their features. To emphasise them, the methods were described together with some illustrative examples.

Based on the research of the compression schemes, the seven following methods were implemented and consequently adapted for the universal library of compression algorithms called ExCom. Shannon-Fano coding, Static Huffman coding and its Adaptive version were created as the typical algorithms of the statistical techniques. For the purpose to provide the representatives for dictionary methods, we decided to choose LZ78, LZW, LZ77 and its derivate LZSS. The significant implementation details were interpreted.

Nevertheless, the main goal of this thesis was to develop a new corpus to objectively compare the lossless compression schemes. After a research of existing corpora, a new set of files, named Prague Corpus, was created. Along with the real example of how the files were chosen, a design of the methodology to maintain its timeliness was presented. To prove a correctness of the Corpus, thorough experiments were performed using both the already existing context methods (ACB, DCA, PPM) and the methods implemented within this work.

The experiments and measurements discovered some remarkable results which verified the effectiveness of the Prague Corpus. On the other hand, one of its predecessors, the Canterbury Corpus, was considered to be too old for the further research. However, it is still widely used, and a distribution of the Prague Corpus would not be effortless. Moreover, LZ78 and LZW methods seemed to perform very fast in all cases of testing.

## 6.1 Future work

Altough all the methods were implemented with regard to be efficient, the experiments showed some drawback of the Dynamic Huffman coding. Therefore, this particular implementation of this algorithm should be inspected to find its bottlenecks. We suppose, that it is caused due to an ineffective manipulation with the binary tree. Moreover, we implemented the FGK version, thus the Vitter's approach should be explored, too.

To complete the list of statistical methods, the Range encoding might be implemented as a free alternative to its predecessor, the Arithmetic coding, which is a patent encumbered technique.

During the research we found the LZMA method designed by Igor Pavlov to be very efficient. The source codes are freely available in many programming languages, hence it can be integrated into the ExCom library.

The files for the Prague Corpus were determined after the mainly manually-operated proceedings. In the future, this should be eliminated. Therefore, we suggest to create a standalone tool or the one integrated within the ExCom library. Such a tool would be able to select the appropriate candidates based only on a given list of file paths and some parameters. This includes the compression with several compression programs or algorithms, further processing of measured data and their interpretation.

# References


[1] C. E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal*, vol. 27, pp. 379–423, 625–56, 1948.

[2] C. E. Shannon and W. Weaver, *The Mathematical Theory of Communication*. Urbana and Chicago: University of Illinois Press, 1949.

[3] F. Šimek, "Data compression library," Master's thesis, Czech Technical University in Prague, May 2009.

[4] D. Salomon, *Data Compression: The Complete Reference*. Springer-Verlag, fourth ed., 2007.

[5] M. Simpson, R. Barua, and S. Biswas, "Analysis of Compression Algorithms for Program Data," *Tech. rep., U. of Maryland, ECE department. August*, Aug. 2003.

[6] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the Institute of Radio Engineers*, vol. 40, pp. 1098–1101, Sept. 1952.

[7] N. Faller, "An adaptive system for data compression," in *In Record of the 7th .4silomar Conference on Circuits, Systems, and Computers*, pp. 593–597, 1973.

[8] R. Gallager, "Variations on a theme by Huffman," *Information Theory, IEEE Transactions on*, vol. 24, pp. 668–674, Nov. 1978.

[9] D. E. Knuth, "Dynamic Huffman coding," *J. Algorithms*, vol. 6, no. 2, pp. 163–180, 1985.

[10] J. S. Vitter, "Design and analysis of dynamic Huffman codes," *J. ACM*, vol. 34, no. 4, pp. 825–845, 1987.

[11] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Commun. ACM*, vol. 30, no. 6, pp. 520–540, 1987.

[12] A. Bookstein, S. T. Klein, and T. Raita, "Is Huffman coding dead?," in *SIGIR '93: Proceedings of the 16th annual international ACM SIGIR conference on Research and development in information retrieval*, (New York, NY, USA), pp. 80–87, ACM, 1993.

[13] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, pp. 337–343, 1977.

[14] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[15] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Research Report 124, Digital SRC, Palo Alto, CA, USA, May 1994.

[16] G. Buyanovsky, "Associative Coding," *Monitor*, pp. 10–22, 1994.

[17] M. Crochemore, F. Mignosi, A. Restivo, and S. Salemi, "Data Compression Using Antidictionaries," in *in Proceedings of the IEEE, Lossless Data Compression*, pp. 1756–1768, 2000.

[18] J. Cleary and I. Witten, "Data Compression Using Adaptive Coding and Partial String Matching," *IEEE Transactions on Communications*, vol. 32, pp. 396–402, Apr. 1984.

[19] A. Moffat, "Implementing the PPM data compression scheme," *IEEE Trans. Comms.*, vol. 38, pp. 1917–1921, Nov. 1990.

[20] M. Fiala, "Implementation of DCA Compression Method," Master's thesis, Czech Technical University in Prague, May 2007.

[21] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text compression*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990.

[22] J. A. Storer and T. G. Szymanski, "Data compression via textual substitution," *J. ACM*, vol. 29, no. 4, pp. 928–951, 1982.

[23] T. Bell and D. Kulp, "Longest-match string searching for Ziv-Lempel compression," *Softw. Pract. Exper.*, vol. 23, no. 7, pp. 757–771, 1993.

[24] K. Sadakane and H. Imai, "Improving the Speed of LZ77 Compression by Hashing and Suffix Sorting," *IEICE transactions on fundamentals of electronics, communications and computer sciences*, vol. 83, no. 12, pp. 2689–2698, 2000-12-25.

[25] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *J. ACM*, vol. 32, no. 3, pp. 652–686, 1985.

[26] T. A. Welch, "A Technique for High-Performance Data Compression," *Computer*, vol. 17, no. 6, pp. 8–19, 1984.

[27] D. Phillips, "LZW data compression," pp. 36–48, 1992.

[28] P. Elias, "Universal codeword sets and representations of the integers," *Information Theory, IEEE Transactions on*, vol. 21, no. 2, pp. 194–203, 1975.

[29] D. Salomon, *Variable-length Codes for Data Compression*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[30] J. Nieminen, "An efficient LZW implementation." http://warp.povusers.org/EfficientLZW/, May 2010.

[31] T. Bell, I. H. Witten, and J. G. Cleary, "Modeling for text compression," *ACM Comput. Surv.*, vol. 21, no. 4, pp. 557–591, 1989.

[32] R. Arnold and T. Bell, "A Corpus for the Evaluation of Lossless Compression Algorithms," in *DCC '97: Proceedings of the Conference on Data Compression*, (Washington, DC, USA), p. 201, IEEE Computer Society, 1997.

[33] S. Deorowicz, *Universal Lossless Data Compression Algorithms*. PhD thesis, Silesian University of Technology, 2003.

[34] C. G. Nevill-Manning and I. H. Witten, "Protein Is Incompressible," in *DCC '99: Proceedings of the Conference on Data Compression*, (Washington, DC, USA), p. 257, IEEE Computer Society, 1999.

[35] R. Islam and S. Rajon, "On the design of an effective corpus for evaluation of Bengali Text Compression Schemes," in *Computer and Information Technology, 2008. ICCIT 2008. 11th International Conference on*, pp. 236 –241, 24-27 2008.

# Appendix A

# The Corpus methodology

The real example of how to proceed to create a new corpus was presented in Chapter 3. However, some steps may be slightly modified or completely removed in the future. Therefore, the following text provides the generalized method of how to act if the need to update the Prague Corpus arises.

- The file type groups should be checked first, i.e., all corresponding files should be inspected whether they still belong to the appropriate data set. Whenever a new file type is considered to be frequently used in practice, it should be included for further testing. In other words, if, for example, a new binary format becomes widely spread, then it should be placed into the relevant category. There are several ways of how to find out such files.

  - URL: http://en.wikipedia.org/wiki/List_of_file_formats

    The free encyclopedia *Wikipedia* provides a comprehensive list of file formats segmented into a large number of groups. However, some groups can be joined if the representatives share some features in common. As an example we can bring in the *Documents* and the *Presentation* types, because the `PPT` or `ODT` files, as examples of presentation, are typically converted into the `PDF` format, which belongs to documents.
  - The web search engines can be used to get a list of typically used file types. The search phrase or keywords can be *most common file extensions*, *common file formats*, *list of file types*, or a combination of them.
  - The statistics of IT security companies, especially of those which focus on antivirus softwares, might be a relatively good criterion to decide what file formats are commonly used. The attackers change the file extensions to mystify a potential victim. Therefore, the companies sometimes provide the list of files which are abused.

  Some other tips and resources which were used in this work are stated in Appendix D.

- All data must come from the real sources used in practice. Randomly generated data do not provide the authentic results during the experiments.

- The files intended to be tested ought to be placed in the *public domain*, since the purpose of the Corpus is to be available for anyone and freely redistributable. A potential author of the Corpus should collect as many sources of file candidates as possible to avoid a *similarity* of data samples. This means that, for example, `XML` documents obtained from one source may have the same structure differing only in the stored information. Moreover, it is suitable for the test data set to contain files of various sizes, from few bytes to several

megabytes. However, the distribution of sizes should be equable. This condition will be described shortly. Especially the documents should not contain only the English texts. It is preferable to include files written in languages from different language families.

- Some formats, such as `PDF` and OpenOffice files, are internaly compressed. Thus, it is difficult to compress them again. If possible, they should be decoded before using them in the experiments. In the case of `PDF` files, for example, a handy tool `pdftk`[1] can be used to uncompress the page streams.

- The terms of usage, licenses and other copyright matters must be checked before using the files planned to be examined.

- It is recommended to maintain a list of all obtained files with the corresponding resources assigned. Because at the end of experiments it is often difficult to retrieve the source again. This approach is also suitable in the sense of backward getting of information of used files.

- When all the candidates are collected, they should be divided into appropriate groups. Furthemore, the subgroups based on the file extensions might be created.

- At this point, the measurements can be performed. All files should be compressed with several (at least three) compression programs using the different algorithms. The following website contains the list of tens of compression programs with the used algorithms stated:

  `http://www.maximumcompression.com/programs.php`

  The compression ratio, as the main factor, is monitored. If more sophisticated techniques to choose the files are demanded, the size of the input alphabet can be involved for finding a decision. This can be suitable for the plain text files written in various languages.

- The obtained compression ratios are then processed to create a scatter plot with the linear regression line fitted to it. Such a graph should be generated to every subgroup containing at least 15–20 files in each data type category. Moreover, to take the advantage of all files, including those which did not satisfy the condition of the minimal number of candidates, the same plot can be constructed to gain a file representing the whole category. If the distribution of file sizes would be unequal, i.e., a plenty of smaller files and only a few of large ones is used or *vice versa*, some files may not be fully processed and the chance to establish them as acceptable would be small.

- Then, the file closest to the regression line (the square of the distance below or above the straight line) in each graph is established.

- The whole process of creating the graphs and choosing the files is repeated for all the compression tools used. Hence, as many files of the corresponding data type is obtained as the number of used compression programs is. However, only one file per subgroup and the whole category is used. To decide which ones should be chosen, their compression ratios are compared with regard to the lowest value.

- Currently, there is no complex tool providing the execution of all experiment steps. However, such a tool is planned to be created to simplify the whole process.

[1] `http://www.pdftk.com/`, May 2010.

- The existing files can be removed completely and replaced with the new ones. The making of the decision is left to the contributor of the Corpus. Nevertheless, the old version of the Corpus should be kept for some purposes. Every Corpus version should be documented using a report, whose recommended (but not mandatory) template is proposed in Appendix B. Any source must be fully acknowledged and their file licenses should be included together with the distribution of the Corpus.
- This way established Corpus is finally exposed on the Internet, preferably on the website dedicated to the Corpus.

# Appendix B

# The Corpus report

This appendix presents a recommended, but not mandatory, template of the report which should be released together with a new version of the Prague Corpus.

---

**Author:** .....
**Date:** .....
**Version:** .....

**Total number of files:** .....
**Number of removed files:** .....
**Revision of the file pool:** ☐
**Revision of the public domain sources:** ☐

**Abstract:** .....
.....
.....

**List of new corpus files:**

**Name:** **Original name:**
**Type:** **Size:**
**Source:**
**License:**
**Description:**
.
.

**Summary:** .....
.....
.....

---

The *version* should be in format `YYYY.MM.N`, where `YYYY` and `MM` is the year and month, respectively. `N` is the counter used in the case that more than one version of corpus is developed within one month. This can occur when the further testing detects some drawbacks.

We denote by *Number of removed files* a number of files deleted from the previous version of the corpus. Hence, it should be in a $k/n$ format, where $k$ is the number of removed files and $n$ a total number of files. The *file pool* is a collection or a database of all files which were used for testing. In other words, it contains also those files which were not used during some previous experiments. By *revision* we mean an update of such files. The *public domain sources* should also be maintained to keep only those resources where the suitable data can be found.

A short abstract should contain the main reasons why an update of the corpus was necessary. The list of chosen files follows in the given format, as it is in Appendix C. The whole process should be concluded, i.e., the significant parts or remarkable results might be mentioned.

# Appendix C

# Prague corpus files

This appendix provides a comprehensive list of all Prague Corpus files together with their description. We acknowledge all the sources, pages and projects to use their data for the research purposes.

**Name:** firewrks **Original name:** 27647__hanstimm__FS_fw2.f10.aiff
**Type:** Audio **Size:** 1,440,054 B
**Source:** http://www.freesound.org/samplesViewSingle.php?id=27647
**License:** Creative Commons Sampling Plus 1.0 License
**Description:** Sound of fireworks

**Name:** thunder **Original name:** 24003__Erdie__mega_thunder.wav
**Type:** Audio **Size:** 3,172,048 B
**Source:** http://www.freesound.org/samplesViewSingle.php?id=24003
**License:** Creative Commons Sampling Plus 1.0 License
**Description:** Sound of thunder

**Name:** drkonqi **Original name:** drkonqi
**Type:** Binary **Size:** 111,056 B
**Source:** The Debian package `kdebase-bin` containing core binaries for the KDE base module
**License:** GNU General Public License
**Description:** KDE crash handler gives the user feedback if a program crashed

**Name:** libc06 **Original name:** libc06.dll
**Type:** Binary **Size:** 48,120 B
**Source:** http://sourceforge.net/projects/tuxpaint/files/
**License:** GNU Lesser General Public License
**Description:** A dynamic library from a drawing program Tux Paint

**Name:** mirror **Original name:** mirror
**Type:** Binary **Size:** 90,968 B
**Source:** The Debian package `apt-mirror`
**License:** GPLv2+
**Description:** A tool providing ability to mirror any parts of Debian GNU/Linux distribution or any other apt sources

**Name:** abbot **Original name:** abbot.jar
**Type:** Binary **Size:** 349,055 B
**Source:** http://www.sweethome3d.eu/
**License:** GNU General Public License
**Description:** Part of a free interior design application Sweet Home 3D

**Name:** gtkprint **Original name:** gtkunixprint.so
**Type:** Binary **Size:** 37,560 B
**Source:** http://www.pygtk.org/
**License:** GNU Lesser General Public License
**Description:** A shared object for PyGTK application whose purpose is to create programs with a GUI using the Python programming language

**Name:** wnvcrdt **Original name:** wnvcrdt.dbf
**Type:** Database **Size:** 328,550 B
**Source:** http://nationalatlas.gov/mld/wnvavit.html
**License:** Public domain
**Description:** A database file containg information about West Nile Virus Surveillance (Wild Bird cases) from 2000

**Name:** w01vett **Original name:** w01vett.dbf
**Type:** Database **Size:** 1,381,141 B
**Source:** http://nature.berkeley.edu/~bingxu/DataSources.htm
**License:** Public domain
**Description:** A database file containg information about West Nile Virus Surveillance (Veterinary cases) from 2001

**Name:** emissions **Original name:** Waterbase_Emissions_v1.mdb
**Type:** Database **Size:** 2,498,560 B
**Source:** http://www.eea.europa.eu/data-and-maps/
**License:** Public domain
**Description:** This database contains data on emissions of nutrients and hazardous substances to water, aggregated within River Basin Districts (RBDs), in the EEA member countries

**Name:** bovary **Original name:** 15711-pdf.pdf
**Type:** Document **Size:** 2,202,291 B
**Source:** http://www.gutenberg.org/etext/15711
**License:** The Project Gutenberg License
**Description:** Gustave Flaubert's first published novel Madame Bovary written in German

**Name:** modern **Original name:** 15703-8.txt
**Type:** Document **Size:** 388,909 B
**Source:** http://www.gutenberg.org/etext/15703
**License:** The Project Gutenberg License
**Description:** A book Modern by by Ernst Ahlgren and Axel Lundegård written in Swedish (ISO-8859-1 encoding)

**Name:** ultima **Original name:** 30334-pdf.pdf
**Type:** Document **Size:** 1,073,079 B
**Source:** http://www.gutenberg.org/etext/30334
**License:** The Project Gutenberg License
**Description:** A Science-Fiction book by Mack Reynolds written in English

**Name:** lusiadas **Original name:** 3333-doc.doc
**Type:** Document **Size:** 625,664 B
**Source:** http://www.gutenberg.org/etext/3333
**License:** The Project Gutenberg License
**Description:** A Portuguese epic poem by Luís Vaz de Camões

**Name:** venus **Original name:** pvo_uv_790226.tiff
**Type:** Graphics **Size:** 13,432,142 B
**Source:** http://nssdc.gsfc.nasa.gov/photo_gallery/photogallery-venus.html
**License:** Public domain
**Description:** Ultraviolet image of Venus' clouds as seen by the Pioneer Venus Orbiter (Feb. 26, 1979), the resolution is 2048 × 2188

**Name:** nightsht **Original name:** nightshot_iso_100.pgm
**Type:** Graphics **Size:** 14,751,763 B
**Source:** http://www.imagecompression.info/
**License:** Public domain
**Description:** A photo of a city at night, the resolution is 3136 × 2352

**Name:** flower **Original name:** flower_foveon.ppm
**Type:** Graphics **Size:** 10,287,665 B
**Source:** http://www.imagecompression.info/
**License:** Public domain
**Description:** A photo of a flower, the resolution is 2268 × 1512

**Name:** corilis **Original name:** corilis06_r5l1_c3b.tif
**Type:** Graphics **Size:** 1,262,483 B
**Source:** http://www.eea.europa.eu/data-and-maps/
**License:** Public domain
**Description:** CORILIS land cover data, from CORIne and LISsage (smoothing in French)—a methodology developed for land cover data generalization and analysis

**Name:** cyprus **Original name:** CY_meta.xml
**Type:** Markup languages **Size:** 555,986 B
**Source:** http://www.eea.europa.eu/data-and-maps/
**License:** Public domain
**Description:** Data from AirBase (The European air quality database) for Cyprus

**Name:** hungary **Original name:** HU_meta.xml
**Type:** Markup languages **Size:** 3,705,107 B
**Source:** http://www.eea.europa.eu/data-and-maps/
**License:** Public domain

**Description:** Data from AirBase (The European air quality database) for Hungary

**Name:** compress **Original name:** compress.html
**Type:** Markup languages **Size:** 111,646 B
**Source:** http://en.wikipedia.org/wiki/Data_compression
**License:** Creative Commons Attribution-ShareAlike License
**Description:** A Wikipedia page containing information about data compression

**Name:** handler **Original name:** CommandPacketHandler.java
**Type:** Scripts **Size:** 11,873 B
**Source:** http://opengts.sourceforge.net/
**License:** Apache Software License, version 2
**Description:** Java source code from the OpenGTS project

**Name:** usstate **Original name:** USState.java
**Type:** Scripts **Size:** 8,251 B
**Source:** http://opengts.sourceforge.net/
**License:** Apache Software License, version 2
**Description:** Java source code from the OpenGTS project

**Name:** lzfindmt **Original name:** LzFindMt.c
**Type:** Scripts **Size:** 22,922 B
**Source:** http://www.7-zip.org
**License:** GNU Lesser General Public License
**Description:** C source code from the 7-zip project

**Name:** render **Original name:** rendercubes.cpp
**Type:** Scripts **Size:** 15,984 B
**Source:** http://assault.cubers.net/
**License:** ZLIB license
**Description:** C++ source code from the AssaultCube game

**Name:** xmlevent **Original name:** xmlevents.php
**Type:** Scripts **Size:** 7,542 B
**Source:** http://www.micronetwork.de/activecalendar/
**License:** GNU Lesser General Public License
**Description:** PHP source code from the Active Calendar project

**Name:** mailflder **Original name:** mailfolder.py
**Type:** Scripts **Size:** 43,732 B
**Source:** http://www.cps-project.org/
**License:** GNU Lesser General Public License
**Description:** Python source code from the ECM framework, from the CPS project

**Name:** age **Original name:** age.xls
**Type:** Spreadsheets **Size:** 137,216 B
**Source:** http://gsociology.icaap.org/

**License:** Public domain
**Description:** Age structure in the world

**Name:** higrowth **Original name:** higrowth.xls
**Type:** Spreadsheets **Size:** 129,536 B
**Source:** http://pages.stern.nyu.edu/~adamodar/New_Home_Page/spreadsh.htm
**License:** Public domain
**Description:** Financial calculations

# Appendix D

# Public domain data sources

This part contains the list of websites of how to obtain data placed in the public domain, or their licenses allow the files to be freely used and distributed. The list is ordered by URLs, not by the data types, since some sources may contain various data.

A comprehensive list public domain resources is available at http://en.wikipedia.org/wiki/Public_Domain_Resource. Furthemore, this website also contains the search tips to find the public domain materials.

**URL:** http://www.baseball1.com/
**Description:** A large database containing the most complete set of baseball data with statistics from 1871–2009. It can be downloaded as `MDB` for Microsoft Access, as a `SQL` script or as comma-separated values.

**URL:** http://www.bls.gov/data/
**Description:** Bureau of Labor Statistics focus on collecting economic data, which are available mainly as Excel spreadsheets or plain text files.

**URL:** http://www.ccmixter.org/
**Description:** A database of music released under Creative Commons license.

**URL:** http://dumps.wikimedia.org/
**Description:** A complete copy of all Wikimedia wikis in `XML` format. A number of raw database tables (in `SQL` form) are available, too.

**URL:** http://www.eea.europa.eu/data-and-maps
**Description:** The European Environment Agency makes available the comprehensive environmental datasets, charts and other related data. These information can be downloaded in a large number of formats, e.g., `XML`, `MDB`, `TIFF` or `XLS`.

**URL:** http://www.esri.com/data/free-data/index.html
**Description:** Geographic data, such as 2D and 3D maps, and other geospatial files. Some databases in the `DBF` format are available.

**URL:** http://www.freesound.org/

**Description:** This website is a database of Creative Commons licensed sounds only, not songs, mainly in `AIFF` and `WAV` files.

**URL:** http://www.gaia-gis.it/spatialite/resources.html
**Description:** Sample databases in a sqlite format. Furthermore, the `EXIF` GPS pictures and some huge sized images are available.

**URL:** http://grass.osgeo.org/download/data.php
**Description:** Geospatial data and information provided by Geographic Resources Analysis Support System. The data sets contain vector, rastrer and satellite data.

**URL:** http://www.gutenberg.org/
**Description:** A well-known project Gutenberg is the place where it is able to download thousands of ebooks. All possible formats are often provided.

**URL:** http://www.imagecompression.info/
**Description:** A set of high-resolution high-precision images intended to be used for evaluating of compression methods.

**URL:** http://www.mlp.cz/
**Description:** The Municipal Library of Prague released the book by famous Czech writers, i.e., Karel Čapek or Božena Němcová. The books can be obtained as a `HTML` page, `PDF` or some other document formats.

**URL:** http://nationalatlas.gov/atlasftp.html
**Description:** Raw data from many fields (i.e. Agriculture, Geology) collected by National Atlas of the United States. Various files are available, such as `DBF`, `GeoTIFF`, or map layers in Shapefile format.

**URL:** http://nssdc.gsfc.nasa.gov/image/
**Description:** Public domain images of Space and Solar system.

**URL:** http://openclipart.org/
**Description:** A project maintaining the clip art to be freely used.

**URL:** http://sourceforge.net
**Description:** SourceForge is a web-based repository for open-source projects. Since many software is available, the various data can be downloaded.

# Appendix E

# Detailed results of experiments

This appendix contains the detailed information and results which were obtained during the experiments on both the Canterbury Corpus and the Prague Corpus data sets. In total, ten methods were tested.

As we can see in Figure E.1, the achieved compression ratios on the files from the Canterbury Corpus were always positive. This is in contrast the values in Table 5.5.

Figure E.2 shows that Shannon-Fano coding was the fastet one in all cases with regard to statistical methods. Nevertheless, the decompression time values in Table E.1 denotes that the differences between Shannon-Fano and Static Huffman coding are eliminated. The compression speed is the same as the time required for a decompression. LZ78 and LZW were very fast in all cases, even if processing the largest files available. The achieved decompression times are stated in E.3. The data for two files, `venus` and `nightsht`, are not provided in Table E.4 for DCA method, since we were not able to compress these files in a reasonable time.

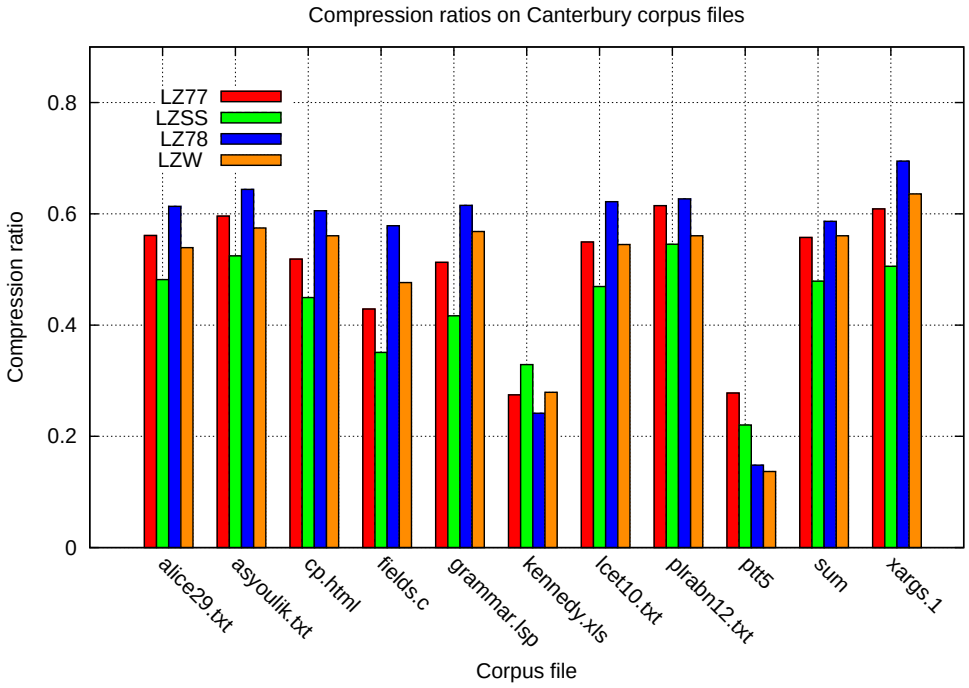



Figure E.1: Compression ratio of the dictionary methods on the files from the Canterbury files

| | Compression time [$\mu s$] | | | Decompression time [$\mu s$] | | | Compression ratio | | |
|---|---|---|---|---|---|---|---|---|---|
| File | SFano | SHuff | DHuff | SFano | SHuff | DHuff | SFano | SHuff | DHuff |
| firewrks | 652,632 | 2,283,204 | 6,906,797 | 309,874 | 310,332 | 6,831,238 | 0.961781 | 0.959567 | 0.960040 |
| thunder | 1,354,083 | 4,809,416 | 12,529,826 | 593,534 | 592,049 | 12,496,053 | 0.803622 | 0.799635 | 0.799896 |
| drkonqi | 45,532 | 125,441 | 342,607 | 28,135 | 17,774 | 339,975 | 0.696297 | 0.693092 | 0.693821 |
| libc06 | 34,156 | 52,593 | 136,974 | 11,990 | 7,517 | 135,683 | 0.648566 | 0.647839 | 0.648026 |
| mirror | 37,872 | 103,187 | 287,639 | 23,240 | 14,611 | 286,595 | 0.696707 | 0.692232 | 0.693123 |
| abbot | 161,851 | 534,720 | 1,565,722 | 76,296 | 76,379 | 1,562,171 | 0.996654 | 0.994328 | 0.995004 |
| gtkprint | 25,315 | 36,218 | 99,483 | 5,200 | 8,246 | 156,005 | 0.556257 | 0.555485 | 0.556709 |
| wnvcrdt | 92,011 | 168,788 | 295,737 | 30,937 | 19,582 | 288,438 | 0.234135 | 0.233891 | 0.233946 |
| w01vett | 402,739 | 832,868 | 1,706,126 | 103,195 | 102,869 | 1,522,457 | 0.298900 | 0.296871 | 0.296905 |
| emission | 882,428 | 2,497,724 | 6,159,451 | 324,793 | 325,014 | 6,798,754 | 0.530048 | 0.529070 | 0.529144 |
| venus | 6,919,808 | 22,671,853 | 60,089,951 | 2,863,556 | 2,908,650 | 60,950,415 | 0.845943 | 0.843725 | 0.843746 |
| nightsht | 7,982,439 | 27,700,377 | 77,547,962 | 3,737,691 | 3,873,055 | 75,159,159 | 0.957356 | 0.954643 | 0.954899 |
| flower | 4,933,640 | 17,526,204 | 47,694,876 | 2,215,176 | 2,123,933 | 47,897,812 | 0.894280 | 0.891376 | 0.891409 |
| corilis | 576,325 | 1,872,041 | 5,619,836 | 266,070 | 265,729 | 5,571,537 | 0.946389 | 0.941720 | 0.941891 |

Table E.1: The measured data of the statistical methods on the files from the Prague Corpus (set A)

| | Compression time [$\mu s$] | | | Decompression time [$\mu s$] | | | Compression ratio | | |
|---|---|---|---|---|---|---|---|---|---|
| File | SFano | SHuff | DHuff | SFano | SHuff | DHuff | SFano | SHuff | DHuff |
| bovary | 879,508 | 2,639,929 | 6,238,631 | 335,797 | 338,891 | 6,228,073 | 0.650319 | 0.648759 | 0.648834 |
| modern | 145,711 | 387,796 | 806,217 | 54,665 | 85,542 | 821,134 | 0.585397 | 0.583733 | 0.583784 |
| ultima | 476,801 | 1,526,985 | 4,715,326 | 214,474 | 217,348 | 4,749,181 | 0.904357 | 0.903240 | 0.903581 |
| lusiadas | 229,001 | 611,001 | 1,425,810 | 84,677 | 85,213 | 1,411,639 | 0.582221 | 0.580538 | 0.580725 |
| cyprus | 200,502 | 531,866 | 1,124,712 | 74,206 | 74,557 | 1,133,652 | 0.565559 | 0.565187 | 0.565243 |
| hungary | 1,344,941 | 4,496,406 | 9,122,491 | 501,285 | 497,637 | 9,045,276 | 0.571133 | 0.570062 | 0.570079 |
| compress | 44,947 | 122,556 | 282,805 | 27,881 | 17,663 | 282,000 | 0.673002 | 0.671650 | 0.671936 |
| lzfindmt | 14,604 | 39,400 | 59,785 | 3,783 | 5,734 | 59,270 | 0.664514 | 0.663337 | 0.663947 |
| render | 6,523 | 26,501 | 39,991 | 4,070 | 2,454 | 40,229 | 0.628566 | 0.625813 | 0.626752 |
| handler | 7,513 | 12,007 | 43,473 | 2,929 | 1,712 | 44,227 | 0.599343 | 0.598248 | 0.599343 |
| usstate | 3,438 | 12,725 | 29,068 | 2,099 | 1,199 | 18,751 | 0.567446 | 0.566234 | 0.567446 |
| collapse | 1,465 | 5,223 | 12,293 | 1,072 | 549 | 7,936 | 0.663880 | 0.660745 | 0.660049 |
| xmlevent | 5,330 | 13,229 | 31,189 | 1,962 | 1,219 | 31,815 | 0.662026 | 0.660567 | 0.661363 |
| mailflder | 16,023 | 39,833 | 86,914 | 5,699 | 5,509 | 87,825 | 0.532082 | 0.530070 | 0.530573 |
| age | 55,045 | 144,416 | 376,572 | 32,100 | 20,286 | 381,268 | 0.631916 | 0.626931 | 0.627558 |
| higrowth | 54,150 | 147,740 | 396,982 | 21,272 | 33,317 | 392,414 | 0.698949 | 0.694409 | 0.695189 |

Table E.2: The measured data of the statistical methods on the files from the Prague Corpus (set B)

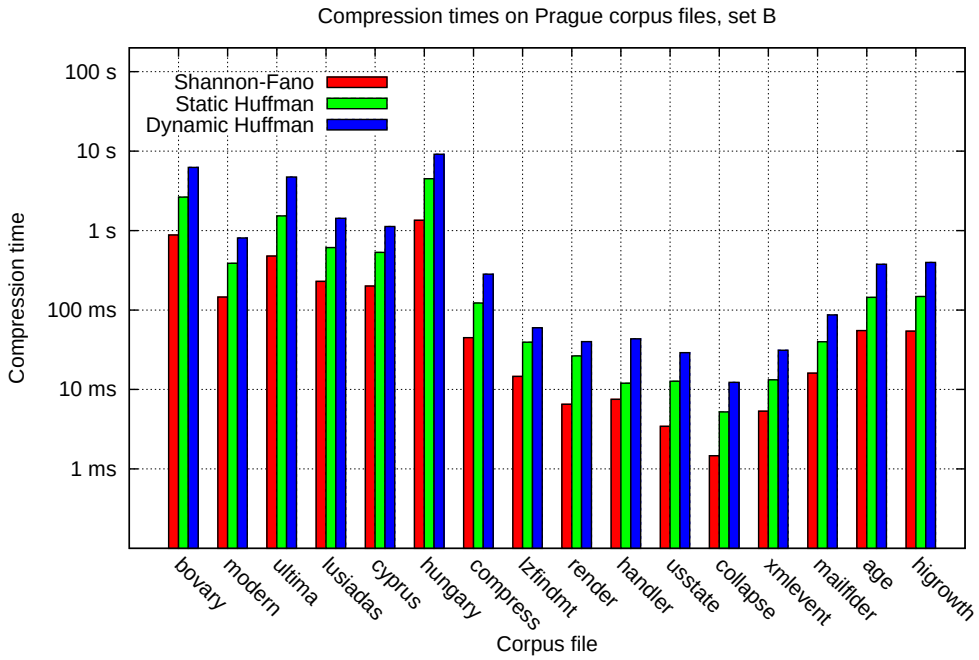



Figure E.2: Compression times of the statistical methods on the files from the Prague Corpus (set B)

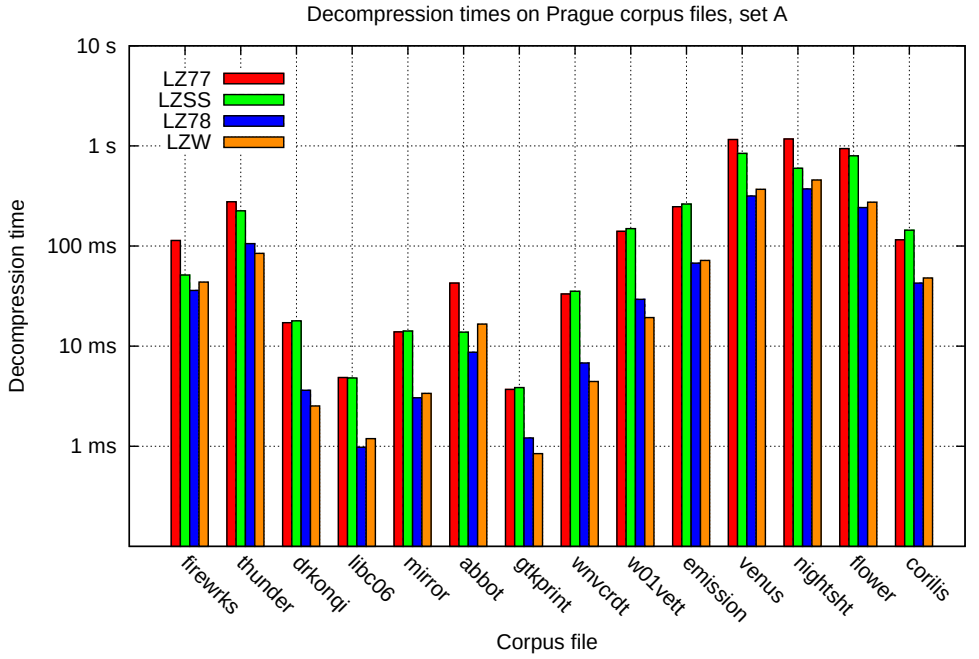



Figure E.3: Compression and decompression times of the dictionary methods on the files from the Prague Corpus (set A)

| | Compression time [$\mu s$] | | | | Decompression time [$\mu s$] | | | | Compression ratio | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| File | LZ77 | LZSS | LZ78 | LZW | LZ77 | LZSS | LZ78 | LZW | LZ77 | LZSS | LZ78 | LZW |
| bovary | 3,107,591 | 2,480,961 | 10,077 | 89,255 | 214,492 | 233,695 | 45,520 | 77,909 | 0.511931 | 0.450844 | 0.550717 | 0.501575 |
| modern | 563,239 | 585,215 | 30,541 | 28,037 | 37,464 | 67,584 | 13,091 | 8,802 | 0.612737 | 0.529551 | 0.641063 | 0.562803 |
| ultima | 722,768 | 935,617 | 106,713 | 27,859 | 92,993 | 74,382 | 38,513 | 28,651 | 0.960318 | 0.770651 | 0.872420 | 0.959221 |
| lusiadas | 1,152,022 | 4,812,306 | 24,995 | 35,832 | 95,727 | 105,913 | 17,900 | 18,915 | 0.509013 | 0.453445 | 0.475500 | 0.439710 |
| cyprus | 186,767 | 576,990 | 24,569 | 20,664 | 87,912 | 59,380 | 9,069 | 9,327 | 0.236742 | 0.178440 | 0.278043 | 0.221725 |
| hungary | 1,339,289 | 4,523,270 | 109,819 | 98,144 | 372,783 | 395,383 | 96,542 | 64,027 | 0.245873 | 0.186380 | 0.303466 | 0.245525 |
| compress | 99,431 | 139,282 | 7,536 | 4,100 | 11,188 | 18,407 | 3,625 | 2,429 | 0.371487 | 0.308153 | 0.517708 | 0.450612 |
| lzfindmt | 23,722 | 184,64 | 1,755 | 1,428 | 3,596 | 3,833 | 808 | 524 | 0.363712 | 0.293081 | 0.546287 | 0.451008 |
| render | 12,568 | 15,835 | 848 | 1,071 | 2,507 | 2,672 | 619 | 660 | 0.397335 | 0.330893 | 0.540478 | 0.475225 |
| handler | 13,057 | 17,170 | 563 | 783 | 1,194 | 1,247 | 297 | 328 | 0.403521 | 0.329234 | 0.501305 | 0.424745 |
| usstate | 6,869 | 14,712 | 400 | 563 | 1,324 | 1,379 | 355 | 386 | 0.408678 | 0.343231 | 0.488304 | 0.435826 |
| collapse | 1,801 | 2,169 | 263 | 243 | 456 | 462 | 89 | 104 | 0.587252 | 0.483455 | 0.703239 | 0.666667 |
| xmlevent | 5,690 | 12,575 | 593 | 528 | 1,164 | 1,211 | 194 | 194 | 0.453063 | 0.369133 | 0.573323 | 0.500000 |
| mailfilder | 63,374 | 179,183 | 1,969 | 2,535 | 4,371 | 4,642 | 913 | 959 | 0.380316 | 0.309705 | 0.467644 | 0.393945 |
| age | 312,791 | 727,557 | 9,880 | 9,108 | 20,979 | 20,944 | 2,781 | 3,220 | 0.584014 | 0.527628 | 0.549907 | 0.587264 |
| higrowth | 303,242 | 396,999 | 6,519 | 8,797 | 19,812 | 12,897 | 4,401 | 4,846 | 0.565210 | 0.490682 | 0.609537 | 0.596267 |

Table E.3: The measured data of the dictionary methods on the files from the Prague Corpus (set B)

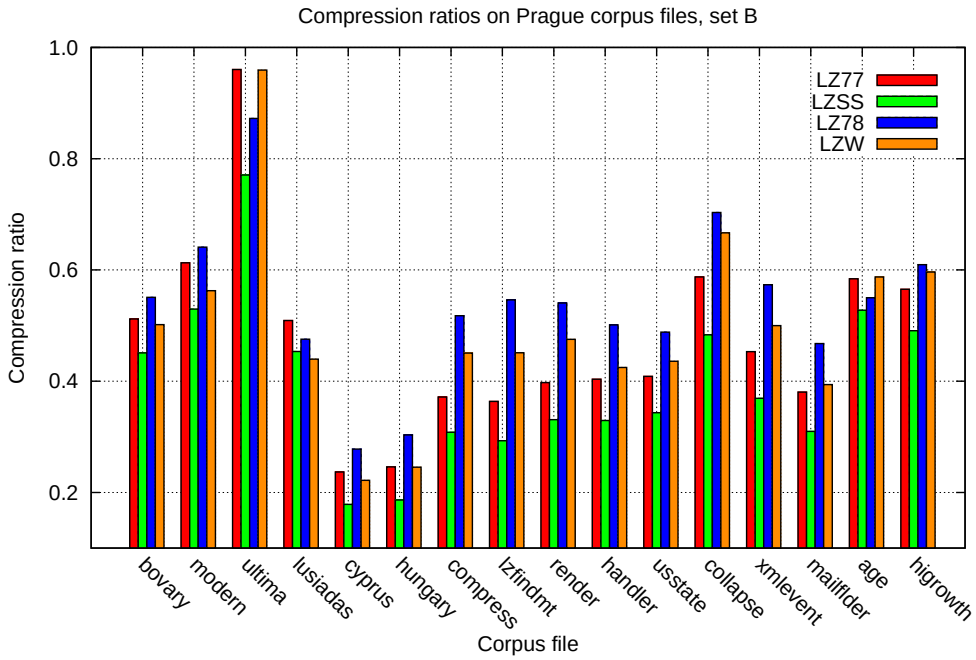



Figure E.4: Compression ratio of the dictionary methods on the files from the Prague Corpus (set B)

| | Compression time [$\mu s$] | | | Decompression time [$\mu s$] | | |
|---|---|---|---|---|---|---|
| File | ACB | DCA | PPM | ACB | DCA | PPM |
| firewrks | 48,057,985 | 5,242,565 | 1,274,331 | 41,197,944 | 5,104,709 | 1,440,946 |
| thunder | 124,252,963 | 5,552,569 | 2,430,774 | 94,710,430 | 6,115,115 | 3,008,003 |
| drkonqi | 3,751,589 | 132,677 | 26,821 | 4,050,922 | 141,355 | 47,112 |
| libc06 | 8,679,164 | 69,186 | 11,794 | 9,646,175 | 70,599 | 20,651 |
| mirror | 5,009,338 | 135,357 | 25,178 | 4,760,792 | 137,534 | 43,966 |
| abbot | 14,096,242 | 1,398,318 | 266,189 | 10,032,042 | 1,421,242 | 310,007 |
| gtkprint | 1,737,008 | 42,990 | 7,531 | 1,724,003 | 43,630 | 13,292 |
| wnvcrdt | 39,559,191 | 54,427 | 13,688 | 39,397,252 | 54,847 | 22,848 |
| w01vett | 78,944,558 | 203,145 | 61,698 | 82,867,377 | 294,805 | 66,710 |
| emission | 768,905,288 | 872,393 | 208,121 | 644,152,665 | 916,919 | 222,477 |
| venus | 474,042,940 | | 7,396,310 | 445,131,893 | | 8,495,536 |
| nightsht | 519,654,273 | | 14,094,811 | 465,438,447 | | 15,522,449 |
| flower | 402,620,708 | 10,393,393 | 3,869,849 | 356,905,754 | 10,098,828 | 4,166,439 |
| corilis | 98,848,120 | 2,041,264 | 496,440 | 81,034,005 | 2,615,244 | 552,184 |

Table E.4: The measured data of the context methods on the files from the Prague Corpus (set A)

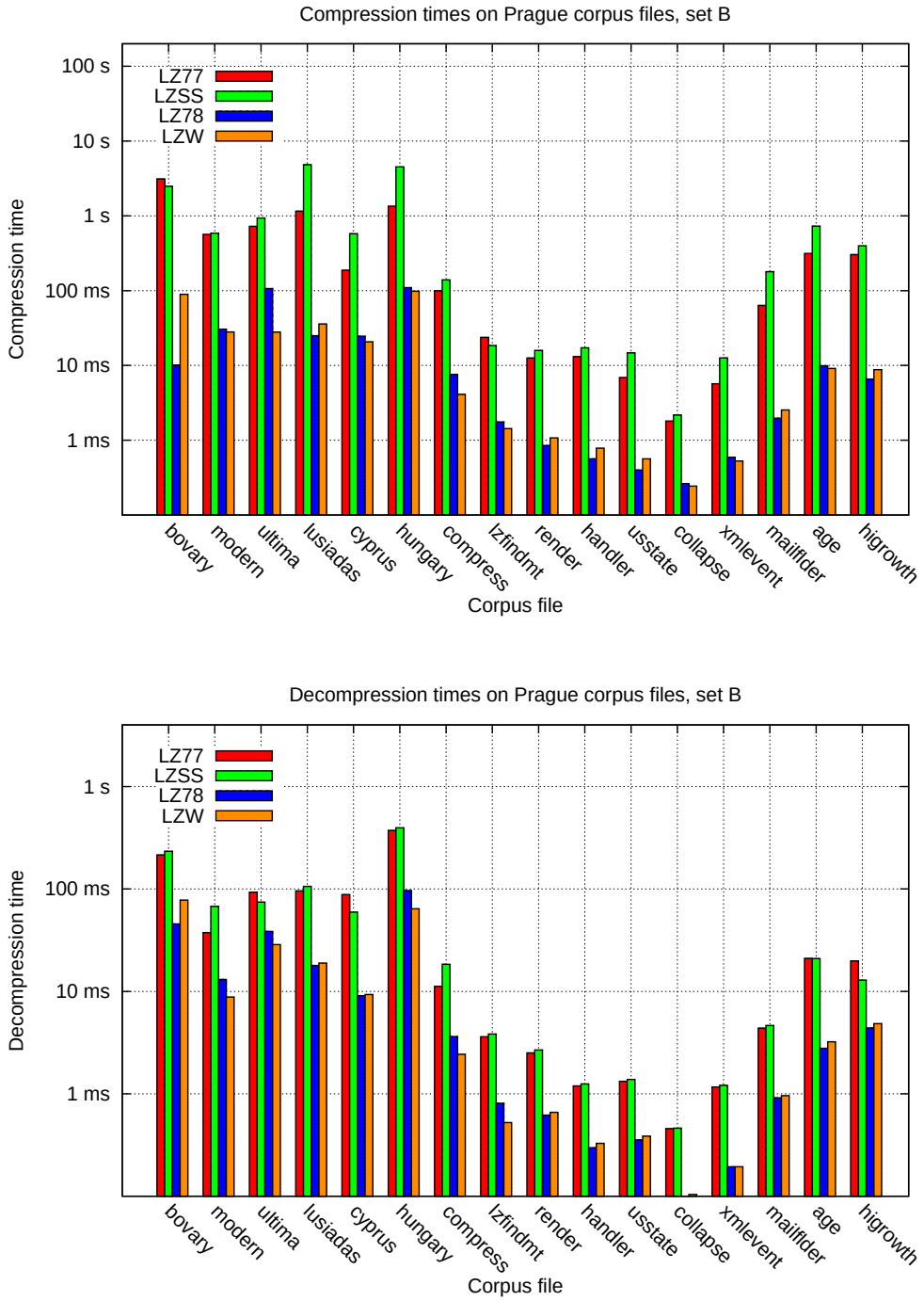



Figure E.5: Compression and decompression times of the dictionary methods on the files from the Prague Corpus (set B)

# Appendix F

# User manual

This appendix describes the usage of all implemented methods via the texting application, which can be found in `bin/src/app` after a building of the ExCom library. To build the library, see *Apendix E: Building the library* in [3]. To use this command-line application, some of the parameters must be set. It depends on the chosen method, however, the complete list of mandatory and optional parameters can be listed by running the following command in the shell:

```
$ ./app --help
Usage: ./app [options]
where options may be:
  -d, --decompress    decompress input file (default is to compress)
  -e, --except=<path> path to exceptions' file (required for DCA, ignored
                      for other)
  -f, --ignore-first  don't count first run to the overall timing
                      The first run may be way off because of empty cache
  -h, --help          print this help
  -i, --input=<path>  path to the input file
  -m, --method=<met>  select method <met>, use ? for a list
  -o, --output=<path> path to the output file
  -p, --param=<prm>   <prm> is a comma separated list of parameters
                      of the method, use ? for a list
  -q, --quiet         don't output anything except errors
  -r, --repeat=<T>    repeat the process T times
  -t, --timing        measure time spent by the process
```

The list of existing methods can be obtained by this command:

```
$ ./app -m ?
Supported compression methods:
  copy    Just copies input to output
  acb     Associative coder of Buyanovsky
  dca     Data compression using antidictionaries
  dhuff   Dynamic Huffman coding
  lz77    Lempel-Ziv compression method from 1977
  lz78    Lempel-Ziv compression method from 1978
  lzss    Lempel-Ziv-Storer-Szymanski compression method from 1982
```

```
  lzw     Lempel-Ziv-Welch compression method from 1984
  ppm     Prediction by partial matching
  sfano   Shannon-Fano coding
  shuff   Static Huffman coding
```

Notice, that Shannon-Fano, Static Huffman and Dynamic Huffman coding are parameter less. The list of parameters for the rest of our implemented methods follows.

LZ77 and LZSS share the same options for parameters. Hence, we mention it once:

```
$ ./app -m lz77 -p ?
Parameters available for compression method 'lz77':
  s=<N>    The size of the Search Buffer (1 <= N <= 65536)
  l=<N>    The size of the Look-ahead Buffer (1 <= N <= 65536)
```

```
$ ./app -m lz78 -p ?
Parameters available for compression method 'lz78':
  d=<N>    The size of the dictionary (1 <= N <= 65536)
```

The size of the LZW dictionary is always initialized to the value of 256. Therefore, the minimal value when the dictionary is emptied and initialized again, can be set to 257:

```
$ ./app -m lzw -p ?
Parameters available for compression method 'lzw':
  d=<N>    The size of the dictionary (256 < N <= 65536)
```

Suppose we want to compress the file `dataIn`, and the encoded output save to `dataOut` using the LZSS method with the Search Buffer size set to 8,129. The following example illustrates this:

```
$ ./app -m lzss -p s=8192 -i dataIn -o dataOut
Summary:
  Method: lzss
  Operation: compression
  Parameters: s=8192
  Repeat 1 time(s)
  Input file: dataIn
  Output file: dataOut
  Measure time? no
```

# Appendix G

# List of abbreviations

**ACB** Associative Coder of Buyanovsky

**AIFF** Audio Interchange File Format

**ASCII** American Standard Code for Information Interchange

**BPS** Bits Per Symbol

**CR** Carriage Return

**DCA** Data Compression using Antidictionaries

**EOF** End Of File

**ExCom** Extensible Compression Library

**FGK** Faller-Gallager-Knuth, a variant of Adaptive Huffman coding

**GIF** Graphics Interchange Format

**GNU** GNU's Not Unix

**GPS** Global Positioning System

**HTML** HyperText Markup Language

**IDE** Integrated Development Environment

**IO** Input/Output

**LF** Line Feed

**LGPL** GNU Lesser General Public License

**LZ77** Lempel-Ziv compression method from 1977

**LZ78** Lempel-Ziv compression method from 1978

**LZSS** Lempel-Ziv-Storer-Szymanski compression method from 1982

**LZW** Lempel-Ziv-Welch compression method from 1984

**PDF** Portable Document Format

**PPM** Prediction by Partial Matching

**STL** C++ Standard Template Library

**TIFF** Tagged Image File Format

**UTF** Unicode Transformation Format

**WAV** Waveform Audio File Format

**XML** Extensible Markup Language

# Appendix H

# Contents of DVD

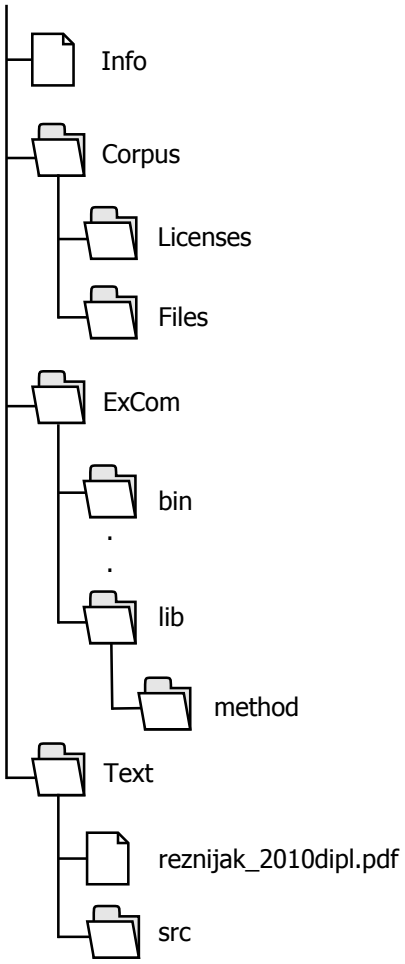



Figure H.1: The general structure of the attached DVD

- `Info` file contains a detailed description of the DVD content.
- The `Corpus` directory contains all the files of the Prague Corpus files along with appropriate licenses under which the files are released. Furthermore, all unused candidates are also available for the future testing purposes.

- All necessary files needed to build the ExCom library are located in the `ExCom` directory. The `bin` directory contains the application which can be used to test the implemented algorithms. The source codes of the implemented compression methods are placed in the `lib/method` directory.
- The `Text` directory contains this thesis both as the `PDF` file and in the form of LaTeX sources including the figures in various formats.