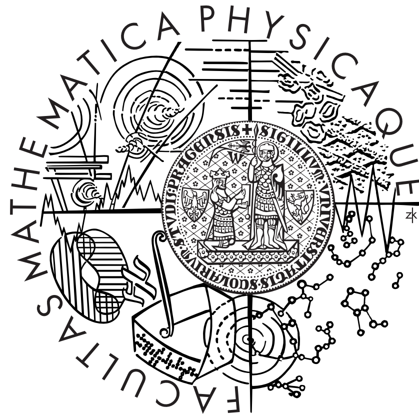
Charles University

Faculty of Mathematics and Physics



Syllable-based Compression

Ph.D. Thesis

Mgr. Jan Lánský

Department of Software Engineering
Malostranské náměstí 25
Prague, Czech Republic

Supervisor: Prof. RNDr. Jaroslav Pokorný, CSc.
Consultant: RNDr. Michal Žemlička, Ph.D.

2008

Abstract:

Classic textual compression methods work over the alphabet of characters or alphabet of words. For languages with rich morphology as well as for compression of smaller files it can be advantageous to use an alphabet of syllables. For some compression methods like the ones based on Burrows-Wheeler transformation the syllable is a reasonable solution also for large files - even for languages having quite simple morphology.

Although the main goal of our research is the compression over the alphabet of syllables, all implemented methods can compress also over the alphabet of words. For small files we use the LZW method and Huffman coding. These methods were improved by the use of initialized dictionary containing characteristic syllables specific for given language. For the compression of very large files we implemented the project XBW allowing combination of compression methods BWT, MTF, RLE, PPM, LZC, and LZSS. We have also tried to compress XML files that are not well-formed.

When compressing over a large alphabet, it is necessary to compress also the used alphabet. We have proposed two solutions. The first one works well especially for small documents. We initialize the compression method with a set of characteristic syllables whereas other syllables are coded when necessary character by character. The second solution is intended for compression of larger documents. The alphabet of used syllables is encoded as a compressed trie what significantly reduces the space necessary for encoding of the alphabet.

Keywords: text and XML compression, syllable and word based compression, Burrows-Wheeler transformation.

Thanks to Prof. RNDr. Jaroslav Pokorný CSc. for his helpfulness while supervising this work.

Thanks to RNDr. Michal Žemlička Ph.D.[123], RNDr. Tomáš Dvořák CSc.[1], RNDr. Leo Galamboš, Ph.D.[2], Mgr. Zuzana Vlčková[2], Filip 'Fajlajpi' Huta[3], and Jiří Marchalín[3].

Thanks to my students, Mgr. Katsiaryna Chernik[2], Mgr. Radovan Šesták[23], Petr Uzel[23], Mgr. Stanislav Kovalčin, Pavol Kumičák, Tomáš Urban, Mária Szabó, Bc. Tomáš Kuthan[23], Bc. Stanislav Kuřík, Bc. Ondřej Kazík, Mgr. Jan Chroustovský, and David Bertoli for programing my idea.

[1] thanks for helpful hints
[2] thanks for helpfulness with writing papers
[3] thanks for helpfulness with grammar correction of this thesis

Acknowledgement: My research was partially supported by Charles University Grant Agency in the project "Text compression" (GAUK no. 1607, section A) and by the Program "Information Society" under projects 1ET100300419 and 1ET100300517.

Prague, 24th August 2008 Mgr. Jan Lánský

Contents

1 Introduction **7**
1.1 Topics of Research 8
1.2 Publications 9
1.3 Contents of the Thesis 11

2 Lossless Compression **12**
2.1 Basic Definitions 12
2.2 Source Modeling 15
2.3 Compression Methods 16
2.4 Statistical Methods 16
2.4.1 Huffman coding 17
2.4.2 Arithmetic coding 18
2.4.3 Prediction by Partial Matching 18
2.5 Dictionary-Based Methods 19
2.5.1 LZW 19
2.5.2 LZSS 20
2.6 Transformations 20
2.6.1 Burrows-Wheeler Transformation 21
2.6.2 Move to Front 21
2.6.3 Run Length Encoding 22
2.7 Coding of Integers 22
2.8 Word-based Methods 23

3 Related Works **25**
3.1 Syllable-based Compression 25
3.2 Language Dependent Methods 26
3.2.1 Words Replacing Methods 26
3.3 Small Text Files Compression 27
3.3.1 Compression of Text for mobile phones 27
3.3.2 Compression Using Optimal Tree Machines 28
3.4 XML Compression 28

3.5 Set of Strings Compression 29
3.5.1 Main Memory Methods 29

4 Syllable-Based Compression 31
4.1 Languages and Syllables 31
4.2 Syllables 32
4.3 Problems in Decomposition of Words into Syllables 33
4.4 Definition of Syllable 34
4.5 Examples of Languages 40
4.6 Algorithms of Decomposition 42

5 Small Text Files Compression 46
5.1 Introduction 46
5.2 Methods with Static Initialization 47
5.2.1 LZWL 49
5.2.2 HuffSyllable (HS) 49
5.2.3 Coding New Syllable. 51
5.2.4 Experiments 52
5.2.5 Results 52
5.3 Methods without Static Initialization 54
5.3.1 Burrows-Wheeler Transformation 54
5.3.2 Technical Details 54
5.3.3 Experiments 55
5.3.4 Results 56
5.4 Conclusion 56

6 Small Textual XML Files Compression 58
6.1 XMLSyl 59
6.1.1 Architecture and Principles of XMLSyl 59
6.1.2 Encoding the Structure of XML document 60
6.1.3 Containers 63
6.1.4 The Syllable Compressor 63
6.2 XMillSyl 64
6.2.1 Implementation 64
6.3 Experiments 65
6.3.1 XML Data Sources 65
6.3.2 Compression Performance Metrics 66
6.3.3 Experimental Results 67
6.4 Conclusion 68

7 Large Alphabet Compression **69**
7.1 Static Approach 69
7.2 Semi-adaptive Approach 70
7.3 Adaptive Approach 70
7.4 Set of Strings Compression 71
7.4.1 Existing Methods 72
7.4.2 Trie-Based Compression 73
7.4.3 Text-Based Set of Strings 77
7.4.4 Results 80
7.4.5 Conclusion 82
7.5 Sets of Characteristic Syllables 83
7.5.1 Creating and Criteria 83
7.5.2 Cumulative Criterion 84
7.5.3 Appearance Criterion 85
7.5.4 Genetic Algorithm Criterion 85
7.5.5 Experimental Results 92
7.5.6 Conclusion 92

8 Large Text Files Compression **94**
8.1 Project XBW 94
8.2 Parts of Project XBW 95
8.2.1 Parser 95
8.2.2 Compression of Set of Used Elements 97
8.2.3 BWT 97
8.2.4 MTF and RLE 98
8.2.5 Dictionary Methods and PPM 98
8.2.6 Entropic Coders 99
8.3 Ambiguity of Name XBW 99
8.4 Corpora 100
8.5 Results 100
8.5.1 Influence of File Size 105
8.5.2 Comparison with Other Programs 105
8.6 Future Work 106

9 Conclusion **109**
9.1 The Assets of the Thesis 111
9.2 Future Work 112

List of Publications **113**

Bibliography **116**

Chapter 1

Introduction

There are usually two reasons for the use of various compression methods - either they serve to decrease hard drive space usage or they decrease the necessary bandwidth for network file transfers.

Given the current increase in overall storage capacity, the issue of compressing files to save hard drive space for the average user becomes limited to the lossy compression of images, sound or video. There are still certain special areas, for example web search engines, where the problem still exists, since they archive web pages from the majority of the internet. In our thesis, one of the problems we examined was data compression for the Egothor search engine [45].

The second reason is usually decreasing the used bandwidth during network transfers. For the common user, this problem is still actual, for example when he is charged for the amount of data transferred. Using compression in this area can also be profitable for the Internet Service Provider by decreasing the outgoing data flow, which can otherwise be very limiting. For this reason, even the compression of very small files becomes important, which was another of the cases examined in this thesis.

There is a third reason for using a compression. Due to its relative recent appereance, it is not usually mentioned. If the compression method is fast enough, compression can be used to speed up the process of reading very large files from hard drives. Since most of today's CPUs can decompress files in less time than the difference between loading compressed and uncompressed file from the hard drive. Usually, these methods do not achieve the best compression ratio but these methods are very fast, for example gzip [44] or compress [107]. Compression methods proposed in this thesis are focusing on achieving the maximal possible compression ratio, and therefore are not particularly suited for this application.

Compression can be divided into two groups - lossy and lossless. During

lossy compression, there is a part of the original information which is lost, but which is also not important due to the specifics of the given format. For example, in audio or video compression it is the information which cannot be perceived by the human ear or eye. When using high amount of compression, it is usually the information the loss whose is the least distracting. On the other hand, in lossless compression the compressed file is, upon decompression, identical to the original file. In the compression of text files, lossless compression is always used. For XML [115] compression, partly lossy compression is usually used. The information contained within the XML file is unchanged, but the formatting is lost, which does not matter as much. For the purpose of this thesis, by compression we always mean lossless compression, unless explicitly stated otherwise.

1.1 Topics of Research

In this thesis, we were examining four main topics of the text compression. Each of them has different problems and their solution uses different compression methods. The topics are not, however, completely distinct and independent. The results of one topic have been used in another one.

1. Small text file compression: The file size is ranging from single kilobytes to tens of kilobytes, rarely up to single megabytes. Files of this size are compressed not to save hard drive space, but mostly to save bandwidth during their transfer, for example across the internet. Web pages are usually between 10 and 20 KB large, according to [78], and they contain large amounts of text. Small file compression methods can therefore be used as the basis for web page compression.

2. Large text file compression: The file size is approximately 20 MB. Files of this size were chosen because files of similar size are being used by the Egothor search engine [45], storing 1000 web pages in single file. Since Egothor has to store a vast amount of data downloaded from the internet, it is having difficulties with hard drive space. By using a suitable compression method, the number of stored web pages with the current drive capacity can be significantly increased.

3. Text XML file compression: While the first two topics examined pure text compression and the possibility of using the methods for web page compression was mentioned, this topic is directly concerned with it. Web pages are mostly written in the HTML format [116], often

containing lots of errors from the HTML norm. In this section, we need to examine the compression of non-well-formed XML.

When we want to compress each web page separately, we will be using the knowledge gained in the first topic. On the other hand, when we want to compress a larger number of pages as a single file, we will utilize the methods from the second section. At first, we were designing compression methods for well-formed XML, later we turned our attention to compressing non-well-formed XML, which can in turn be used to compress HTML files.

4. Large alphabet compression: In this thesis we study the methods of syllable-based and word-based compression. When we use syllables or words as source units (elements of the alphabet, which will be used for the file compression), we have to transmit the information about the elements of the alphabet between the encoder and the decoder. For the compression purposes we can understand the alphabet as a set of elements, so we do not need to know how they are ordered. The ordering is created during the encoding and it must be respected during the decoding, explicitly or implicitly. The alphabet has to be attached to the encoded message for the transfer between the encoder and decoder. It is useful to compress it as much as possible, which is also examined in this topic. The results from this section will improve the results achieved by the compression methods from topics 2 and 3.

1.2 Publications

The content of this thesis was published in [1] through [21]. For easier orientation, articles written by the author were put in the beginning of the reference list. For the same reason his works are sorted by publishing date and not alphabetically. It is therefore easy to determine, without searching the references, what was the author involved in. From the list of published works, 17 of them are in peer reviewed conference proceedings [2, 3, 5, 6, 7, 8, 9, 11, 12, 13, 14, 16, 17, 18, 19, 20, 21], six of those in IEEE ([3, 11, 12, 19, 20, 21]) and one in LNCS [18]. Another article is in the IJIT journal [10].

In Master thesis [1], we have examined the theoretical basis of syllable-based compression and defined the problems that needed solving. We have also created, implemented and tested LZWL and HuffSyll algorithms. Extract from this work was published [2].

Another work [3] was focused on the right configuration of the characteristic words and syllables sets used by LZWL and HuffSyll algorithms for initialization. Extended version of this article was published as a technical report [4]. Edited Czech version was published in [17].

In another article [5] we have introduced compression algorithms (TD1, TD2 and TD3) based on compressing a set of strings, whose usage was planned for the XBW algorithm. The algorithm was based on encoding the structure of the trie in which the strings were stored. We have encoded the number of children for each node, the distance from its left sibling and one bit that determined whether the given string belonged into the set or not. Along with this article we have published methods XMillSyl and XMLsyl [6] and their results for small XML files. These methods combined XML and text compression.

Following that, we have published an article about a compression method for large XML files, based on Burrows-Wheeler Transformation [7]. Journal article [10] is an improvement of this concept. In article [8] we have examined the properties of XMillSyl and XMLSyl algorithms and other methods for XML compression on large files in non-well-formed format. Article [9] contains selected interesting facts about syllable-based compression. Article [11] is an abstract of the article [5] about methods for compression of a set of strings.

Article [12] offers a comparison of various approaches to small and medium text file compression using Burrows-Wheeler transformation. While in the case of the smallest files, letter-based compression was the best approach, for larger files, syllable-based approach was better. Word-based compression was also examined, however it did not excel in any category. Extended version of this article was published as [13].

Article [14] examines suitable inflation of the characteristic syllables and words sets, which are used to initialize the HuffSyll and LZWL methods.

XBW Project studied the compression of large, non-well-formed XML files. The basis for XBW was the Burrows-Wheeler transformation, but for subsequent phases of compression, besides the usual combination of MTF and RLE, also dictionary methods LZC and LZSS or statistical method PPM. We can also decide whether large output should be encoded using arithmetic coding or Huffman coding. The project is freely downloadable [15]. Source codes, executables and documentation, including measured results, are available. Description of the project and its results were published in [16, 18]. Different types of algorithms for Burrows-Wheeler transformation over large alphabet used in XBW are described in [19].

A new statistical lossless compression method called MultiStream compression, created by Kochánek, is desribed in [20, 21]. This method was

added into XBW. We used advantage of this modular system for testing this method.

1.3 Contents of the Thesis

In Chapter 2, we gather the basic definitions from the area of lossless compression and described the principles of the most common compression methods.

In Chapter 3, we describe works related to the topics of the research. This chapter concludes the already known facts, all other chapters are author's original research.

In Chapter 4 we explain the motivation behind the research of the syllable-based compression. We also state the problems which need to be solved. Then we define certain concepts, for example: letter, consonant, vowel, word, decomposing words into syllables. We have included examples of algorithms for the decomposition process.

Chapter 5 examines the compression of small text files up to 5 MB of size. We have introduced two syllable-based compression methods, LZWL and HuffSyllable, and compared the results with the results of word-based compression methods.

Chapter 6 studies the compression of small XML files, comprising mostly of text. We join the principles of XML compression with the principles of word- and syllable-based compression. We have also introduced compression methods XMillSyl and XMLSyl.

Compression methods working over large alphabet (syllables, words) must be able to transmit this alphabet in some form between the encoder and the decoder. In Chapter 7, we describe certain approaches to this problem, more thoroughly examining the semi-adaptive and adaptive with a static initialization approaches.

Chapter 8 examines the compression of large text files in non-well-formed XML format. The core of this chapter is the description of the XBW software project.

Chapter 2

Lossless Compression

There exist many books, papers, and technical reports [79, 36, 27, 65, 70, 89, 88] giving the overview about the basic definitions and notions of data compression. There are described also the principles of the most important compression methods. Our goal is not the detailed description of the known methods, we will give the brief overview about the ideas. We will focus to the methods of the lossless compression as lossy compression of the text has less importance than it has for multimedia.

The basic definitions from the compression domain are in Chapter 2.1. Chapter 2.2 describes different types of methods used in compression: static, semi-adaptive, and adaptive. The basic classification of compression methods is in Chapter 2.3. Chapter 2.4 describes basic statistical compression methods, Chapter 2.5 basic dictionary methods. Chapter 2.6 describes basic transforms used in compression. Chapter 2.7 is dedicated to the integer compression. Chapter 2.8 describes word-based compression methods.

There are many methods that can be included in Chapters 2.4 - 2.7, but we describe methods used in the following parts of this thesis only.

2.1 Basic Definitions

This chapter collects basic definitions from the compression area. We recall the ones from the original sources [70, 93] and also their later interpretations [109, 38].

Definition 2.1 (Alphabet):

Let Σ be a finite nonempty set of characters, then Σ is called *alphabet*. The cardinality of Σ is denoted as $|\Sigma|$.

Definition 2.2 (String):
A sequence of characters $\alpha = \alpha_1 ... \alpha_n,\ \alpha_i \in \Sigma$ is called *string over alphabet* Σ. The *length* of α is $|\alpha| = n$. Symbol $\lambda \notin \Sigma$ is called *empty string*. We denote the set of all finite strings over alphabet Σ (including the empty string λ) by Σ^*, and the set of all nonempty strings over Σ by Σ^+.

Definition 2.3 (Substring):
Let $\alpha = \alpha_1 ... \alpha_n,\ \alpha_i \in \Sigma$ and $\beta = \beta_1 ... \beta_m,\ \beta_i \in \Sigma$. If $\exists h \geq 0$ for $i = 1, ..., m : \beta_i = \alpha_{i+h}$ then β is a *substring* of α. If $h = 0$ then β is a *prefix* of α, if $h = n - m$ then β is a *suffix* of α.

Definition 2.4 (Concatenation):
Let $\alpha = \alpha_1 ... \alpha_n,\ \alpha_i \in \Sigma$ and $\beta = \beta_1 ... \beta_m,\ \beta_i \in \Sigma$ and $\gamma = \gamma_1 ... \gamma_{m+n},\ \gamma_i \in \Sigma$. If for $i = 1, ..., n : \alpha_i = \gamma_i$ and for $i = 1, ..., m : \beta_i = \gamma_{n+i}$ then the string γ is a *concatenation* of the strings α and β, we denote $\gamma = \alpha \cdot \beta$.

Definition 2.5 (Code):
The *code* K is a triple $K = (S, C, f)$, where S is a finite set of *source units*, C is a finite set of *code units*, and f is a mapping from S to C^+. The mapping f assigns to every source unit from S just one *codeword* from C^+. The string $\alpha = \alpha_1, ..., \alpha_n,\ \alpha_i \in S$, is called *message*.

A codeword consists of a sequence of code units. Two distinct source units should be never assigned to the same codewords, therefore f has to be an injective mapping.

The set of source units is usually an alphabet. We can understand the set of source units also as n-tuples of the characters called n-grams. For the text compression methods the set of source units can be the set of words, syllables, or other groups of characters. We called these sets of source units as *alphabets of n-grams, words, syllables*. If we know the set of source units before the start of the compression and the set of source units is reasonably small (having e.g. up to 256 items), then we can talk about the *compression over a small alphabet*, in the opposite case we talk about the *compression over a large alphabet*.

Definition 2.6 (Uniquely Decodable):
We say that a string $\alpha \in C^+$ is *uniquely decodable* with respect to f, if there is at most one message $\beta \in S^+$ such that $f(\beta) = \alpha$. Similarly, the code $K = (S, C, f)$ is *uniquely decodable*, if all strings in C^+ are uniquely decodable with respect to f.

Definition 2.7 (Prefix Code):
The code K is said to be a *prefix code*, if it has a prefix property, which requires that no codeword is a proper prefix of any other codeword.

Prefix codes represent an important and frequently used class of codes, since they are uniquely decodable. The class of uniquely decodable codes does not offer any further compression opportunities, any hope for minimizing the average codeword length, than the class of prefix codes (see e.g. [36], Chapter 5.5).

Definition 2.8 (Encoding and Decoding):
The conversion of the original messages to the encoded (or compressed) messages is referred to as *coding* or *encoding*. In the reverse process, *decoding*, the compressed messages are decoded (or decompressed) to reproduce the original messages. The corresponding algorithms are called *encoder* (or *compressor*) and *decoder* (or *decompressor*), respectively.

The compression ratio gives an average number of bits required for encoding a single character of 8-byte alphabet.

Definition 2.9 (Compression Ratio):

$$\text{Let } \nu = \frac{Size\ of\ compressed\ data}{Size\ of\ original\ data}$$

We define *compression ratio* as $\nu * 8$.

Definition 2.10 (Entropy of Unit):
Let $S = \{x_1, x_2, \ldots, x_n\}$ be the set of source units. Let p_i be the probability of occurrence of source unit x_i, $1 \leq i \leq n$. The entropy of unit x_i is then:

$$H_i = -\log_2 p_i \text{ bits.}$$

Definition 2.11 (Entropy of Message):
Let $S = \{x_1, x_2, \ldots, x_n\}$ be the set of source units. Let p_i be the probability of occurrence of source unit x_i, $1 \leq i \leq n$. The entropy of a source message $X = x_{i_1} x_{i_2} \ldots x_{i_k}$ from S^+ is then:

$$H(X) = -\sum_{j=1}^{k} p_{i_j} \log_2 p_{i_j} \text{ bits.}$$

Both definitions of entropy are from Shannon's work [93] from 1948. Entropy of a message is a measure of quantity of the information encoded in the message. The larger entropy of a message is the higher information quantity it has.

In general, it is not possible to know the entropy of message, so we have estimate the entropy. The estimate of the entropy depends on our assumptions about the structure of the message. It is impossible to compress a message by lossless statistical method to be smaller than its entropy.

2.2 Source Modeling

The development of data compression algorithms for a variety of data can be divided into two phases. The first phase is usually referred to as *modeling*. In this phase we try to extract information about any redundancy that exists in the data and describe the redundancy in the form of *model*. The second phase is called *coding*. A description of the model and a "description" of how the data differ from the model are encoded, generally using a binary alphabet.[1]

For example: a compression method for a sequence of integers $a_1, ..., a_n$ can use the model $a_i = i * 50$. If a message is 48, 100, 150, 201, 249 then the message is coded into -2, 0, 0, 1, -1.

Compression works with an assumption that the data corresponds to some model and the data is compressed based on the model. The more similar to the model the data is, the better compression ratio can be achieved. It is necessary that both coder and decoder use the same model to ensure that the file can be decoded. Models are usually divided into three groups: static, semi-adaptive and adaptive.

Static model is usually constructed for a particular type of file, mostly based on analysis of characteristic attributes of the given file. Coder and decoder then use this model without modifying it. Also the model is not affected by the input file. Problems may arise in case the input file does not conform to the model, for example, if we have a model for text files and the input file is compressed video. It is possible that substantial expansion instead of compression may happen in this case.

Semi-adaptive model most often requires two-pass compression method. The model is constructed at the first pass through the file and then, at the second pass, the file is compressed using this model. Model has to be encoded and transferred from coder to decoder which causes some overhead.

[1] This paragraph is taken from [89].

This overhead can be bigger than the size of the original document while compressing very small documents.

Adaptive model is constructed by both coder and decoder based on the already processed part of the file. The advantage is that the model does not need to be transferred from coder to decoder. A disadvantage is that the model is not able to learn on the very small files and thus the compression result is not usually very good. Adaptive models are also usually quite slower than the other two types.

2.3 Compression Methods

Compression methods are usually divided into two groups: statistical methods (Chapter 2.4) and dictionary-based methods (Chapter 2.5). There are also methods for compression of integer numbers (Chapter 2.7) and also various transformations (Chapter 2.6) that can be used along with other compression methods, but often also independently.

Another way to divide compression methods is based on the size of used alphabet. The oldest methods work with single characters, the newer ones work with character sequences of fixed length (n-grams). In case of text compression, there are also word-based (see Chapter 2.8) and syllable-based methods.

2.4 Statistical Methods

Statistical compression methods work with the probability of occurrence of individual characters from the alphabet. These probabilities can be determined statically, adaptively or semi-adaptively. During the compression, individual elements of the input alphabet are read and a probability of occurrence of every such element in the given model is found out. This probability is coded into the output file.

Statistical compression methods can be divided based on the order of compression. The order determines how many preceding characters are taken into account for calculating the probability of actual character. Model is called *0-order* if it does not take into account any preceding characters. If the model takes into account x preceding characters, it is called *x-order*.

The most important representatives of statistical compression methods are Huffman coding, arithmetic coding and also Prediction by Partial Matching.

2.4.1 Huffman coding

Huffman coding [54] is statistical compression method that assigns codes of fixed size (usually a number of bits) to individual characters. Huffman coding generates optimal prefix code. If all probabilities of occurrence of individual characters from the alphabet are negative powers of two, then each encoded character has size of its entropy. In other cases the size of each encoded character is lower than its entropy plus one bit.

Codes for every character are stored in the binary tree called Huffman tree in which edges are labeled with characters 0 and 1. The code for any character can be determined by traversing the tree from the root to the leaf that corresponds to this character.

Static Huffman Coding

In case of static or semi-adaptive version, character counts are known before constructing the Huffman tree. These characters are sorted in ascending order according to their counts. In every step, the two elements A and B with the lowest counts in the list are chosen. New node C is created with count equal to sum of counts of nodes A and B. A and B become children of the node C. Node C is inserted into the sorted list to the appropriate position. This procedure is repeated until only one element remains in the list. Huffman tree constructed in the presented way can be used for compression of the input file.

Maintaining a sorted list of alphabet symbols during the construction of the Huffman tree is not only way to go. Actually, using a binary heap is better in the worst case.

Adaptive Huffman Coding

When using the adaptive version [59], we start with a tree that contains only one node, which represents escape symbol. Escape symbol is used for inserting new characters that have not occurred in the tree yet. When inserting new node, we have to create new node A that represents this character and also new node B that represents escape symbol. Both nodes A and B will have weight 1 and node C representing original escape symbol becomes their parent. Weight of node A is increased according to the procedure described in the following paragraph. This procedure is also used for the elements that were already present in the tree.

The update procedure requires that the nodes be numbered in a fixed order. The largest node count is given to the root of the tree. As we progress deeper into the tree from the root, the number are decremented. The smallest

number is assigned to the escape symbol. The set of nodes with the same count makes up a block. When increasing count of the character, the node N representing this character is moved rightmost in the group of nodes with the same count. The node can not change position with its parent. Afterwards, count of this character is increased and the whole procedure is repeated recursively for the parent of N. Recursion stops in the root.

2.4.2 Arithmetic coding

Arithmetic coding is most often used in its adaptive version [74]. The idea of arithmetic coding is to represent the input file as a number from interval [0,1). Arithmetic coding usually achieves better compression ratio than Huffman coding because characters are not assigned with codes of fixed size length.

Suppose we have a string composed of characters over some finite alphabet. Suppose that p_i is the probability of the i-th character in the alphabet, and that variables L and R are initialized to 0 and 1 respectively. Value L represents the smallest binary value consistent with a code representing the characters processed so far, and Rrepresents the product of the probabilities of those characters. To encode the next character, which (say) is the j-th of the alphabet, both L and R must be refined: L is replaced by $L + R\sum_{i=1}^{j-1} p_i$ and R is replaced by $R \cdot p_j$, preserving the relationship between L, R, and the characters so far processed.

At the end of the message, any binary value between L and $L + R$ will unambiguously specify the input message. We transmit the shortest such binary string.

2.4.3 Prediction by Partial Matching

Prediction by Partial Matching (PPM) [33] is adaptive statistical method that makes use of contexts of variable order. The context is a finite sequence of characters preceding the current character. The length of the sequence is called an order of the context. The context model keeps information about count of characters' appearances for the context.

This method tries to encode the character according to context of the highest order k it was constructed for. If this is not possible because the character has not occurred in such context yet, then it switches to the lower order context $k - 1$ using escape mechanism. In the worst case, usually at the beginning of compression, it can get to the order -1, where all characters have the same probability. Context order, which the character was encoded in, is denoted as n.

Compressed character is added to the context of order $n+1$ and its count is increased in all contexts of order 0 to n.

In PPM, accent is put to find appropriate probability to model usage of the escape symbol. PPMA and PPMB methods were introduced in the original article. There are many enhancements to the basic method, such as PPMC [72], PPMD [53], PPMII [95], and [96].

2.5 Dictionary-Based Methods

Dictionary-based methods are mostly adaptive and they update the phrase dictionary during compression. The method then searches for the longest match of some phrase in the dictionary with prefix of the non-coded part of the document. Dictionary-based methods are especially suitable for files with many repetitious long strings (max. 5-15 characters), for example text files.

There are two main types of dictionary-based methods. One type is based on the LZ77 method [117], the second on LZ78 method [118]. LZ77 method has the dictionary represented in the encoded part of the document in form of sliding window of fixed size length, which gradually moves to the right. On the contrary, LZ78 method constructs the dictionary explicitly. In every step of compression, one phrase is extended - one that was used for compression. It is extended with one character that follows it in the non-coded part of the document

There are a lot of enhancements to both methods, for example LZW [112], that is an enhancement to LZ78, and LZSS [100], that enhances LZ77.

2.5.1 LZW

Algorithm LZW [112] is a dictionary compression character-based method based on LZ78. The algorithm uses a dictionary of phrases, which is represented by a trie data structure. Phrases are numbered by integers according order of adding.

In the initialization step the dictionary is filled up with all characters from the alphabet. In each next step it is searched for the maximal string S, which is in the dictionary and matches a prefix of still non-coded part of the input. The number of phrase S is then sent to the output. Actual input position is moved forward by the length of S. A new phrase F is created by concatenating of S and actual character in the input. The phrase F is added to the dictionary.

Only one complicated situation can occur in decoding phase. We can receive a number of a phrase which is not in the dictionary. In this case we can create that phrase by a concatenation of the last added phrase with its first character.

LZC method [50] is a slightly modified method described above. Codes with increasing code length is used to code number of the phrase. The second enhancement is cleaning the dictionary if the compression ratio starts to deteriorate. Some or all phrases are removed from the dictionary.

2.5.2 LZSS

Algorithm LZSS [100] is a dictionary compression character-based method based on LZ77. Similarly to LZ77, we also have dictionary of phrases represented by a sliding window that gradually moves to the right. The compression method searches for the longest match of non-coded part of the document with some string S in the sliding window.

LZ77 method outputs always ordered triple $\langle D, L, N \rangle$ where D is distance between the string S in the sliding window from the beginning of the window, L is the length of S and N is the character that follows in the non-coded part of the document just after S.

Disadvantage of LZ77 is that it is necessary to output needlessly long codes in case there is no match, i.e. the string S is of zero length. The LZSS enhancement consists in the fact that some minimal length of the string S is chosen - in case the string is shorter than the minimal length, only the first character N is coded and the rest of the string is tried to be extended in the next step of the algorithm. It can be determined whether D and L or N is output using one bit. Thus the output is always either triple $\langle 0, D, L \rangle$ or the pair $\langle 1, N \rangle$.

In case the string S had zero length, the first character N in the non-coded part is coded using the pair $\langle 1, N \rangle$

2.6 Transformations

There are many other transformations. Some of them are independent compression methods (e.g. RLE), the other just transform the data to the form that is more suitable for other compression methods (e.g. BWT).

The combination of transformations BWT + MTF + RLE is the most commonly used for compression. Individual transformations are described in more detail.

2.6.1 Burrows-Wheeler Transformation

In a BWT [29] step we transform the input stream into a "better" output stream. The "better" stream means achieving some better final compression ratio. Obviously the transform should be reversible, otherwise we could lose some information. Specifically we achieve a partial grouping of the same input characters.

A	A	L	A	B	A	**M**
A	B	A	M	A	A	**L**
A	**L**	**A**	**B**	**A**	**M**	**A**
A	M	A	A	L	A	**B**
B	A	M	A	A	L	**A**
L	A	B	A	M	A	**A**
M	A	A	L	A	B	**A**

Table 2.1: Burrows-Wheeler transformation for input $S = "ALABAMA"$. The output: $L = "MLABAAA"$ and $I = 2$

BWT takes as input a string S of N characters $S[0], ..., S[N-1]$ selected from an ordered alphabet X of characters. To illustrate the technique, we also give a running example, using the string $S = "ALABAMA", N = 7$, and the alphabet $X = "A", "B", "L", "M"$. BWT consists of two steps: sort rotations and find last characters in rotations.

Sort rotations: We form a conceptual $N \times N$ matrix M whose elements are characters, and whose rows are the rotations (cyclic shifts) of S, sorted in lexicographical order. At least one of the rows of M contains the original string S. Let I be the index of the first such row, numbering from zero. In our example, the index $I = 2$ and the matrix M is in Table 2.1.

Find last characters in rotations: Let the string L be the last column of M , with characters $L[0], ..., L[N-1]$ (equal to $M[0, N-1], ..., M[N-1, N-1]$). The output of the transformation is the pair $\langle L, I \rangle$. In our example, $L = "MLABAAA"$ and $I = 2$.

Usually, not the whole matrix is used in practice, but we need only a few arrays. Individual algorithms for sorting rotations differ in their time and space complexity. One of these algorithms is for example [92].

2.6.2 Move to Front

BWT output string usually contains long sequences of identical characters and thus BWT is most commonly followed by Move to Front (MTF) trans-

formation [28], which translates every character to an integer denoting how many different preceding characters followed since its last occurrence in the string.

Algorithm works in the following way. Suppose a numbered list of alphabet elements, then MTF reads these input elements and writes their list order. As soon as the element is processed it is moved up to the front of the list. This way, a string of characters containing long sequences of zeroes and also other identical characters are obtained. An advanced approach using splay trees instead of ordered lists is desribed in [56].

Example 2.12 *MTF transformation*

```
input: c c c a b b b b a b
starting alphabet: a b c

final alphabet: b a c
output: 2 0 0 1 2 0 0 0 1 1
```

2.6.3 Run Length Encoding

To the result string of MTF we usually apply Run Length Encoding (RLE) transformation, which replaces long sequences of identical characters with single symbol. The replacement takes place only if the sequence is longer than some minimal threshold. It is usually worthless to replace less than three identical characters. This method is very often reduced to replace only sequences of zeroes, so sequences of other identical characters remain unchanged. The result is usually coded using Huffman or arithmetic coding.

The replacement of sequence of zeroes (with minimal length of two) with special symbol Nx, where x is length of the sequence, is introduced in the following example.

Example 2.13 *RLE transformation*

```
input: 0 0 0 0 0 5 2 2 0 0 1 0 0 0 2
output: N5 5 2 2 N2 1 N3 2
```

2.7 Coding of Integers

We have used Elias codes [40] alpha, beta, gamma and delta in our work. There is also omega code, which we have not used.

Definition 2.14 (alpha code):
Let x be an integer greater or equal to one. Then $\alpha(x)$ is defined by the following formula.

$$\alpha(1) = 1$$

$$\alpha(x) = 0 \cdot \alpha(x-1) \text{ for } x > 1$$

Definition 2.15 (beta code):
Let x be an integer greater or equal to one. Then $\beta(x)$ is defined by the following formula. If β is supplemented with zeroes from the left so that it has length n, we get n-bit binary code. If the first bit from β code is omitted, we get β'.

$$\beta(1) = 1$$

$$\beta(2x) = \beta(x) \cdot 0 \text{ for } x > 1$$

$$\beta(2x+1) = \beta(x) \cdot 1 \text{ for } x > 1$$

Definition 2.16 (gamma code):
Let x be an integer greater or equal to one. Then $\gamma(x)$ is permutation of $\gamma'(x)$, in which every bit from code $\alpha(|\beta(x)|)$ is followed by bit from $\beta'(x)$ code.

$$\gamma'(x) = \alpha(|\beta(x)|) \cdot \beta'(x)$$

Definition 2.17 (delta code):
Let x be an integer greater or equal to one. Then $\delta(x)$ is defined by the following formula.

$$\delta(x) = \gamma(|\beta(x)|) \cdot \beta'(x)$$

2.8 Word-based Methods

The word-based methods [73] require to divide the input document into a stream of words and non-words. The words are usually defined as longest alphanumeric strings in a text, while the non-words are the remaining fragments. This definition of words and non-words implies that one can assume that the elements of the two groups are alternated regularly. In practice the words length is often limited by some constant value. Longer words are broken up and the resulting parts are interleaved with a special empty word of an

opposite type (word versus non-word). For instance, if a long word is divided into two parts, then the parts are interleaved with an empty non-word.

Next, another heuristic called the spaceless model approach [76] can be also used. The word is often followed by a special non-word space. So we can skip over the space without any encoding. If the word is not followed with the non-word space, then the coder inserts special symbol for "Missing space". The right decompression is guaranteed, one must only ensure that two successive words are interleaved with the space in a decoder.

Extensive and closely described overview of word-based methods can be found in [97]. This overview was also a primary inspiration for this subchapter. There are word-based variants of all main groups of compression algorithms, mostly in a large number of implementations with various enhancements, for example: word-based Huffman encoding [51, 87], word-based LZW [52, 38], word-based BWT [55, 39], word-based PPM [73, 24].

In the further part of the thesis, we use the term word model, which differs from the one described in this chapter. Contrary to the usual division to words and non-words, we distinguish five types of words (lower, upper, mixed, special, and numeral), see def. 4.5.

Chapter 3

Related Works

Some of general compression methods mentioned in chapters 2.4 to 2.8 can be considered as related works. But real related work to this thesis is only one [110], mentioned in section 3.1.

However, in this section we will mention works related to particular subjects dealt with this work. There are related works: language dependent methods, compression of small text files, compression of XML and compression of set of strings.

3.1 Syllable-based Compression

There are two kinds of source units being used for text compression: characters (letters) and words. We suppose that there is yet another kind of source units: syllables. In some languages the words are naturally divided into syllables (e.g. Czech, Russian, German) while in the others it is harder to recognize them (e.g. English). It is therefore reasonable to expect that syllable compression would be more successful for the languages where syllables are naturally used than for the others.

Term syllable-based compression was firstly used in the article [110], however in a slightly different meaning that is used in this thesis. The authors used genetic algorithms for finding syllables. They denoted group of letters containing one vowel as syllable, if this group was frequent in Turkish. These syllables were replaced by one symbol in textual document. It was actually compression over union of letters and syllables. The authors used static Huffman coding.

3.2 Language Dependent Methods

There is number of works that improve text compression based on the regularities that occur in English. Extensive and closely described overview of word-based methods can be found in [97]. This overview was the primary inspiration for this chapter, similarly to the previous chapter.

One of the examples of these works are articles, where authors improved compression ratio in Burrows-Wheeler transformation, when they used following order of letters in the alphabet for lexicographic sorting:

`AEIOUBCDGFHRLSMNPQJKTWVXYZ` described in [31],

`SNLMGQZBPCFMRHAOUIYXVDKTJE` described in [23].

This preprocessing can increases lengths of null sequences produced in MTF phase and increases effect of RLE.

Another group of works deal with replacing common clusters of letters (n-grams) in English text with one symbol [104]. A static initialization of PPM method is used in Shkarin's program Durilca [94].

Nevertheless, these methods are usually unusable for files in other than English languages and they usually worsen the compression. The solution might inhere in separately determining statistics for every language and passing the language of document into compression program. Translations between encodings would be necessary for languages that use diacritics. This approach is not common since majority of authors is from English speaking countries. An exception is the work desribed in Chapter 3.2.1.

3.2.1 Words Replacing Methods

Two preprocessing methods are described in the work [97]. Word Replacing Transformation (WRT) is an English text preprocessing algorithm. It exploits the redundancy in English texts by using a fixed dictionary of English words. WRT transforms an input text by matching words in this text with words in the dictionary and replacing such words with a pointer into the dictionary. The pointer is represented by the codeword, which is shorter and easier to compress using universal compression algorithms than the original word. WRT combines several well-known preprocessing techniques: the static word replacement, the capital conversion, the n-gram replacement, and the EOL coding.

The extension of WRT called Two-level Word Replacing Transformation (TWRT) uses several dictionaries for different languages (English, German, Polish, Russian, French) and their various encodings (ASCII, iso, cp).

Moreover, TWRT automatically recognizes multilingual text files. TWRT also includes the fixed-length record aligned data preprocessing and the DNA sequence preprocessing, which were not implemented in WRT.

The biggest advantage of TWRT is, that it can be used in combinations with other universal lossless compression methods, including the powerful ones: PAQ [67] or PPMonstr [96].

3.3 Small Text Files Compression

Compression of small text files is quite untraditional subject of research. Common compression methods do not specialize on it and they achieve bad results. Statistical methods seem to be a suitable solution for very small files (cca 50B), while the model is trained on natural texts. A method based on a static version of PPM is introduced in [83]. In [62] a statistic model based on sequence of variously pruned suffix trees is used for compression. Description of both methods is almost literally adopted from respective articles.

3.3.1 Compression of Text for mobile phones

The paper [83] details a method for lossless compression of short files larger than 50 Bytes. The method uses arithmetic coding and context modeling with a low-complexity data model. A data model that takes 32 KB of RAM already cuts the data size in half (compression ratio 4 bits per character). The compression scheme just takes a few pages of source code, is scalable in memory size, and may be useful in sensor or cellular networks to spare bandwidth. As the authors demonstrated the method allows for battery savings when applied to mobile phones.

The authors discussed the statistical coding technique prediction by partial matching (PPM), which employs a statistical context model and an arithmetic coder. The authors intended to develop and to apply a low-complexity version of PPM that demands for low memory, is conceptually very simple, and can compress very short data sequences starting from 50 Bytes. The authors have selected the PPMC version of PPM as a basis for their research investigations because it gives good compression while it is conceptually simple. The authors developed a library for the design of low-complexity context models and applied the library to design and evaluate a context model for embedded systems. This model already achieves reasonable compression results for short messages with 32 KB RAM when appropriate statistical data is preloaded. For construction of a static model the authors used the text file book2 from [26] with the model sizes 32, 64, 128, and 256 KB.

3.3.2 Compression Using Optimal Tree Machines

The paper [62] discusses a lossless data compression method that uses fixed Tree Machines to encode data. A general Tree Machine is a special suffix trie where a context string is associated with each node, which is the suffix of the string of any of its descendants. The root node corresponds to the empty string. Each node contains a list of characters that have occurred in the context of that node with their occurrence counts.

The idea is to create a sequence of Tree Machines and a robust escape method aimed at preventing expansion of the encoded string for data whose statistics deviate from those represented by the machines. The resulting algorithm is shown to have superior compression of short files compared to other methods.

Optimal Tree Machine is created by pruning some nodes from TM. For every context in TM, whole path from the root to the node representing this context is traversed. If the special value called Local Order Estimation [84] is for some node better than this value for the whole following context, then the contexts is cut from the tree in this place.

We gradually obtain sequence of trees with different level of pruning. A static model consisting of sequence of trees sorted from the most pruned to the least pruned is used for compression. During compression phase, if it is not possible to encode the character using some tree, an escape sequence as an transition to a less pruned tree is used.

3.4 XML Compression

Interesting thirty page comparison of compression methods for XML is presented in article [77]. However, we describe the methods more briefly.

There are many algorithms which compress XML data. One of the first available was XMill [66]. Many other successors are based on similar principles, i.e. XMLPPM [32]. Some algorithms also add new features: XGrind [108] and XPress [71] are able to query the compressed data structure, but this ability worses compression ratio.

Articles that use document scheme description languages to achieve better compression ratio are also interesting. For example, DTD is used in [102], or Relax NG schema in [64]. One can considerably reduce the set of possible tags and attributes that can follow in a particular context. However, these methods do not achieve any substantial improvement for documents with mainly textual character and cannot be used neither to compress non-well-formed documents, nor to compress non-valid documents

A serious problem of all methods mentioned above is that they require well-formed XML document as an input, whilst much of the practical data contains certain amount of errors. Further, these methods are mainly aimed to XML documents with high ratio of tags. These methods are not usually quite successful in case of documents consisting mainly of text.

3.5 Set of Strings Compression

One of the subjects of this thesis is also a problem how to encode set of strings. This is useful for coding dictionaries of various kinds of source units, in case of syllable-based compression with various compression methods, since the dictionary must be passed between the coder and the decoder. We will address this in more detail in Chapter 7.

The word-based compression methods known to the author do not care about the compression of the used dictionary, as it is for large compressed bodies in comparison to the encoded message quite small. Simple methods used in word-based methods for the compression of the used dictionary are desribed in Chapter 7.4.1.

The issue of efficient compression of a set of strings is solved in the papers mentioned below. Complex methods like applying combination of prefix and suffix compression (like in [25] or [35]) seems to be useful especially for dictionaries with long elements (what is not the case of syllables, the average length of a syllable is 3 characters). Other compression methods (like the one proposed by Maly in [68]) are based on the knowledge of limited alphabet that cannot be applied when compressing also non alphabetical elements.

3.5.1 Main Memory Methods

There are several methods for storing a set of strings in the main memory. These methods are usually focused on reduction of time used for searching a string in the set. The secondary goal of these methods is reduction used memory for storing the whole set. These methods can be used as a part of applications such as spelling checkers, word games, and database indexes.

Minimal Acyclic Deterministic Finite Automata

Minimal acyclic deterministic finite automata (ADFAs) [37] can be used as a compact representation of finite string sets with fast access time. Creating them with traditional algorithms of deterministic finite automata minimization is a resource hog when a large collection of strings is involved. The

authors popularized an efficient but little known algorithm for creating minimal ADFAs recognizing a finite language. The algorithm is presented for three variants of ADFAs, its minor improvements are discussed, and minimal ADFAs are compared to competitive data structures.

LZ Trie and Dictionary Compression

The idea of LZ trie compression [85] is following. Merging equivalent states in a trie to produce a deterministic finite automata with multiple initial states effectively reduces redundancy in a trie by substituting all identical repeated branches with only one. However, this may still leave a number of repeated identical subsections of a trie. The authors proposed to replace each repeated subsection with pointers to its first occurrence. If a pointer is smaller in size than the replaced part, then overall size is reduced according to LZ77. Hence the name LZ trie.

Burst Trie

There is proposed a new data structure, the burst trie [49], that has significant advantages: it requires no more memory than a binary tree; it is as fast as a trie; and, while not as fast as a hash table, a burst trie maintains the strings in sorted or near-sorted order. The authors described burst tries and explored the parameters that govern their performance. The authors experimentally determined good choices of parameters, and compared burst tries to other structures, with a variety of data sets. These experiments shown that the burst trie is particularly effective for the skewed frequency distributions common in text collections, and dramatically outperforms all other data structures for the task of managing strings while maintaining sort order.

Chapter 4

Syllable-Based Compression

This chapter is mostly based on the author's Master thesis [1] and author's publications [2, 4, 14].

Common universal compression methods decompose the text into single characters or n-grams of these characters, and use them to compress the file. For text compression, word-based methods are often used. These methods decompose the document into a sequences of words (sequences of alphanumeric characters) and non-words (sequences of other characters), and call these new elements the alphabet. This new alphabet is used to compress the whole document.

Syllable-based compression has, unlike word-based compression, many specifics, see 4.1. Informal definition of the notion of "syllable" is given in Chapter 4.2. Decomposition of text into words is easy and unambiguous; dividing words into syllables, however, is often ambiguous and brings many complications, described in Chapter 4.3. Formal definition of "syllable" can be found in Chapter 4.4. In Chapter 4.5, we have defined Czech and English languages using our formalism. In Chapter 4.6, we have proposed four universal algorithms for the decomposition of words into syllables.

4.1 Languages and Syllables

Understanding the structure of coded message can be very helpful in designing new compression method. In our case the coded message is a text in natural language. Its structure is determined by the characteristic of the particular language. One linguistic aspect is the morphology. There are languages like English having simple morphology where only a few word forms can be derived from one stem. There are also languages with rich morphology

(like Czech, German, Turkish, or Russian) where from a single stem several tens of word forms can be derived.

Languages with richer morphology tend to creating new words and word-forms by concatenating the root of the word with one or several prefixes or suffixes. On the other hand in languages like English the same effect is achieved by accumulating words. In the first category of languages we may find (thanks to their agglutinative nature) many rather long words composed of higher number of syllables. Such words are not very common in English. We can expect, that syllable-based compression will give better results on the first group of languages.

We will demonstrate it on some examples from Czech and English. The English verb *take* has only following 5 forms: *take*, *takes*, *taking*, *took*, *taken*. Czech verb *vzít*, which corresponds to the English verb *take*, has following 24 forms: *vzít*, *vzíti*, *vezmu*, *vezmeš*, *vezme*, *vezmeme*, *vezmete*, *vezmou*, *vzal*, *vzala*, *vzalo*, *vzali*, *vzaly*, *vezmi*, *vezměme*, *vezměte*, *vzat*, *vzata*, *vzato*, *vzati*, *vzaty*, *vzav*, *vzavši*, *vzavše*. Another difference is in creation of words with similar meaning and negations. In English there are used combinations of more words for getting different meaning, for example *get on*, *get off*. The negation in English is formed by combination with word *not*, for example *not get on*. In Czech prefixes and suffixes are used instead. To the English forms *get on*, *get off*, *not get on* correspond the Czech ones *nastoupit*, *vystoupit*, *nenastoupit*. In Czech we can create from the verb *skočit* (*jump* in English) using prefixes 10 following similar verbs: *přeskočit*, *nadskočit*, *podskočit*, *poskočit*, *odskočit*, *rozskočit*, *naskočit*, *vskočit*, *uskočit*, *vyskočit*. For each of these verbs we can create their antonyms by using prefix *ne*: *neskočit*, *nepřeskočit*, *nenadskočit*, *nepodskočit*, *neposkočit*, *neodskočit*, *nerozskočit*, *nenaskočit*, *nevskočit*, *neuskočit*, *nevyskočit*. For each of these 22 verbs there exist about 23 different grammatical forms. So from this one word *skočit* we can derive over 500 similar words, but these words are composed from only a few tens of syllables.

4.2 Syllables

What is actually a syllable? Usually it is presented as a phonetic phenomenon. American Heritage Dictionary [106] gives us the following definition: 'A unit of spoken language consisting of a single uninterrupted sound formed by a vowel, diphthong, or syllabic consonant alone, or by any of these sounds preceded, followed, or surrounded by one or more consonants.'

As the decomposition of words into syllables is used in data compression, it is not necessary to decompose words into syllables always grammatically

correctly. It is sufficient if the decomposition produces groups of letters that occur quite frequently. We therefore use the following simplified definition that is not equivalent with the above mentioned definition. 'Syllable is a sequence of sounds, which contains exactly one maximal subsequence of vowels.' This definition implies that the number of syllables in a word is equal to the number of maximal sequences of vowels in the same word. For example, the word *famous* contains two maximal sequences of vowels: *a* and *ou*, so this word is created from two syllables: *fa* and *mous*. Word *pour* contains only one maximal sequence of vowels *ou*, so the whole word is composed from one and only syllable.

Decomposition of words into syllables is used for example for text formatting when we want to split word exceeding an end of line. Disadvantage of this way is that we cannot decompose all words and some words must be left unsplitted.

One of the reasons why we selected syllables is that documents contain less unique syllables than unique words. Exemplary Czech document (Karel Čapek: Hordubal [41]) with the size of 195 KB contains 33,135 words where 8,071 of them are distinct and 61,259 syllables where 3,187 of them are distinct. English translation of the Bible [113] with the size of 4MB contains 767,857 words where 13,455 of them are distinct and 1,073,882 syllables where 5,604 of them are distinct. More results you can see in Chapter 7.5.3, Table 7.5 or in Master thesis [1], Chapter 2.5.1, Table 1.

4.3 Problems in Decomposition of Words into Syllables

Decomposition of words into syllables is not always unique. To choose the correct decomposition of a word, we must often know the origin of that word. Some problems will be demonstrated on selected Czech words. We supposed that for compression it is sufficient to use some approximation of correct decomposition of words into syllables. We supposed that this approximation should only have a small negative effect on reached compression ratio.

The word *Ostrava* is an example of non-uniqueness of decomposition of words into syllables: its correct decompositions are *Os-tra-va* and *Ost-ra-va*. Generally sequence of letters *st* is often a source of ambiguity of decomposition of Czech words to syllables.

An example of a variant decomposition of similar sequences of letters, which is caused by origin of words, is words *obletí* and *obřečí*, that have first two letters same. Word *obletí* (will fly around) was created by adding prefix

ob to the word *letí* (flies). Word *obrečí* (will cry over) was created by adding prefix *o* to the word *brečí* (cries). So the word *obletí* is decomposed into *ob-le-tí*, word *obrečí* is decomposed into *o-bre-čí*. A big group of problems is brought by words of foreign origin and their adapted forms.

Sometimes it can be quite difficult to recognize the real number of syllables in a given word. Although the word *neuron* is a prefix of the word *neuronit*, these words have different decompositions into syllables. Word *neuron* is decomposed to *neu-ron*, word *neuronit* is decomposed to *ne-u-ro-nit*. In the first case the sequence of letters *neu* is composed by one syllable, in the second case it is composed from two syllables.

In German we face to the following problems: The lower-case letter 'ß' becomes, when turned into upper-case, two following letters 'SS'. German uses for a description of a single vowel one to four characters what complicates recognition of the syllable boundaries.

Full correctness of decomposition of words into syllables can be reached only at the price of very high effort. For the use in compression it is not important whether the decomposition is absolutely correct, but whether the produced groups of letters are frequent enough.

4.4 Definition of Syllable

This chapter formalizes the process of splitting the input document into the sequence of syllables. We suppose that the document is written in some natural language. Our formalization has to be strong enough to be able to process also a document that can be in a random binary form. We will formally split this random binary file into the syllables, but these syllables will not be from any natural language.

We have to determine which symbols are upper-case, lower-case letters, digits and other special characters for each natural language. We need to determine a specific context role for each symbol. This context is affected by the symbol on the left and on the right from the determined symbol.

Role of the letter can be vowel, consonant, vowel followed by consonant or consonant followed by vowel. We define all these properties in the language definition (see def. 4.7). Our language definition can be an approximation of the natural language only.

Our syllable definition (see def. 4.8) uses only the information gathered from the language definition. Any syllable in the natural language fulfils this definition. On the contrary this definition is fulfilled by many strings that are not a syllable in the given natural language. This is because we have to be able to decompose a random binary file too.

There are two groups of the algorithms splitting the words to syllables. Universal algorithms use the information from the language definition and do not use any other properties of the given natural language. Specific algorithms use the properties of the given natural language.

Definition 4.1 (Types of Characters):

Let Σ be an alphabet (see def. 2.1).

Let $\Sigma_{\text{Letter}} \subseteq \Sigma$ be *set of letters*, then $\Sigma_{\text{NonLetter}} = \Sigma \backslash \Sigma_{\text{Letter}}$ is called *set of non-letters*.

Let $\Sigma_{\text{Digit}} \subseteq \Sigma_{\text{NonLetter}}$ be *set of digits*, then $\Sigma_{\text{Special}} = \Sigma_{\text{NonLetter}} \backslash \Sigma_{\text{Digit}}$ is called *set of special characters*.

Definition 4.2 (Types of Letters):

Let Σ be an alphabet, Σ_{Letter} be a set of letters, and $\Sigma_{\text{Lower}} \subseteq \Sigma_{\text{Letter}}$ be *set of lower-case letters*. Then $\Sigma_{\text{Upper}} = \Sigma_{\text{Letter}} \backslash \Sigma_{\text{Lower}}$ is called *set of upper-case letters*.

If there exists a bijection $\psi : \Sigma_{\text{Lower}} \rightarrow \Sigma_{\text{Upper}}$, then Σ_{Letter} is called *correct set of letters*.

The set of letters is correct for most of natural languages, but there are exceptions, for example German. The lower-case letter 'ß' becomes 'SS', when turned into upper-case.

Further in this thesis, we will be working with correct sets of letters only, so as not to complicate the theory. In practice, designing algorithms for non-correct sets of letters is discouraged, since it only serves to increase their time complexity without practically any improvement of correctness.

If we want to use our syllable theory for the given language, we have to turn its set of letters artificially into a correct set, which can slightly worsen the compression ratio, but must not affect the decompression process.

A letter can have different roles, according to context (see def. 4.3). The roles of letters correspond with *sounds* during pronunciation of letters. We recognize two types of sounds: *vowel* and *consonant*.

A pair of sounds, where the first sound is a vowel followed by the second sound, which is a consonant will be of type $\langle vowel, consonant \rangle$.

A pair of sounds, where the first sound is a consonant followed by the second sound, which is a vowel will be of type $\langle consonant, vowel \rangle$.

Let *LetterRoles* be a set $\{vowel, consonant, \langle vowel, consonant \rangle, \langle consonant, vowel \rangle\}$

Definition 4.3 (Role of Letter in Given Context):

Let $\nu \in \Sigma_{\text{Letter}}$, $\omega \in \mathit{LetterRoles}$, $\mu, \pi \in (\Sigma_{\text{Letter}} \cup \{\lambda\})$.

Let $\phi : (\Sigma \cup \{\lambda\}) \times \Sigma_{\text{Letter}} \times (\Sigma \cup \{\lambda\}) \rightarrow \mathit{LetterRoles}$ be a function.

If $\phi(\mu, \nu, \pi) = \omega$ then ν *has the role* of ω in the context μ, π.

The letter μ is called *left context* of ν, the letter π is called *right context* of ν.

Notation:

Let $\omega \in \mathit{LetterRoles}$. Since expressions like *letter* ν *has the role* ω in the context μ, π are too formal and long, we use sometimes the simplified notation *letter* ν *has the role* ω for better understanding.

We say that *function* ϕ *determines if the given letter is a vowel or a consonant.*

It is probable that there exist languages where we do not need to know the context μ, π to decide if letter ν has the role *vowel* or *consonant*. In both Czech and English the use of context is necessary.

In Czech the letters r and l can be used as vowels or as consonants according their context. If μ or π are consonants, then $\nu = r$ (respectively $\nu = l$) has the role *vowel* (examples: *mlčet*, *vrtat*), in the opposite case it has the role *consonant* (examples: *mluvit*, *vrátit*).

In English the letter y in context of two vowels has the role *consonant* (example: *buying*).

In English words of type *trying* has y the role $\langle \mathit{vowel}, \mathit{consonant} \rangle$, so y is pronounced as a vowel sound followed by a consonant sound.

In Russian some letters (for example *E*) can have role $\langle \mathit{consonant}, \mathit{vowel} \rangle$.

We suppose that one letter on the left side and one letter on the right side is a sufficient context for Czech and for most of English words. This context is also sufficient for other languages.

Definition 4.4 (Blocks of Vowels and Consonants):

Let $\alpha = \alpha_1 \dots \alpha_n$, $\alpha_i \in \Sigma_{\text{Letter}}$, and $\beta = \beta_1 \dots \beta_m$, $\beta_i \in \Sigma_{\text{Letter}}$. Let β be a substring of α. We determine for each α_i its role in the context α_{i-1}, α_{i+1} according the definition 4.3. We use the context λ, α_2 for α_1, we use the context $\alpha_n - 1, \lambda$ for α_n. Because β is a substring of α, we know the role of each β_i.

If β_i has the role *consonant* in its context for $i = 1, \dots, m$, then β is called *block of consonants.*

If β_1 has the role *vowel* or $\langle \mathit{consonant}, \mathit{vowel} \rangle$ & β_m has the role *vowel* or $\langle \mathit{vowel}, \mathit{consonant} \rangle$ & β_i has the role *vowel* for $i = 2, \dots, m-1$, then β is called *block of vowels.*

Definition 4.5 (Words):

Let Σ be an alphabet, Σ_{Lower} be a set of lower-case letters, Σ_{Upper} be a set of upper-case letters, Σ_{Digit} be a set of digits, and Σ_{Special} be a set of special characters. Let $\alpha = \alpha_1, \ldots, \alpha_n$, $\alpha_i \in \Sigma$. If one of the following cases is valid, then α is called a *word over alphabet* Σ.

If $\alpha_i \in \Sigma_{\text{Lower}}$ for $i = 1, \ldots, n$, word α is called *lower-case.*
If $\alpha_i \in \Sigma_{\text{Upper}}$ for $i = 1, \ldots, n$, word α is called *upper-case.*
If $|\alpha| > 1$ & $\alpha_1 \in \Sigma_{\text{Upper}}$ & $\alpha_i \in \Sigma_{\text{Lower}}$ for $i = 2, \ldots, n$, word α is called *mixed.*
If $\alpha_i \in \Sigma_{\text{Digit}}$ for $i = 1, \ldots, n$, word α is called *numeric.*
If $\alpha_i \in \Sigma_{\text{Special}}$ for $i = 1, \ldots, n$, word α is called *other.*

Numeric words and other words are called together *words from non-letters.* Lower-case, upper-case, and mixed words are called *words from letters.*

There are five types of words in total, three being composed of letters, one of numbers and the last one of special characters, which are neither letters nor numbers. In word-based compression, words are usually [114] divided into words (composed of letters and numbers) and non-words (composed of other characters). Non-words correspond with our last group (composed of neither letters nor numbers), words then correspond with the other four groups.

This division of alphanumeric strings is proving to be very efficient in the compression of a set of strings, which we need to encode the set of used words or syllables at any given moment. Further, it is useful in statistical compression methods, since they are able to follow the model of a natural sentence more precisely.

Definition 4.6 (Decomposition into Words):

Let Σ be an alphabet and $\alpha = \alpha_1 \ldots \alpha_n, \alpha_i \in \Sigma$. Let $\beta_1, ..., \beta_m$ are words over alphabet Σ. Let $\gamma_i = \beta_i \cdot \beta_{i+1}$ and γ_i is not a word over alphabet Σ for $i = 1, \ldots, m-1$. Let $\alpha = \beta_1 \cdot ... \cdot \beta_m$. Then $\langle \beta_1, ..., \beta_m \rangle$ is called *decomposition of the string* α *into words.*

It follows from definitions 4.5 and 4.6 that pro each string its decomposition into words exists.

It follows from definitions 4.5 and 4.6 that decomposition of strings into words is almost unique. There is an exception when at least one lower-case letter follows after two ore more upper-case letters (example CDs), but this case is very rare in natural languages.

There exist two algorithms of decomposition of strings into words. Both algorithms differ only in solving that exception. The algorithm A_1 (see alg. 1) creates an upper-case word and a mixed word from this string (example C

Algorithm 1 Decomposition into words, algorithm A_1

```
1: input message M = α_i...α_n, α_i ∈ Σ
2: output decomposition M into words
3: S = 1 /*begin of word*/
4: for i = 2, ..., n do
5:    if α_i ∈ Σ_Digit then
6:       if α_{i-1} ∉ Σ_Digit then
7:          output(ω = α_S...α_{i-1})
8:          S = i
9:       end if
10:   else if α_i ∈ Σ_Special then
11:      if α_{i-1} ∉ Σ_Special then
12:         output(ω = α_S...α_{i-1})
13:         S = i
14:      end if
15:   else if α_i ∈ Σ_Upper then
16:      if α_{i-1} ∉ Σ_Upper then
17:         output(ω = α_S...α_{i-1})
18:         S = i
19:      end if
20:   else if α_i ∈ Σ_Lower then
21:      if α_{i-1} ∉ Σ_Lower and (α_{i-1} ∉ Σ_Upper or S ≠ i - 1) then
22:         output(ω = α_S...α_{i-1})
23:         S = i
24:      end if
25:   end if
26: end for
27: output(ω = α_S...α_n)
```

and Ds). The algorithm A_2 creates an upper-case word and a lower-case word from this string (example CD and s).

Definition 4.7 (Language):
A *language* L is an ordered 6-tuple $(\Sigma, \Sigma_{\text{Letter}}, \Sigma_{\text{Digit}}, \Sigma_{\text{Lower}}, \phi, A)$, where

- Σ is an alphabet.
- Σ_{Letter} is a correct set of letters. (see def. 4.2)
- Σ_{Digit} is a set of digits.
- Σ_{Lower} is a set of lower-case letters.
- $\phi : (\Sigma \cup \{\lambda\}) \times \Sigma_{\text{Letter}} \times (\Sigma \cup \{\lambda\}) \rightarrow \mathit{LetterRoles}$ is a function which according definition 4.3 specifies whether letter ν in context μ, $\pi \in (\Sigma \cup \{\lambda\})$ has the role of a vowel, consonant, vowel followed by consonant or consonant followed by vowel.
- A is an algorithm which for each string $\alpha = \alpha_1 \ldots \alpha_n$, $\alpha_i \in \Sigma$ finds some decomposition of α into words.

Definition 4.8 (Syllable):
Let $L = (\Sigma, \Sigma_{\text{Letter}}, \Sigma_{\text{Digit}}, \Sigma_{\text{Lower}}, \phi, A)$ be a language.

- Let α, γ are λ or blocks of consonant, β is λ or block of vowels of maximal length 3. Let $\omega = \alpha \cdot \beta \cdot \gamma$. If ω is a word over alphabet Σ, then ω is *syllable of the language L*.
- Let ω be a word from non-letters, then ω is syllable of the language L.

It follows from the definition 4.8 that each syllable is also a word. So we will recognize (according Def. 4.5) five types of syllables: other syllables, numeric syllables, lower-case syllables, upper-case syllables, and mixed syllables.

For example, the string $xxAxx$ is not word (according Def. 4.5), because it contains upper-case letter between two lower-case letters. Therefore it cannot be (according Def. 4.8) syllable. This string must be decomposed into words xx a Axx (according Def. 4.5) and then it can be decomposed into syllables.

Definition 4.9 (Decomposable Words):
Let $L = (\Sigma, \Sigma_{\text{Letter}}, \Sigma_{\text{Digit}}, \Sigma_{\text{Lower}}, \phi, A)$ be a language and α be a word from letters (see def. 4.5).

If α contains at least one block of vowels, then α is called *word decomposable into syllables*, else α is called *non-syllable word*.

Definition 4.10 (Decomposition into Syllables):
Let $L = (\Sigma, \Sigma_{\text{Letter}}, \Sigma_{\text{Digit}}, \Sigma_{\text{Lower}}, \phi, A)$ be a language. Let $\alpha = \alpha_1 \ldots \alpha_n$, $\alpha_i \in \Sigma_{\text{Letter}}$ be a word decomposable into syllables. Let $\beta_1, ..., \beta_m$ be syllables of the language L. Let $\alpha = \beta_1 \cdot ... \cdot \beta_m$ and $\gamma_i = \beta_i \cdot \beta_{i+1}$. If γ_i is not a syllable of the language L for $i = 1, \ldots, m-1$. Then $\langle \beta_1, ..., \beta_m \rangle$ is called *decomposition of word α into syllables.*

Definition 4.11 (Algorithms of Decomposing):
Let $L = (\Sigma, \Sigma_{\text{Letter}}, \Sigma_{\text{Digit}}, \Sigma_{\text{Lower}}, \phi, A)$ be a language. Let P be an algorithm whose input is a message M decomposed into words $\alpha_1, ..., \alpha_n$ over alphabet Σ by the algorithm A. Let for all α_i the following conditions are valid.

- If α_i is a word decomposable into syllables, then the algorithm returns a decomposition of the word α_i into syllables.
- If α_i is a non-syllable word, numeric word, or other word, then the algorithm returns the word α_i.

The algorithm P is called *algorithm of decomposition into syllables for language L.*

Definition 4.12 (Universal and Specific Algorithms):
Let P be an algorithm of decomposition into syllables for a language L_1,

- If P is an algorithm of decomposition into syllables for all languages L, then we say that P is a *universal algorithm of decomposition into syllables.*
- If there exist a language L_2 for which P is not algorithm of decomposition into syllables, then we say that P is *specific algorithm of decomposition into syllables.*

4.5 Examples of Languages

We will consider two examples of languages: English and Czech. Czech language has larger set of letters in comparison with English, because some letters have diacritical marks. The biggest differences between those two languages are in definition of the function ϕ.

Example 4.13

English language can be characterized as $\mathrm{L_{EN}} = (\Sigma, \Sigma_{\mathrm{Letter}}, \Sigma_{\mathrm{Digit}}, \Sigma_{\mathrm{Lower}}, \phi, A)$ where:

- Σ = character set
- $\Sigma_{\mathrm{Letter}} = \{a, \ldots, z, A, \ldots, Z\}$ is a set of *letters*;
- $\Sigma_{\mathrm{Digit}} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ is a set of *digits*;
- $\Sigma_{\mathrm{Lower}} = \{a, \ldots, z\}$ is a set of *lower-case letters*;
- ϕ is defined as:
  - Let $M = \{a, e, i, o, u, y, A, E, I, O, U, Y\}$.
  - $\forall \mu, \pi \in \Sigma \cup \{\lambda\},\ \forall \nu \in M \backslash \{y, Y\} :\ \phi(\mu, \nu, \pi) = \mathit{vowel}$ (a, e, i, o, u, A, E, I, O, U are always recognized as vowels);
  - $\forall \mu, \pi \in \Sigma \cup \{\lambda\},\ \forall \nu \in \Sigma_{\mathrm{Letter}} \setminus M :\ \phi(\mu, \nu, \pi) = \mathit{consonant}$ all characters from $\Sigma_{\mathrm{Letter}} \backslash M$ are always recognized as consonants);
  - $\forall \mu, \pi \in (\Sigma \backslash M) \cup \{\lambda\},\ \forall \nu \in \{y, Y\}\ \phi(\mu, \nu, \pi) = \mathit{vowel}$ (y or Y is recognized as a vowel when surrounded by a consonant or nothing);
  - $\forall \mu \in (\Sigma_{\mathrm{Letter}} \backslash M), \forall \nu \in \{y, Y\}, \forall \pi \in M\ \phi(\mu, \nu, \pi) = \langle \mathit{vowel}, \mathit{consonant} \rangle$ (y or Y is recognized as a vowel followed by a consonant when they are preceded by a consonant and followed by an initial vowel);
  - $\forall \mu \in M, \forall \nu \in \{y, Y\}, \forall \pi \in \Sigma \cup \{\lambda\} :\ \phi(\mu, \nu, \pi) = \mathit{consonant}$ (y or Y is recognized as consonant when preceded by an initial vowel);
  - $\forall \mu \in (\Sigma \setminus \Sigma_{\mathrm{Letter}}) \cup \{\lambda\},\ \forall \nu \in \{y, Y\}, \forall \pi \in M :$ $\phi(\mu, \nu, \pi) = \mathit{consonant}$ (y or Y is recognized as consonant when it is at the beginning of a word and when followed by an initial vowel).
- $A = A_1$ (see alg. 1)

Example 4.14

Czech language can be characterized as $L_{CZ} = (\Sigma, \Sigma_{Letter}, \Sigma_{Digit}, \Sigma_{Lower}, \phi, A)$ where:

- Σ = character set;
- Σ_{Letter} = {a,...,z,A,...,Z,á,č,ď,é,ě,í,ň,ó,ř,š,ť,ú,ů,ý,ž,Á,Č,Ď,É,Ě,Í,Ň,Ó,Ř,Š,Ť,Ú,Ů,Ý,Ž} is a set of *letters*;
- $\Sigma_{Digit} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ is a set of *digits*;
- Σ_{Lower} = {a,...,z,á,č,ď,é,ě,í,ň,ó,ř,š,ť,ú,ů,ý,ž} is a set of *lower-case letters*;
- ϕ is defined as:
  - Let M = {a,á,e,é,ě,i,í,o,ó,u,ú,ů,y,ý,A,Á,E,É,Ě,I,Í,O,Ó,U,Ú,Ů,Y,Ý}.
  - $\forall \mu, \pi \in \Sigma \cup \{\lambda\},\ \forall \nu \in M :\ \phi(\mu, \nu, \pi) = vowel$
    (all initial vowels are always recognized as vowels);
  - $\forall \mu \in \Sigma_{Letter} \setminus M,\ \forall \nu \in \{r, l, R, L\},\ \forall \pi \in (\Sigma \cup \{\lambda\}) \setminus M :$
    $\phi(\mu, \nu, \pi) = vowel$
    (l, r, L, R are recognized as vowels only when preceded by a consonant and followed by nothing or by a consonant);
  - else $\phi(\mu, \nu, \pi) = consonant$
    (in all other cases the letter is recognized as a consonant);
- $A = A_1$ (see alg. 1)

4.6 Algorithms of Decomposition

We describe four universal algorithms of decomposition into syllables (see alg. 2): universal left P_{UL}, universal right P_{UR}, universal middle-left P_{UML}, and universal middle-right P_{UMR}. These four algorithms are called *algorithms of class P_U*. Inputs of these algorithms are message M and language L. These algorithms are composed from two phases. The first one is an initialization common for all algorithms of the class P_U. The second one is different for each algorithm.

- In the initialization phase we decompose the message M into words by algorithm A. Algorithm of class P_U is processing single words.
- Words from non-letters are automatically declared as syllables.

Algorithm 2 Decomposing into syllables

1: **input** langauge $L = (\Sigma, \Sigma_{\text{Letter}}, \Sigma_{\text{Digit}}, \Sigma_{\text{Lower}}, \phi, A)$ and message $M = \alpha_i ... \alpha_n, \alpha_i \in \Sigma$
2: **output** decomposition M into syllables
3: decompose M into words ω_1, ..., ω_m by A
4: **for** $i = 1, ..., n$ **do**
5:    Let $\omega_i = \omega_{i1} ... \omega_{ik}$
6:    **if** ω_i is word from non-letters **then**
7:       output(ω_i), **continue**
8:    **end if**
9:    **for** $j = 1, ..., k$ **do** determine role ω_{ij} by function ϕ **endfor**
10:    find maximal blocks of vowels β_{i1}, ..., β_{ip} in word ω_i
11:    find maximal blocks of consonants γ_{i1}, ..., γ_{ir} in word ω_i
12:    **if** $p < 2$ **then**
13:       output(ω_i), **continue**
14:    **end if**
15:    **if** γ_{ir} is a suffix of ω_i **then**
16:       $\beta_{ip} = \beta_{ip} \cdot \gamma_{ir}$
17:       remove γ_{ir} from list of maximal blocks of consonants, $r = r - 1$
18:    **end if**
19:    **if** γ_{i1} is a prefix of ω_i **then**
20:       $\beta_{i1} = \gamma_{i1} \cdot \beta_{i1}$
21:       remove γ_{i1} from list of maximal blocks of consonants, $r = r - 1$
22:    **end if**
23:    **for** $j = 1, ..., r$ **do**
24:       Let $\gamma_{ij} = \gamma_{ij_1} ... \gamma_{ij_h}$ /*β_{ij} is the first block of vowels before γ_{ij} in ω_i and $\beta_{i(j+1)}$ is the first block of vowels after γ_{ij} in ω_i*/
25:       **if** algorithm is P_{UL} **then**
26:          $\beta_{ij} = \beta_{ij} \cdot \gamma_{ij}$
27:       **else if** algorithm is P_{UR} **then**
28:          $\beta_{i(j+1)} = \gamma_{ij} \cdot \beta_{i(j+1)}$
29:       **else if** algorithm is P_{UMR} **or** (algorithm is P_{UML} **and** $h = 1$) **then**
30:          $\beta_{ij} = \beta_{ij} \cdot \gamma_{ij_1} ... \gamma_{ij_{\lfloor h/2 \rfloor}}$
31:          $\beta_{i(j+1)} = \gamma_{ij_{\lfloor h/2 \rfloor + 1}} ... \gamma_{ij_h} \cdot \beta_{i(j+1)}$
32:       **else if** algorithm is P_{UML} **and** $h \neq 1$ **then**
33:          $\beta_{ij} = \beta_{ij} \cdot \gamma_{ij_1} ... \gamma_{ij_{\lceil h/2 \rceil}}$
34:          $\beta_{i(j+1)} = \gamma_{ij_{\lceil h/2 \rceil + 1}} ... \gamma_{ij_h} \cdot \beta_{i(j+1)}$
35:       **end if**
36:    **end for**
37:    **for** $j = 1, ..., p$ **do** output(β_{ij}) **endfor**
38: **end for**

- For each word ω_i from letters and for each letter ω_{ij} in ω_i the function ϕ decides if ω_{ij} has the role of consonant or vowel.
- Maximal blocks (blocks that cannot be extended) of vowels β_{ij} and maximal blocks of consonats γ_{ij} are found afterwards. Blocks of vowels longer than three are usually not in natural languages, so maximal length of block of vowels is set to 3. For each block of vowels we must keep in memory its begin and end.
- The number of syllables of ω_i is equal to the number of maximal blocks of vowels p. If ω_i have none or one block of vowels, then the whole ω_i is marked as one syllable. If ω_i have at least two blocks of vowels, then syllables will be created by adding consonants to blocks of vowels.
- Consonants γ_{i1}, which are in ω_i before first block of vowels, are added to this block β_{i1}. Consonants γ_{ir}, which are in the word following the last block of vowels, are added to this block β_{ip}.

Particular algorithms of class $\mathrm{P_U}$ are different in the way of adding consonants, which are between two blocks of vowels. They are named according to to the ways of the adding.

- Universal left $\mathrm{P_{UL}}$ adds all consonants between blocks of vowels to the left block.
- Universal right $\mathrm{P_{UR}}$ adds all consonants between blocks of vowels to the right block.
- Universal right $\mathrm{P_{UMR}}$ in the case of $2n$ (even count) consonants between blocks adds to both blocks n consonants. In the case of $2n + 1$ (odd count) consonants between blocks it adds to the left block n consonants and to the right block $n + 1$ consonants.
- Universal right $\mathrm{P_{UML}}$ in case of $2n$ (even count) consonants between blocks adds to both blocks n consonants. In the case of $2n + 1$ (odd count) consonants between blocks it adds $n + 1$ consonants to the left block and n consonants to the right block. The only exception from this rule is the case when between blocks it is only one consonant, this consonant is added to the right block.

Example 4.15

We will decompose word *priesthood* into syllables. We are using language L_{EN}. Blocks of vowels are (in order of appearance): *ie*, *oo*.

correct decomposition into syllables:	priest-hood
universal left P_{UL}:	priesth-ood
universal right P_{UR}:	prie-sthood
universal middle-left P_{UML}:	priest-hood (correct form)
universal middle-right P_{UMR}:	pries-thood

Chapter 5

Small Text Files Compression

Small text files were the first task in our research of syllable-based compression. We have chosen them because we expected that syllable-based compression could handle the small files better than the word-based compression. As the file size increased, we expected this chance to decrease. Our expectation was that with the smallest files, character-based compression would be the best, with larger files, syllable-based compression, and word-based compression would be the most efficient for the largest files. Finding the exact division points of these changes was also our priority.

The next step was to pick the suitable compression methods for the implementation. We have decided to pick one example of statistical compression methods and one example of dictionary compression methods. We have decided to design and implement syllable-based variants of LZW and Huffman Coding methods. In this phase, we were mostly interested to see whether syllable-based methods can in some cases achieve a better compression ratio than word-based methods or not and we did not consider time or space complexity too much. This chapter is based on our works [1, 2, 4].

The last part of this chapter 5.3.1 examines the possibility of compressing small files using Burrows-Wheeler Transformation. It is related to the rest of the chapter only by being focused on syllable-based compression of small text files. This part was written based on our previous articles [13, 12].

5.1 Introduction

Text compression methods are usually optimized for large or very large text files. In practice it is usually necessary to compress (collections of) smaller files like newspaper articles, mail messages, etc.

As the syllables are somewhere between characters and words, it is rea-

sonable to expect that the syllable compression could be advantageous somewhere between character compression and word compression – it is, on middle-sized files.

Knowledge of the structure of the coded message can be very useful for the design of a successful compression method. When compressing text documents, the structure of messages depends on the language used. We can expect that documents written in the same language could posses a similar structure.

The similarity of languages can be seen considering many aspects. Language classification can be made, for example, according to their use of fixed or free word order or whether they have a simple or rich morphology.

The languages with rich morphology include for example Czech or German. In these languages a syllable is a natural element logically somewhere between a character and a word. Words are often composed from two or more syllables.

5.2 Methods with Static Initialization

Related works are described in Chapters 3.2 and 3.3. Character-based methods [83, 62, 94] use static initialization and archieve good results for small files. These methods are usually trained only for English documents. Preprocessing methods [104, 97] replace each n-gram (or word) in the file by a single codeword.

We suppose that the input text for the compression is structured into the sentences and described by the follwoing rules: A sentence begins with a mixed word (first letter is an upper-case letter, other letters are lower-case) and ends with the other word (from non-alphanumeric characters), which contains a dot. Inside the sentences lower-case words and other words alternate regularly. If the sentence begins with upper-case word, then inside the sentence upper-case words and other words alternate regularly. Numeric words appear rarely and are usually followed by other words.

After the decomposition of words into the syllables there will occur a problem, we describe it in the following paragraph. Each word has a different count of syllables. Lower-case word is usually followed by other word, whereas lower-case syllable can be followed not only by other syllables but also by another lower-case syllable.

To improve a compression over the alphabet of syllables (or words) we have created a sets of characteristic syllables for each language. More details will be given in section 7.5. Syllables from those sets are used for the initialization of the compression algorithms. When coding the alphabet of

a given document, we can code only syllables that are not from the sets of characteristic syllables. This is especially useful for smaller documents; on larger files the effect is rather lower.

In this chapter the cumulative criterion has been used. Sets of characteristic syllables was created from syllables occurring in more than 1/65,000 of all occurrences of all syllables throughout the entire collection for the given language. Let us call this set C65. We created similar sets of frequent words for word-based versions of our algorithms too.

Sizes of sets of characteristic syllables are approximately 50 KB, Sizes of sets of characteristic syllables are approximately 100 KB.

Algorithm 3 LZWL compression

1: **input** message M
2: **output** encoded M
3: initialize dictionary with empty syllable and characteristic syllables of given language
4: $OldString$ = empty syllable
5: $NewString$ = empty syllable
6: $Syllable$ = empty syllable
7: **while** not end of M **or** $Syllable$ is not empty **do**
8:   **if** $NewString + Syllable$ is in the dictionary **then**
9:     $NewString = NewString + Syllable$
10:     $Syllable$ = next syllable from M
11:   **else**
12:     **if** $NewString$ is empty syllable **then**
13:       output(the code of empty syllable)
14:       output(encoded $Syllable$ by character-by-character method)
15:       add $Syllable$ to the dictionary
16:       $Syllable$ = empty syllable
17:     **else**
18:       output(the code for $NewString$)
19:       **if** $OldString$ is not empty syllable **then**
20:         $FirstSyllable$ = first syllable of $NewString$
21:         add $OldString + FirstSyllable$ to the dictionary
22:       **end if**
23:     **end if**
24:     $OldString = NewString$
25:     $NewString$ = empty syllable
26:   **end if**
27: **end while**

5.2.1 LZWL

Algorithm LZW [112] is a character-based dictionary compression method. We call the syllable-based version of this method LZWL. Algorithm LZWL can work with syllables obtained by any algorithm of decomposition into syllables, but it can be used for words too. The word-based version of LZW compression is described in [39].

We will provide a brief description of the classic LZW method [112]. The algorithm is using a dictionary of phrases, which is represented by a trie data structure. Phrases are numbered by integers according order of the addition.

In the initialization step the dictionary is filled up with all characters from the alphabet. In each next step it is searched for the maximal string S, which is in the dictionary and matches a prefix of still non-coded part of the input. The number of phrase S is then sent to the output. Actual input position is moved forward by the length of S.

Decoding has only one situation for solving. We can receive a number of a phrase which is not in the dictionary. In this case we can create that phrase by a concatenation of the last added phrase with its first character.

The syllable-based version (see Algorithm 3) is working over an alphabet of syllables. In the initialization step we add to the dictionary an empty syllable and all lower-case syllables from the set of characteristic syllables.

Finding string $NewString$, coding its number and concatenating is analogical to the character-based version, only that the string $NewString$ is a string of syllables. It is possible that the string $NewString$ can be an empty syllable. In that case we must get from the file one syllable called $Syllable$ and encode $Syllable$ by character-by-character coding algorithm.

5.2.2 HuffSyllable (HS)

HuffSyllable (see Algorithm 4) is a statistical compression method based on the adaptive Huffman coding which uses the structure of sentence in a natural language. The idea of this algorithm was inspired by HuffWord [114]. Algorithm HuffSyllable can work with syllables obtained by all algorithms of decomposition into syllables mentioned above. This algorithm can be used for words too.

An addaptive Huffman tree [59] coding syllables of a given type is built for each type of syllables (lower-case, upper-case, mixed, numeric, other). In the initialization step of the algorithm we add to the Huffman tree for lower-case syllables all syllables and their frequencies from the sets of characteristic syllables.

In each step of the algorithm expected type of actually processed syllable

Algorithm 4 HuffSyllable compression

```
 1: input message M
 2: output encoded M
 3: initialize data structures
 4: while not end of M do
 5:    Syllable = next syllable from M
 6:    Type = type of Syllable
 7:    ExpectedType = expected type of Syllable
 8:    if ExpectedType ≠ Type then
 9:       output(escape sequence for correct type)
10:    end if
11:    if Syllable is unknown then
12:       output(escape code for new node in Type Huffman tree)
13:       output(encoded Syllable by unknown-syllable coding algorithm)
14:       insert Syllable to the Type Huffman tree
15:    else
16:       output(code of Syllable in Type Huffman tree)
17:    end if
18:    increment weight of Syllable in Type Huffman
19:    if necessary, reorganize the Type Huffman tree
20: end while
```

previous type of syllable	Expected syllable
lower-case	lower-case
upper-case	upper-case
mixed	lower-case
numeric	other
other syllable without dot, last syllable from letters is not upper-case	lower-case
other syllable with dot, last syllable from letters is not upper-case	mixed
other, last syllable from letters is upper-case	upper-case

Table 5.1: Expected types of syllables according type of previous syllable.

Syllable is calculated. If *Syllable* has different type than it is expected, an escape sequence is generated. Syllable *Syllable* is then encoded by the Huffman tree corresponding to the syllable type. The calculation of the expected type of syllable uses information from the encoded part of input.

We need to know the type of last syllable. If the last syllable is other syllable, then it is known that this syllable contains a dot and that the type of the last syllable is a syllable from letters, see Table 5.1.

5.2.3 Coding New Syllable.

Although we have sets of characteristic syllables, sometimes we receive syllable K, which is not from this sets and we have to encode it. The first way is to encode K as code of length of K followed by the codes of individual characters from the syllable. The second (and better) way is to encode K as code of syllable type followed by code of length of K and codes of individual characters. We use the second way, because domain of coding function for distinct characters is given by the type of syllable and as it is smaller than in the first way. Numeric syllables are coded differently.

Encoding type of syllable depends on types of previous syllable and other criteria as in HuffSyllable. Length of codes for each types are 1, 2, 3, and 4. Average code length is 1.5 bits.

For the encoding length of syllables are used two static Huffman trees, the first one for syllables from letters and the second one for other syllables. Trees are initialized from statistics received from text documents.

For the encoding distinct characters there are used two adaptive Huffman trees, the first one for syllables from letters and the second one for other syllables.

Numeric syllables are coded differently from other types of syllables. We discover that numbers in text are naturally divided into a few categories. The first category contains small numbers (1–100), the second category represents year (1800–2000), in the third category there are very large numbers (for example 5,236,964) that usually have separated groups of digits to blocks by three. But these large numbers are decomposed into numeric words and other words. So we set maximal length of numeric word to 4, longer numeric words are split. For coding number of digits 2-bits binary coding is used. For coding distinct digits binary coding is used.

5.2.4 Experiments

For testing there were used two sets of documents in plain text format. The first set contains 69 documents in Czech with total size of 15 MB. Most of these documents were received from [41]. The second set contains 334 documents in English with total size of 144 MB. In this set there are documents from project Gutenberg [82] and `bible.txt` from Canterbury corpus [113]. From each file form project Gutenberg there were removed first 12 KB of information about project because it was the same in all documents.

We have used the set C65 for the initialization of the compression algorithms LZWL and HuffSyllable during the testing.

We have compared following methods: word-based and 4 syllable-based versions of HuffSyllable (HS) and LZWL, adaptive character-based Huffman coding (FGK), adaptive word-based arithmetic coding (ACM), bzip2, and compress 4.0. We have compared the combination of preprocessing word-based method WRT and compress 4.0 on English documents. WRT was run with parameter -0, (optimal for LZ77, other parameters are optimal for BWT, PPM, PAQ).

5.2.5 Results

We created two syllable-based compression methods that use static initialization with sets of characteristic syllables and the model of alternation of syllable types in sentences. The first method is based on LZW algorithm, the second on Huffman coding. The experimental results of these algorithms confirm our predictions, that tested syllable-based algorithms outperformed their character-based counterparts for both tested languages. Comparison of word-based and syllable-based versions of Huffman and LZW codings led to the result that in English the word-based versions of both algorithms outperform their syllable-based counterparts and in Czech the results are ambiguous: for Huffman coding word-based version outperformed syllable-based

File size Method	5 KB 50 KB	50 KB 100 KB	100 KB 500 KB	500 KB 2000 KB	2000 KB 5000 KB
LZWL+P_{UL}	3.31	3.09	2.87	2.64	2.37
LZWL+P_{UR}	3.36	3.14	2.92	2.69	2.39
LZWL+P_{UML}	3.32	3.10	2.88	2.65	2.38
LZWL+P_{UMR}	3.32	3.10	2.89	2.66	2.38
LZWL(words)	3.22	3.03	2.86	2.62	2.36
compress 4.0	3.79	3.57	3.34	3.27	3.08
HS+P_{UL}	3.23	3.18	3.15	3.10	2.97
HS+P_{UR}	3.30	3.26	3.22	3.18	3.03
HS+P_{UML}	3.26	3.22	3.19	3.15	3.02
HS+P_{UMR}	3.27	3.23	3.20	3.16	3.02
HS(words)	2.65	2.58	2.52	2.38	2.31
ACM(words) [74]	2.93	2.74	2.55	2.35	2.27
FGK [59]	4.59	4.60	4.60	4.58	4.54
bzip2 [91]	2.86	2.60	2.40	2.21	2.03
WRT [97]	2.80	2.69	2.58	2.44	2.60

Table 5.2: Comparison of compression ratio in bits per character on English documents

File size Method	5 KB 50 KB	50 KB 100 KB	100 KB 500 KB	500 KB 2000 KB	2000 KB 5000 KB
LZWL+P_{UL}	4.14	3.83	3.59	3.34	—
LZWL+P_{UR}	4.07	3.77	3.56	3.32	—
LZWL+P_{UML}	4.07	3.77	3.56	3.31	—
LZWL+P_{UMR}	4.07	3.77	3.55	3.31	—
LZWL(words)	4.56	4.19	3.99	3.69	—
compress 4.0	4.35	4.08	3.90	3.81	—
HS+P_{UL}	3.97	3.89	3.89	3.81	—
HS+P_{UR}	3.86	3.79	3.80	3.75	—
HS+P_{UML}	3.86	3.79	3.80	3.74	—
HS+P_{UMR}	3.87	3.79	3.80	3.75	—
HS(words)	3.71	3.51	3.43	3.21	—
ACM(words)	3.83	3.50	3.29	3.14	—
FGK	4.97	4.95	5.00	4.99	—
bzip2	3.42	3.10	2.88	2.67	—

Table 5.3: Comparison of compression ratio in bytes per character on Czech documents

one, for LZW coding the syllable-based one outperformed the word-based one.

There are strong inter-syllable correlations between words, which are obviously lost in zero order syllable model. This leads to poor results of syllable-based version of HuffSyllable.

Word-based preprocessing method WRT (in combination with compress 4.0) was more successful than our word-based LZWL. The explanation can be following: LZWL is using the set of characteristic words of the size 100 KB, WRT is using dictionary of words of the size 1 MB.

5.3 Methods without Static Initialization

5.3.1 Burrows-Wheeler Transformation

Although it is generally accepted that Burrows-Wheeler Transformation [29] is not suited to small files, we decided to conduct our own measurements using out small file set. The BWT algorithm was originally implemented for large file compression [10], where it proved well, so we were interested to see how it would fare with small files. Since it was implemented for large files, however, it does not use, for example, characteristic syllables set but a method of compressing the set of used syllables as a whole (see Chapter 7.4).

The Burrows-Wheeler Transformation is an algorithm that takes a block of text as input and rearranges it using a sorting algorithm. The output can be compressed with another algorithms such as bzip. To compare different ways of document parsing and their influence on BWT, we decided to deal with natural units of the text: characters, syllables and words; and compare these approaches with each other and also with the methods that divide the text into unnatural units – N-grams.

In our measurements we found out that depending upon the language, words have 5–6 characters and syllables 2–3 characters in average. Therefore 3-grams were chosen to conform to the syllable length and 5-grams to correspond to average words length.

If we want to compare different BWT for various ways of text file parsing, it is necessary to use an implementation modified only in document parsing; the rest of compression method will stay unchanged.

5.3.2 Technical Details

Documents are devided into blocks and the blocks are compressed separately. The Block size is a very important parameter of BWT. If the blocks are larger,

then results are better. For example, in bzip2 algorithm [91], the maximum block size is 900 KB. We decided to choose the block size so that any document up to 5MB could be considered as a single block. This approach is fairly-minded since e.g. word-based methods will not be favoured by considering the document as one block whilst character-based methods would split the text into several blocks.

Our implementation of BWT method consists of these steps:

1. Division into the source units: characters, words, syllables or n-grams (see Table 5.4)
2. Encoding of the set of used source units - see Chapter 7.4.
3. Burrows-Wheeler transformation (BWT) - see Chapter 2.6.1
4. Move to Front transformation (MTF) - see Chapter 2.6.2
5. Run Length Encoding of null sequences (RLE) - see Chapter 2.6.3
6. Huffman coding - see Chapter 2.4.1

We parsed an input document into 3-grams and 5-grams, words, syllables, characters. Examples of parsing the string 'consists of' are introduced in Table 5.4.

parsing	source units
orginal	`"consists of"`
letters	`"c", "o", "n", "s", "i", "s", "t", "s", " ", "o", "f"`
3-grams	`"con", "sis", "ts ", "of"`
5-grams	`"consi", "sts o", "f"`
syllables	`"con", "sists", " ", "of"`
words	`"consists", " ", "of"`

Table 5.4: Examples of parsing string "consists of" into words, syllables, letters, 3-grams and 5-grams

5.3.3 Experiments

We compared the compression effectivity using BWT of characters, syllables, words, 3-grams and 5-grams. Testing procedures proceeded on commonly used text files of different sizes (1 KB - 5 MB) in various languages: English

(EN), Czech (CZ), and German (GE). The results are in Table 5.5 and was published in [13].

Each of tested languages (EN, CZ, GE) had its own plain text testing data. Testing set for Czech language contained 1000 random news articles selected from PDT [81] and 69 books from eKnihy [41]. Testing set for English contained 1000 random juridical documents from [30] and 1094 books from Gutenberg project [82]. For German, we used 184 news articles from Sueddeutche [101] and 175 books from Gutenberg project [82].

5.3.4 Results

The primary goal was to compare the character-based, syllable-based, and word-based compression. For files sized up to 200 KB, the character-based compression appears to be optimal; for files 200 KB - 5 MB syllable-based compression is the most effective. The used language affects the results. English has a simple morphology: in the large documents the difference between words and syllables is insignificant. In languages with rich morphology (Czech, German) words are still about 10% worse then syllables, even on the large documents. Language type influence on compression is detailed in [2].

As the second aim we tried to compare the syllable-based compression with 3-gram-based and word-based compression with 5-gram-based compression. Syllables as well as words are natural language units therefore we supposed that using them will be more effective than using 3-grams and 5-grams. These assumptions were confirmed. Natural units were the most effective for small documents, where the improvement is 20 - 30 %. By increasing the document size efficiency falls to 10 - 15 % for documents of size 2 - 5 MB.

5.4 Conclusion

We compared character-based, syllable-based, and word-based versions of three different types of algorithms (LZW, adaptive Huffman coding, and BWT). We tested these methods on few thousands of small files (with the size up to 5 MB) in Czech and English language (BWT was tested for German language too). Our implementations of syllable-based and word-based versions of LZW (called LZWL) and adaptive Huffman coding (called Huff-Syllable) use initialization with characteristic syllables or words of the given language. This advantage is helpful on very small files (up to 5 - 10 KB), but the effect of this initialization is decreasing with the size of the file.

The main goal was determine for each pair of the compression method and the language what kind of source units is the best for different sizes of

— Lang.	File size Method	100 B 1 KB	1 KB 10 KB	10 KB 50 KB	50 KB 200 KB	200 KB 500 KB	500 KB 2 MB	2 MB 5 MB
CZ	Characters	**5.715**	**4.346**	**3.512**	**3.200**	**2.998**	2.846	—
CZ	Syllable	6.712	4.996	3.765	3.280	3.003	**2.825**	—
CZ	Word	7.751	6.111	4.629	3.871	3.476	3.149	—
CZ	3-gram	8.539	6.432	4.629	3.851	3.463	3.166	—
CZ	5-gram	10.104	8.415	6.566	5.448	4.796	4.265	—
EN	Characters	**5.042**	**3.018**	**2.552**	**2.647**	2.513	2.336	2.066
EN	Syllable	5.974	3.267	2.647	2.685	**2.486**	**2.282**	**1.996**
EN	Word	6.323	3.651	2.969	2.944	2.668	2.382	2.014
EN	3-gram	7.740	4.571	3.421	3.148	2.823	2.530	2.136
EN	5-gram	9.358	6.293	4.877	4.367	3.769	3.246	2.530
GE	Characters	**4.545**	**3.853**	**2.914**	**2.629**	**2.491**	2.323	2.416
GE	Syllable	5.591	4.671	3.201	2.724	2.505	**2.295**	**2.354**
GE	Word	6.343	5.491	3.679	3.119	2.865	2.545	2.608
GE	3-gram	6.813	5.583	3.760	3.117	2.820	2.525	2.519
GE	5-gram	8.545	7.429	5.324	4.320	3.744	3.237	3.004

Table 5.5: Comparison of different input parsing strategies for Burrows-Wheeler transform. Values are in bits per character.

tested documents. This results can be found in Chapters 5.2.5 and 5.3.4. We can summary these results as follows: The best results for the files up to 5 - 50 KB are achieved by word-based version of HuffSyllable. The best results for files between 5 - 50 KB and 200 - 500 KB are achieved by character-based version of BWT. The best results for files larger than 200 - 500 KB are achieved by syllable-based version of BWT. The exactly sizes of files are different for each language.

Chapter 6

Small Textual XML Files Compression

The eXtensible Markup Language (XML) [115] is a simple text format for structured text documents. XML provides flexibility in storing, processing and exchanging data on the Web. However, due to their verbosity, XML documents are usually larger in size than other exchange formats containing the same data content. One solution of this problem consists of compressing XML documents. Because XML is a text format, it is possible to compress XML documents with existing text compression methods. These methods are more effective, when XML documents have simple structure and long text content. Methods designed for compression of common XML files (see section 3.4) are the other way how to compress these files.

There are different types of text compression: text compression by characters and text compression by words. There is also a new method: text compression by syllables [2]. There were known two syllable-based methods when we were working on this chapter. The first one is LZWL (see section 5.2.1), and the second one is HuffSyllable (see section 5.2.2). Since single text compression is not able to discover and utilize the redundancy in the structure of XML, we combine syllable-based compression methods with idea of methods used for XML compression.

Syllable-based compression achieves good results on the medium-sized text documents. Since the majority of XML documents are of that size, we suppose that the syllable-based method can give good results on XML documents, especially on documents with and relatively long character data a simple structure (small amount of elements and attributes) and relatively long text content.

In this chapter we propose two syllable-based compression methods for XML documents. The first method, XMLSyl, replaces XML tokens (element

tags and attributes) by special codes in input document and then compresses this document using a syllable-based method. The second method, XMillSyl, incorporates syllable-based compression into the existing method for XML compression XMill. XMLSyl and XMillSyl are compared with a non-XML syllable-based method and with other existing methods for XML compression.

This chapter is based on our work [6].

6.1 XMLSyl

Our goal was to modify the syllable-compression method to compress XML documents efficiently. We attempted to modify existing syllable-based method so, that it treats XML tokens (element tags and attributes) as single syllables instead of decomposing them into many syllables. There were two possibilities to compel the syllable-based method to treat XML tokens as syllables:

1. Modify parser used in the syllable-based method and combine it with an XML parser, so that it can recognize XML tokens and treat them as a single syllable.

2. Replace XML tokens with bytes in the input document and then compress such a document with an existing syllable-based method.

We decided to implement the second way because this implementation allows us to make some future improvements easily. For example, we may compel the syllable-based method to assign codes with minimal length to XML tokens by adding this single bytes to the set of charactrersitic syllables (7.5). This improvement is impossible in the first variant. The encoding of XML tokens is inspired by existing XML compression methods like XMLPPM [32], XGrind [108], XPress [71], XMill [66].

6.1.1 Architecture and Principles of XMLSyl

The architecture of XMLSyl is shown in Figure 6.1. It has four major modules: the *SAX Parser* [69], the *Structure Encoder*, the *Containers* and the *Syllable Compressor*. First, the XML document is sent to the SAX Parser. Next the parser decomposes document into SAX events (start-tags, end-tags, data items, comments and etc.) and forwards them to the Structure Encoder.

The Structure Encoder encodes the SAX events and routes them to the different Containers. There are three containers in our implementation:

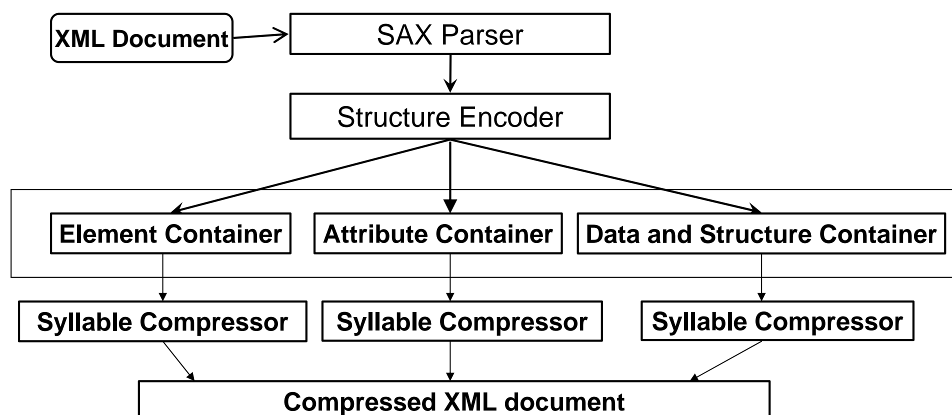


Figure 6.1: The Architecture of XMLSyl

1. **Element Container:** The Element Container stores the names of all elements that occur in an XML document. The Structure Encoder also uses the Element Container as the dictionary for encoding XML structure.

2. **Attribute Container:** The Attribute Container stores the names of all attributes which occur in an XML document. The Structure Encoder also uses the Attribute Container as the dictionary for encoding XML structure.

3. **Structure and Data Container:** The Structure and Data Container stores an XML document, in which all meta-data are replaced with special codes. The encoding process is presented in section 3.2.

When a document is parsed and separated into the containers completely, the contents of the containers are sent to the Syllable Compressor. It compresses the content of each container separately using syllable-based compression and sends the result to the output.

We have not written the SAX parser by ourselves, rather we have used the Expat parser[105] which is an open-source SAX parser written in C.

6.1.2 Encoding the Structure of XML document

The structure of XML document is encoded in XMLSyl as follows. Whenever a new element or attribute is encountered, its name is sent to the dictionary and the index of the element is sent to the Data and Structure Container.

Two different dictionaries are used for attributes and elements: the Element Dictionary and the Attribute Dictionary. The Attribute Container operates as the Attribute Dictionary and the Element Container as the Element Dictionary. Whenever an end tag is encountered a token `END_TAG` is sent to the Data and Structure container. Whenever a character sequence is encountered, it is sent to the Data and Structure Container without changes. Start and end of character sequences are indicated by special tokens. We distinguish four different character sequences: value of attribute, value of element, comment, and white spaces between tags, if white spaces are preserved.

To illustrate the encoding process, consider the encoding of the following small XML document:

```
<book>
  <title lang="en">XML</title>
  <author>Brown</author>
  <author>Smith</author>
  <price currency="EURO">49</price>
</book>
<!-- Comment-->
```

First, the XML document is converted into a corresponding stream of SAX events:

```
startElement("book")
startElement("title",("lang","en"))
characters("XML")
endElement("title")
startElement("author")
characters("Smith")
endElement("author")
startElement("author")
characters("Brown")
endElement("author")
startElement("price","currency","EURO")
characters("49")
endElement("price")
endElement("book")
comment("Comment")
```

The tokens in the SAX event stream are sent to the Structure Encoder. It encodes them and sends them to their corresponding containers. When the *book* start element token is encountered, the string *book* is sent to the Element

Element Container

element	index
book	E0
title	E1
author	E2
price	E3

Attribute Container

attribute	index
lang	A0
currency	A1

Data and Structure Container

`<book>`	`<title`	`lang="en">`	XML	`</title>`
E0	E1	A0 en `END_ATT`	`CHAR` XML `END_CHAR`	`END_TAG`

`<author>`	Brown	`</author>`	`<author>`
E2	`CHAR` Brown `END_CHAR`	`END_TAG`	E2

Smith	`</author>`	`<price`	`currency="EURO">`
`CHAR` Smith `END_CHAR`	`END_TAG`	E3	A1 Euro `END_ATT`

49	`</price>`	`</book>`	`<!--Comment-->`
`CHAR` 49 `END_CHAR`	`END_TAG`	`END_TAG`	`CMNT` Comment `END_CMNT`

Figure 6.2: Content of containers

Container since this element name was not encountered before. An index *E0* is assigned to this entry. This index is sent to the Data and Structure Container. The same operation is executed for *title* start element. String *title* is sent to The Element Container and an index *E1* is assigned to it. The index *E1* is sent to the Data and Structure Container. The element *title* has the attribute *lang*. The attribute name is sent to the Attribute Container and the index *A0* is assigned to it. The index *A0* is sent to the Data and Structure Container. Then the attribute value *"en"* is sent without a modification to the Data and Structure Container. The *"en"* attribute is followed by the token `END_ATT`, signaling the end of the attribute value. When an element value such as *"XML"* is encountered, the token `CHAR`, signaling the beginning of a character sequence, the data value and then the token `END_CHAR` are all sent to the Data and Structure Container. Finally, all the end tags are replaced by the token `END_TAG`. When a comment event is encountered, the code `CMNT` is put into the Data and Structure Container. The comment is also sent to the container and is enclosed by `END_CMNT` code. The final state of all containers is shown in Figure 6.2.

In this example we have ignored white spaces between tags, e.g. `<book>` and `<title>`, so the decompressor then produces a standard indentation. Optionally, XMLSyl can preserve the white spaces. In that case, it stores

the white spaces as the sequence of characters in the Data and Structure Container between tokens `WS` and `END_WS`.

6.1.3 Containers

The containers are the basic units for grouping XML data. The Attribute Container holds attribute names and the Element Container holds element names. As long as the number of all element and attribute names in any XML document is not high, this two containers are kept in main memory. During parsing, the containers size increases as the container is filled with entries. Each entry in the Element container is assigned a byte in the range 00-A9. These bytes are used for encoding the element names. Each entry in the Attribute container is assigned a byte in the range AA-F9. These bytes are used for encoding the attribute names. The residual 6 bytes are reserved for special codes like `CHAR`, `END_TAG` etc. In most cases, 170 (or 80) bytes are enough to encode element (or attribute) names. If the number of elements (or attributes) are greater than 170 (or 80), entries are encoded with two bytes, then tree and so on.

There is another situation with The Data and Structure Container. We do not know the size of the input XML document. The size of XML document can be so big, that document will not fit into memory, and it is not possible to increase the size of container endlessly. Therefore, the container consists of two memory block of constant size. The content of the first memory block is compressed, as soon as the container is filled. We do not compress two blocks at once, because the context of the second memory block is used for compression of the first one. After the compression, the compressed content of the first block is sent to the output and the first block swaps its purpose with the second one. Now the first block is filled with data. When it is full, the second block is compressed, and so on.

6.1.4 The Syllable Compressor

The Syllable Compressor compresses the Structure and Data Container first and sends the output to the output file. Then the Attribute Containers are compressed and sent to the output file and finally the same happens with the Element Container. LZWL is used for the compression of data. HuffSyll could be also chosen, but the performance is worse, so we decided to use only LZWL.

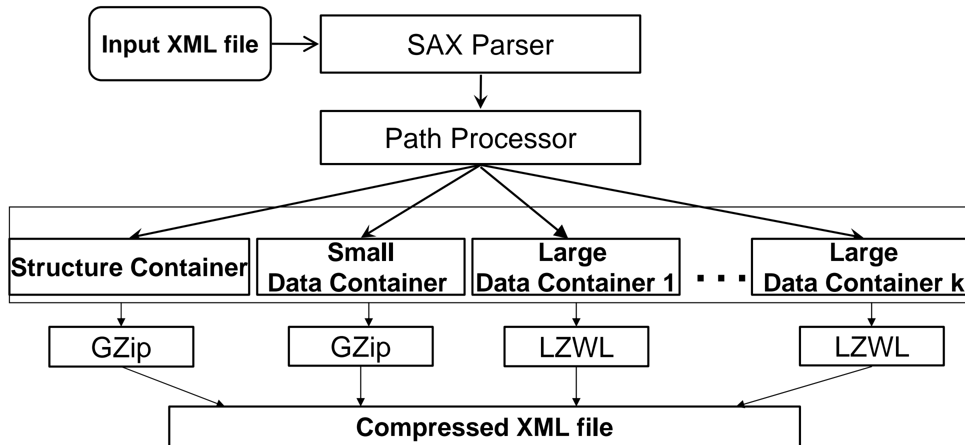


Figure 6.3: Architecture of XMillSyl

6.2 XMillSyl

This chapter introduces our second syllable-based XML method, XMillSyl. This second method incorporates syllable-based compression with the existing method for XML compression of XMill [66]. XMill has two main principles in order to optimize XML compression:

- separating structure from data content, and
- grouping Data values with related semantics in the same "container".

Each data container is then compressed individually with gzip [44]. In XMill-Syl, containers are compressed with LZWL.

We do not suppose that XMillSyl method gives better results than XMill because gzip compression performs better than LZWL. We have implemented XMillSyl in order to compare the power of XMLSyl with the power of two main principles of XMill.

6.2.1 Implementation

We did not write our implementation of XMill method. We decided to use existing sources of XMill [66].

XMill operates as follows: a SAX parser parses the XML file and the SAX events are sent to the core module of the XMill called the path processor. It determines how to map tokens to containers: element tag names and attribute names are encoded and sent to the structure container, while the

	Size	Lang	Description
elts	103919	English	Periodic table of the elements in XML
pcc	2600257	English	Formal proofs transformed to XML
stats	869059	English	One year statistics if baseball players
tal	1364576	English	Safe-annotated assembly language converted to XML
tpc	313193	English	The XML representation of the TPC_D benchmark database.

Table 6.1: The data set DS1.

data values are sent to various data containers, according to their semantic. Finally, the containers are gzipped independently and stored on disk.

We have modified compression and decompression functions (operating on containers) in the way they compress and decompress the data containers with the syllable-based method (see Figure 6.3). Moreover we have modified the syllable-based method so that it can work with the containers of XMill implementation instead of a file stream.

XMillSyl discerns the difference between small and large containers. Since LZWL is not suitable for extremely small data, the small containers are compressed with gzip. The structure container is also gzipped in XMillSyl. The large containers are compressed with LZWL.

6.3 Experiments

To show the effectiveness of XMLSyll and XMillSyl, we compared the performance of this two compression methods with one representative of XML compression methods XMill and the syllable-based method LZWL (see section 5.2.1).

6.3.1 XML Data Sources

XMLSyl and XMillSyl were tested on two data sets that cover a wide range of XML data formats and structures. The first data set (DS1) is shown in Table 6.1. It contains English XML documents with different inner structure. It includes regular data that has regular markup and short character data content (elts, stats, weblog, tpc). It also includes irregular data, that has irregular markup (pcc, tall).

The second data set (DS2) is shown in Table 6.2. It contains textual XML documents of simple structure with long character data content. It contains five stage plays marked up as XML, four in English and one in Czech. It also contains data in DocBook format in Czech and in English.

	Size	Lan	Description
errors	153530	English	"The Comedy of Errors" marked up as XML
hamlet	314677	English	"The Tragedy of Hamlet, Prince of Denmark" marked up as XML
antony	289865	English	"The Tragedy of Antony and Cleopatra" marked up as XML
much_ado	220495	English	"Much Ado about Nothing" marked up as XML
ch00	13916	English	"DocBook: The Definitive Guide" in DocBook format (1)
ch01	55015	English	"DocBook: The Definitive Guide" in DocBook format (2)
ch02	160728	English	"DocBook: The Definitive Guide" in DocBook format (3)
ch03	27799	English	"DocBook: The Definitive Guide" in DocBook format (4)
ch04	137440	English	"DocBook: The Definitive Guide" in DocBook format (6)
ch05	67142	English	"DocBook: The Definitive Guide" in DocBook format (7)
glossary	24701	English	"DocBook: The Definitive Guide" in DocBook format (8)
howto	42853	English	"DocBook V5.0, Transition Guide" in DocBook format.
hledani	16429	Czech	"Inteligentní podpora navigace na WWW s využitím XML" in DocBook (1)
komunikace	50881	Czech	"Inteligentní podpora navigace na WWW s využitím XML" in DocBook (2)
navihace	18495	Czech	"Inteligentní podpora navigace na WWW s využitím XML" in DocBook (3)
robot	25405	Czech	"Inteligentní podpora navigace na WWW s využitím XML" in DocBook (4)
xml	28467	Czech	"Inteligentní podpora navigace na WWW s využitím XML" in DocBook (5)
rur1	59609	Czech	"R.U.R" marked up as XML.

Table 6.2: The data set DS2.

Some data was distributed with the XMLPPM [32] and the Exalt [109] method while others were found on Internet [61], [111]. All Czech documents use Windows-1250 encoding.

6.3.2 Compression Performance Metrics

The compression ratio is defined in definition 2.9. We compare XMillSyl and XMLSyl compression ratios with those of XMill. The compression ratio factor shows normalization of the compression ratio of XMillSyll or XMLSyl

	CR_{LZWL}	CR_{Xmill}	$CR_{XMillSyl}$	$CRF_{XMillSyll}$	CR_{XMLSyl}	CRF_{XMLSyl}
elts	1,04	0,47	0,54	**1,15**	0,72	**1,53**
pcc	0,22	0,02	0,03	**1,50**	0,04	**2,00**
stats	0,67	0,33	0,40	**1,21**	0,39	**1,18**
tal	0,36	0,09	0,12	**1,33**	0,15	**1,67**
tpc	1,82	1,05	1,54	**1,47**	1,60	**1,52**
Average	0,82	0,39	0,53	**1,33**	0,58	**1,58**

Table 6.3: Compression ratio for the data set DS1 (in bits per bytes).

	CR_{LZWL}	CR_{Xmill}	$CR_{XMillSyl}$	$CRF_{XMillSyll}$	CR_{XMLSyl}	CRF_{XMLSyl}
errors	1,98	1,83	2,00	**1,09**	1,83	**1,00**
hamlet	1,96	1,91	2,00	**1,05**	1,85	**0,97**
antony	1,84	1,79	1,88	**1,05**	1,69	**0,94**
much_ado	1,88	1,80	1,89	**1,05**	1,77	**0,98**
ch00	3,28	2,69	3,00	**1,12**	2,88	**1,07**
ch01	2,69	2,20	2,43	**1,10**	2,46	**1,12**
ch02	1,76	1,43	1,70	**1,19**	1,57	**1,10**
ch03	2,90	1,87	2,70	**1,44**	2,08	**1,11**
ch04	2,09	1,66	1,78	**1,07**	1,83	**1,10**
ch05	2,28	1,81	2,03	**1,12**	2,04	**1,13**
glossary	2,07	1,64	1,84	**1,12**	1,89	**1,15**
howto	6,69	2,30	2,50	**1,09**	2,59	**1,13**
hledani	3,79	3,13	3,62	**1,16**	3,40	**1,09**
komunikace	3,25	2,65	2,93	**1,11**	3,01	**1,14**
navihace	3,79	3,14	3,68	**1,17**	3,44	**1,10**
robot	3,43	2,86	3,22	**1,13**	3,04	**1,06**
xml	3,74	3,23	3,69	**1,14**	3,30	**1,02**
rur1	2,33	2,07	2,37	**1,14**	2,15	**1,04**
Average	**2,88**	**2,22**	**2,51**	**1,13**	**2,38**	**1,07**

Table 6.4: Compression ratio for the data set DS2 (in bits per bytes).

with respect to XMill. The compression ratio factor is defined as follows:

$$CRF_{XSyl} = \frac{CR_{XSyl}}{CR_{XMill}}.$$

6.3.3 Experimental Results

The compression ratio statistics of two sets of XML documents are shown in Table 6.3 and Table 6.4.

The syllable-based method performed worse on documents from the data set DS1. On the other hand, both XMLSyl and XMillSyl shows great improvement comparing to LZWL. They compressed the input to 50-60% of the size of the compressed file with LZWL.

On XML documents of the data set DS2, LZWL provides a reasonably good compression ratio - on the average, about two-thirds that of XMill. This confirms our prediction, that syllable-based compression is effective for textual XML documents. Moreover our compression methods show even greater improvement.

On the document of the data set DS2, XMillSyl achieves about 15% and XMLSyl is about 20% better compression ratio than LZWL. Compared to XMill, both methods perform slightly worse. XMillSyl compresses about 13% and XMLSyl about 7% worse than XMill.

6.4 Conclusion

In this chapter we introduced syllable-based compression methods for XML documents called XMLSyl and XMillSyl. We presented the architecture and implementation of our methods and tested their performance on a variety of XML documents. In our experiments, XMLSyl and XMillSyl were compared with LZWL and XMill. Both methods are more suitable for textual XML documents. XMill outperformed our methods only marginally. XMLSyl performs better than XMillSyl. It implies that in our case encoding of XML structure is more efficient than separating a structure from data and grouping data values with related meaning. XMillSyl and XMLSyl show better results for Czech language.

Results of our algorithms were much worse than we expected, so we decided to stop working on this research. The compression ratio reached by our methods was comparable or worse than the compression ratio reached by the original method XMill. We did not optimize our methods for compression speed, it was 3 - 5 worse than the compression speed of XMill. Moreover, in section 3.4 are mentioned more powerful methods for XML compression than is XMill. Some of these methods are able to query the compressed data structure.

We focused on compression of non-well-formed XML files in our next works, see Chapter 8.1.

Chapter 7

Large Alphabet Compression

We talk about a large alphabet in case of an alphabet of strings (words or syllables). For compression methods working over large alphabet there has to be a way, how to hand over this alphabet from coder to decoder. Usually the encoded alphabet forms a part of the compressed file. While compressing very large files, the size of the encoded alphabet is insignificant to the size of the resulting output file. Efficient encoding is therefore not a considerable matter in word compression area. On the other hand, if we take smaller files into concern, the need for more efficient alphabet encoding grows. It could happen, that the alphabet code would be even longer, than the encoded file itself.

In this chapter there are three main approaches to large alphabet compression introduced. We are speaking about static, semi-adaptive and adaptive approach. We have not yet come across a study containing an elaborate comparison of these three concepts. Static approach is described in section 7.1, semi-adaptive in section 7.2 and finally fully adaptive in section 7.3. Section 7.4 referrers to our trie based semi-adaptive method aimed for compression of set of strings [5, 11]. New trend in large alphabet compression research are discussed in section 7.5. This involves among others the adaptive method with static initialization. Inventing an effectual mechanism of this initialization is one of the leading topics of this research.

7.1 Static Approach

When it comes to compression methods that operate on a small alphabet, the alphabet is well known to both the coder and the decoder in advance. It is formed just by the members of the character set. Although most of

the texts do not contain more than one third of all available characters, the compression ratio is not affected much by the redundant characters.

In case of syllables or words this philosophy cannot be directly applied, as there are far too many strings, which match this definition. Clearly, their number grows exponentially with the maximal possible token length. Only a tiny minority of these strings are such, that form a word or a syllable in a language. Therefore initializing the coder with all these strings would lead to unacceptably long token codes. This could totally ruin the efficiency of compression by producing compressed files larger, than their uncoded counterparts.

This does not mean, that static initialization would be in general unsuitable for word-based [98] and syllable-based [110] compression. An indirect technique can be applied. First a set of characteristic syllables (or words) is created. The compression then uses joined set of these characteristic syllables (or words) and alphabet of characters. These methods are often called as word or syllable based by their authors, but the qualification as *hybrid* would probably suit them better.

7.2 Semi-adaptive Approach

Another possible approach is called semi-adaptive. This involves parsing the whole document to obtain the set of used words or syllables. This set does not need to be ordered. But this concept has one drawback; the input file has to either be whole loaded into memory or processed twice. It is used in XBW [15] method, because this method requires loading the input into memory anyway, so that the Burrows-Wheeler transformation may be performed. Hence using the semi-adaptive approach does not lay any extra memory or processing time requirements.

Semi-adaptive alphabet compression is broadly used. Particular methods are discussed among related works in Chapter 3.5

The class of semi-adaptive compression methods for compression a set of string is represented by TD1, TD2 and TD3 [5, 11]. Their common principle is to load the set of strings into trie data structure, which is then encoded as whole. These methods are discussed in detail in Chapter 7.4.

7.3 Adaptive Approach

Last approach, we are going mention, is called adaptive. Its advantage is, that the syllable or word may be encoded, whenever in the coding phase

(usually in the time of the first appearance of the syllable in the coded file). Its drawback is, that the encoding of strings one by one may be less efficient, than encoding the whole set of strings together.

There are two ways how to encode a string. We can encode it using alphabet of characters and a special terminating symbol indicating end of the string, or we can encode the length of the string followed by its characters.

Adaptive methods can be improved significantly with the knowledge of the language of the message to be compressed. A set of characteristic syllables can be assembled and used in a compression process. There are two ways how to use this set.

This set can be used during the initialization of the compression method, the strings in the set do not need to be in the encoded form of the used alphabet. This approach is called *adaptive with static initialization* and it has been another topic of our research. It is described in more details in Chapter 7.5. This concept of set of characteristic strings was used in LZWL and HuffSyll [2] compression methods, which were developed for the compression of very small files. The main issue is a proper choice of syllables for the set of characteristic syllables. This matter is also under our attention [3, 4, 14].

The other possibility is not using the set of characteristic syllables for initialization, but for new syllable encoding later on. If we assign an unique number (code) to each syllable in the set, then we get a dictionary. When new syllable is encountered, the dictionary is searched and when the syllable is present, its code from the dictionary is used.

Similar sets of charactersitic words can be made for word-based compression.

7.4 Set of Strings Compression

Compression methods working over large alphabet need transfer the used alphabet (set of strings) as a part of the compressed message. The set of strings needs to get some ordering, because we need to identify the strings during the decompression phase. We do not need to encode the ordering explicit, but we need to reconstruct them by a decoder. The best way is transforming the set of strings into a dictionary of strings. We suppose that a *dictionary* is a set of ordered pairs (*string*, *number*), where the *string* is a string over an alphabet Σ and the *number* is an integer of the range 1–n where n is the number (identifier) of the ordered pair in the dictionary.

This chapter describes a method of compressing the set of strings based on the coding of the set by a trie and on effective encoding of the trie. During

the encoding there is an unique number assigned to each string using depth-first traversal of the trie. So the set is transformed into a dictionary.

We describe general methods TD1 and TD2 based on the encoding the set of string by a trie and on following compression of the trie. There is also an improvement, called TD3, applicable in the cases when the set of the strings can be split into several parts containing different (easily recognizable) classes of strings. The efficiency of the described methods is evaluated on the set of string collected as words or syllables from Czech, English, and German documents.

7.4.1 Existing Methods

Our research is focused on the text documents compression hence we will provide a brief description of the methods used in this field.

It is quite common for the papers on word-based and syllable-based compression methods that their authors give no big importance to the compression of the used alphabet as the alphabet often makes only a small part of the compressed message. It is probably true for very large documents but for middle-sized documents the importance of the alphabet size grows as the alphabet takes larger part of the compressed message.

The following two approaches are the most widely used: The first approach is based on coding of a succession of strings (words or syllables) contained in it. In the second approach the set of string is compressed as a whole. All the strings are concatenated using special separators. The resulting file is then compressed using some general method.

There are described more related works about the compression of a dictionary in Chapter 3.5, but these methods was never (in works that we know) used for the compression of the used alphabet.

Character-by-character compression – CD

We will describe a method published in [2] for the encoding of strings using a partitioning of the strings into five categories, similarly to the method TD3 described below. Every string is encoded as a sequence of string type code, string length code and by the codes of the individual characters. String type is encoded using binary phase coding (*c1*), string length is encoded by adaptive Huffman code (*c2*), and individual characters are coded also using adaptive Huffman code (each class has its own Huffman tree – hence we distinguish three different codes: letters by $c3$, numbers by $c4$, and other characters by $c5$). Lower-case and upper-case letters use the same code value

$c3$, they are distinguished by the syllable type. All adaptive Huffman trees are initialized according language specification. Examples are given in Fig. 7.1.

```
code("to") = c1(mixed), c2(2), c3('t'), c3('o')
code("153") = c1(numeric), c2(3), c4('1'), c4('5'), c4('3')
code(". ") = c1(other), c2(2), c5('.'), c5('0')
```

Figure 7.1: An example of a coding a string by the CD method

It is not necessary to know the whole set of strings at the beginning. It is possible to compress individual strings on the fly. It is then possible to encode new string whenever they are encountered. Other methods discussed in this paper need to compress the whole set of strings at once.

External Compression

Let us have a separator τ being not part of the used alphabet Σ. Let all the strings over alphabet Σ forming the set S of strings that are concatenated to a single string using the separator τ. The resulting string is then encoded using an arbitrary compression method. In [55] the authors tried to encode the set of words using gzip, PPM, and bzip2 methods and recognized as best for this purpose bzip2. We tried to encode the set of strings using bzip2 [91] (in the tables denoted as BzipD – bzip compressed dictionary) and LZW [112] (denoted in the tables as LZWD – LZW compressed dictionary).

Front Compression

Front compression is a method for compression of a lexicographically sorted set S of strings $\omega_1, ..., \omega_n$, where $\omega_i = \omega_{i1}...\omega_{ip}$. Let $\delta_i = \delta_{i1}...\delta_{ir}$ is the maximal common prefix of ω_{i-1} and ω_i, $\delta_1 = \lambda$. The set S is encoded string by string. Every string ω_i is encoded as gamma code of r, gamma code of $p - r$ and characters $\omega_{i(r+1)}...\omega_{ip}$.

Our methods TD1, TD2, TD3 desribed in Chapter 7.4.2 are derived from the front compression method.

7.4.2 Trie-Based Compression

When designing the introduced methods TD1, TD2, and TD3 we decided to represent the set of strings by a data structure *trie* [60, Section 6.3: Digital Searching, pp. 492–512]. Trie T is a tree of maximal degree n, where n is the size of the alphabet Σ and satisfies following conditions: The root represents

an empty string. Let the string α be represented by the node A, the string β represented by the node B. If the node A is father of the node B, then the string β is created by concatenation of the string α and one character from Σ. For all nodes A and B there exists a node C that represents common prefix of strings α and β and this node is on both paths (including border points) from the root to B and from the root to A.

The trie is created from the strings appearing in the set. Then the trie is encoded. During this encoding there is a unique number assigned to each string using depth-first traversal of the trie.

Basic Versions – TD1 and TD2

Our basic versions of set of strings compression are based on sharing common prefixes of the strings. Individual strings are represented as the paths in a trie representing the whole set. The compression is based on sharing common prefixes and efficient encoding of the trie.

Node type:	Coded inforrmation:		
root		# of sons $gamma_0$	represents single bit
inner node	distance $delta_0$	# of sons $gamma_0$	represents single bit
leaf	distance $delta_0$	# of sons $gamma_0$	

Figure 7.2: Different sequences of bits generated by different nodes

Trie compression of a set of strings (TD) is based on coding structure of a trie representing the set. For each node in the trie we know the following: whether the node represents a string (*represents*), the number of sons (*count*), the array of sons (*son*), and the first character of an extension for each son (*extension*). Basic version of such encoding (TD1) is given by a recursive procedure *EncodeNode* in alg. 5 which traverse the trie by a depth first search (DFS) method. For encoding the whole set of strings we run this procedure on the root of the trie representing the set.

Algorithm 5 Trie compression by TD1, TD2: procedure EncodeNode

1: **input** *node* of trie
2: **output** encoded *node* and its subtree
3: output(WriteGamma$_0$(*node.count*)) /*we encode number of sons.*/
4: **if** $node.count = 0$ **then return** /*we are in list, recursion ends*/
5: **if** *node.represents* **then**
6: output(WriteBit(1)) /*node represents string from set*/
7: **else**
8: output(WriteBit(0)) /*node does not represent string from set*/
9: **end if**
10: $previous = 0$;
11: **for** $i = 0, ..., node.count - 1$ **do**
12: $actual$ = reord($node.son[i].extension$) /*we set reord to identity for TD1, we set reord e.g. according frequency of the characters for TD2*/
13: $distance = actual - previous$ /*we calculate distance between the son and his left brother*/
14: output(WriteDelta$_0$($distance$))
15: EncodeNode($node.son[i]$) /*recursive calling of procedure on the son*/
16: $previous = actual$
17: **end for**

In procedure *EncodeNode* we code only a number of sons and the distances between the extensions of sons. For non-leaf nodes we must encode in one bit whether that node represents a string from the set (e.g. syllable or word) or not. Leafs represent strings always, it is not necessary to code it. Differences between extensions of the sons are given as distances of binary values (function reorder is identity) of the extending characters. For coding of a number of sons and the distances between them we use Elias gamma and delta codes [40]. We have tested other Elias codes too, but we achieved the best results for the gamma and delta codes. The numbers of sons and the distances between them can reach the value 0, but standard versions of gamma and delta codes starts from 1 what means that these codings do not support this value. We therefore use slight modifications of Elias *gamma* and *delta* codes: $gamma_0(x) = gamma(x+1)$ and $delta_0(x) = delta(x+1)$.

An example is given in Fig. 7.3. The example set contains the strings `".\n"`, `"ACM"`, `"AC"`, `"to"`, and `"the"`. Let us introduce the TD1 method by coding the root of the trie representing our example set:

In the node we must first encode the number of its sons. Root has 3 sons, hence we say that $gamma_0$-code of the 3 (sons) is a string of bits '00001' and we write $gamma_0(3) = 00001$.

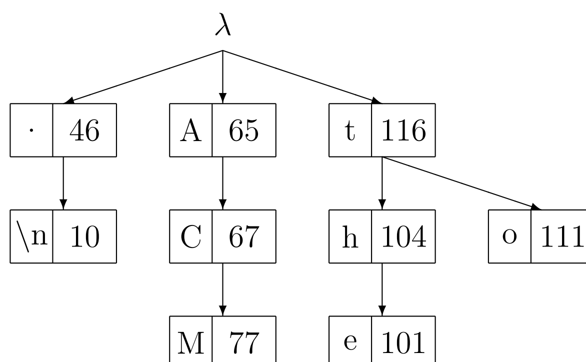


Figure 7.3: Example of a set for TD1

Then we state that the already represented word (an empty string) is not part of the set by writing a bit 0.

Value of the the first son is encoded as a distance between its value and zero by $delta_0(46 - 0) = 0100101111$.

Then the first subtrie is encoded by a recursive call of the encoding procedure on the first son of the actual node.

When the first subtrie is fully encoded, we should specify what the second son is. The difference between the first and the second son is $65 - 46$, hence we write $delta_0(65 - 46) = 000110011$.

Then we encode the second subtrie and the third son and the subtrie rooted in it. Now the whole node and all it subtries are encoded. As our example node is the root, we have encoded the whole trie representing the set of strings.

Version with Reordering – TD2

In TD1 version the distances between sons are coded according binary values of the extending characters. These distances are encoded by Elias delta coding representing smaller numbers by shorter codes and larger numbers by longer codes. In the version TD2 we reorder the characters in the alphabet according the types of the characters and their frequencies typical for given language. In our example the characters 0–27 are reserved for lower-case letters, 28–53 for upper-case letters, 54–63 for digits and 64–255 for other characters. There are some examples in Table 7.1.

Improving the procedure TD1 by an adaptation of the *reord* function from identity to a table closer to the order of the characters according their frequency in the document (or in given class of the documents) we get TD2

character	’e’	’t’	’a’	’I’	’T’	’A’	’0’	’1’	’2’	’ ’	’,’	’.’
reord(character)	0	1	2	28	29	30	54	55	56	64	65	66

Table 7.1: An example of new ordering of the characters

method. As the more frequent characters are closer to each other, the encoding of their differences requires less space.

Let us demonstrate the differences between TD1 and TD2 methods on our running example. The same set of strings is for TD2 method represented by a different tree (Fig. 7.4) – in comparison to TD1. The difference between the nodes `'h'` and `'o'` is $111 - 104 = 7$ by TD1 whereas by TD2 $6 - 3 = 3$ giving shorter encoding.

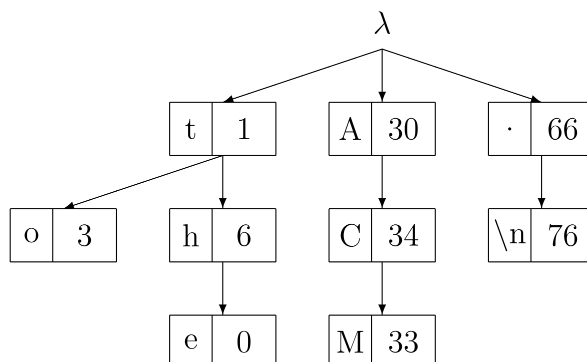


Figure 7.4: Example of a set for TD2 and TD3

7.4.3 Text-Based Set of Strings

It is sometimes useful to partition the set of all *strings* into several disjoint categories. It is possible that the join of the categories does not cover the set of all possible strings over Σ. In this case it is necessary to ensure that the input strings always fit in the given categories.

For the text compression purposes this requirement can be met e.g. by a proper input string selection (partition of the input message into properly formed subparts). Words and syllables are special types of such strings.

Version Using Types of Strings – TD3

Words and syllables are special types of strings. According [2] we recognize these five types of words (and syllables): lower-case words (from lower-case letters), upper-case words (from upper-case letters), mixed words (having the

first letter upper-case and the following letters lower-case), numeric words (from digits), and other words (from special characters). We know the type of a coded string for some nodes in the trie (in Fig. 6 *IsKnownTypeOfSons*) and we can use this information.

If a string begins with a lower-case letter (lower-case word or lower-case syllable), the following letters must be lower-case too. In a trie each son of a lower-case letter can be only a lower-case letter too. Similar situation is for other words and numeric words. If a string begins with an upper-case letter, we must look at the second character to recognize the type of the string (mixed or upper). In our example (Fig. 7.4) we know for the nodes `'t'`, `'o'`, `'h'` and `'e'` that all their sons are lower-case letters.

In the new ordering described in the version TD2 it is given for each character type some interval of the new orders. Function first returns for each type of characters the lowest orders available for given character type. Function *first* is described in Tab. 7.2.

type of characters	lower-case letter	upper-case letter	digit	other
first(type)	0	28	54	64

Table 7.2: Values of function first

We are counting and coding (Alg. 6, lines 15-17) the distances between the sons. For the first sons of some nodes of a known type, we can use function *first* and decrease the value of the distance and shorten the code. We modify the version TD2 by a modifying of the line 9 and inserting the lines 10 - 13 getting the version TD3.

We show the differences between TD3 and TD2 on our example (Fig. 7.4).

We start directly by the node `'t'`. Here we must first encode the number of the sons of this node (2), we write $gamma_0(2) = 011$.

Then we state that the already represented word (string `"t"`) is not a part of the set by writing a bit 0.

Value of the first son of our node between its value (3) and zero (it is the first son) is decreased by a value of the function $first$ for a lower-case letter (0). Encoded value is $delta_0(3 - 0 - 0) = 01100$. Only lower-case letters can occur in a subtrie of the node `'t'`, so we can use the function first. The encoded value is the same as in TD2 but there is a difference is in the calculation.

Other codings are made accordingly.

Algorithm 6 Trie compression by TD3: procedure EncodeNode3

1: **input** *node* of trie
2: **output** encoded *node* and its subtree
3: output(WriteGamma$_0$(*node.count*)) /*we encode number of sons.*/
4: **if** $node.count = 0$ **then return** /*we are in list, recursion ends*/
5: **if** *node.represents* **then**
6: output(WriteBit(1)) /*node represents string from set*/
7: **else**
8: output(WriteBit(0)) /*node does not represent string from set*/
9: **end if**
10: **if** IsKnownTypeOfSons(*node*) **then**
11: $previous =$ first(TypeOfCharacter(*node.extension*)) /*for the first son of the node of a known type, we can modify previous and decrease the value of the distance*/
12: **else**
13: $previous = 0$;
14: **end if**
15: **for** $i = 0, ..., node.count - 1$ **do**
16: $actual =$ reord(*node.son*[*i*].*extension*) /*we set reord e.g. according frequency of the characters*/
17: $distance = actual - previous$ /*we calculate distance between the son and his left brother*/
18: output(WriteDelta$_0$(*distance*))
19: EncodeNode3(*node.son*[*i*]) /*recursive calling of procedure on the son*/
20: $previous = actual$
21: **end for**

7.4.4 Results

— Lang.	File size Method	100 B 1 KB	1 KB 10 KB	10 KB 50 KB	50 KB 200 KB	200 KB 500 KB	500 KB 2 MB	2 MB 5 MB
CZ	PS	6.719	4.409	2.074	0.855	0.523	0.319	—
CZ	FC	4.077	2.346	0.983	0.374	0.225	0.136	—
CZ	LZWD	5.359	3.233	1.423	0.562	0.343	0.204	—
CZ	CD	3.741	2.432	1.130	0.461	0.284	0.169	—
CZ	BzipD	5.285	2.952	1.227	0.468	0.285	0.168	—
CZ	TD1	4.124	2.232	0.870	0.315	0.185	0.115	—
CZ	TD2	2.944	1.594	0.638	0.240	0.143	0.093	—
CZ	TD3	**2.801**	**1.532**	**0.612**	**0.226**	**0.134**	**0.081**	—
EN	PS	5.624	2.453	1.129	0.709	0.461	0.263	0.104
EN	FC	3.659	1.327	0.551	0.311	0.189	0.105	0.041
EN	LZWD	4.580	1.715	0.732	0.426	0.269	0.152	0.059
EN	CD	2.983	1.287	0.583	0.360	0.234	0.133	0.052
EN	BzipD	4.390	1.523	0.626	0.353	0.222	0.124	0.047
EN	TD1	3.792	1.276	0.506	0.272	0.158	0.086	0.033
EN	TD2	2.871	0.954	0.384	0.212	0.124	0.069	0.028
EN	TD3	**2.666**	**0.890**	**0.354**	**0.195**	**0.116**	**0.063**	**0.024**
GE	PS	5.820	4.410	1.877	0.993	0.593	0.356	0.171
GE	FC	3.697	2.476	0.875	0.422	0.241	0.142	0.063
GE	LZWD	4.259	2.995	1.139	0.580	0.345	0.202	0.104
GE	CD	3.068	2.360	0.997	0.530	0.315	0.185	0.091
GE	BzipD	4.127	2.689	0.949	0.479	0.285	0.166	0.087
GE	TD1	3.952	2.539	0.832	0.377	0.207	0.122	0.045
GE	TD2	3.020	1.914	0.627	0.284	0.157	0.097	0.035
GE	TD3	**2.730**	**1.805**	**0.599**	**0.275**	**0.150**	**0.086**	**0.033**

Table 7.3: Set of syllables: Compression ratio (Compared with the size of a whole file) in bits per character

We have tested three versions of the method compressing the set of using the trie data structure (TD – variants TD1, TD2, TD3), one method compressing the dictionary character-by-character (CD), two methods using external compressing methods for the concatenated directory items (LZWD, BzipD), and front compression method (FC). As basic measure we have used simple store of the strings in pascal format (length and characters; PS).

It is necessary to note that TDx methods assign to each string form the set an unique identifier, so they transformation the set into dictionary.

We tested the sets of words and syllables for variously sized documents written in the following three languages: English (EN), German (GE), and Czech (CZ).

The best for the sets of syllables to be the method TD3 that outperformed all other tested methods on all tested document sizes. For example, when

—	File size	100 B	1 KB	10 KB	50 KB	200 KB	500 KB	2 MB
Lang.	Method	1 KB	10 KB	50 KB	200 KB	500 KB	2 MB	5 MB
CZ	PS	7.245	6.390	4.969	3.360	2.809	2.162	—
CZ	FC	5.842	4.613	3.064	1.775	1.375	0.976	—
CZ	LZWD	5.984	4.549	3.076	1.934	1.557	1.161	—
CZ	CD	**4.378**	**3.830**	2.948	1.968	1.648	1.260	—
CZ	BzipD	5.784	4.045	**2.559**	1.582	1.255	0.921	—
CZ	TD1	8.443	6.520	4.146	2.250	1.713	1.178	—
CZ	TD2	5.935	4.531	2.874	1.550	1.176	0.814	—
CZ	TD3	5.781	4.462	2.844	**1.534**	**1.167**	**0.800**	—
EN	PS	5.734	3.310	2.057	1.604	1.314	0.860	0.375
EN	FC	4.620	2.342	1.297	0.894	0.676	0.420	0.169
EN	LZWD	4.699	2.195	1.203	0.872	0.687	0.443	0.189
EN	CD	**3.100**	**1.776**	1.095	0.847	0.695	0.454	0.197
EN	BzipD	4.508	1.915	**1.002**	**0.714**	0.563	0.361	0.154
EN	TD1	6.320	3.144	1.698	1.108	0.813	0.498	0.191
EN	TD2	4.526	2.142	1.144	0.753	0.554	0.341	0.132
EN	TD3	4.219	2.062	1.110	0.734	**0.544**	**0.333**	**0.128**
GE	PS	6.497	5.550	3.289	2.391	2.031	1.485	1.430
GE	FC	5.108	4.117	1.983	1.319	1.064	0.733	0.660
GE	LZWD	4.712	3.634	1.819	1.227	0.996	0.706	0.716
GE	CD	**3.582**	**3.091**	1.787	1.293	1.096	0.799	0.789
GE	BzipD	4.409	3.216	**1.506**	**1.001**	**0.797**	**0.558**	0.565
GE	TD1	7.187	5.748	2.585	1.700	1.383	0.945	0.844
GE	TD2	4.985	3.885	1.691	1.094	0.875	0.601	0.534
GE	TD3	4.699	3.776	1.660	1.085	0.867	0.591	**0.532**

Table 7.4: Set of words: Compression ratio (Compared with the size of a whole file) in bits per character

compressing a 10 KB document, TD3-compressed set takes about 770 bytes whereas the second best method (CD) takes about 1450 bytes. In the case of the compression of dictionaries of words the best-performing method has been for small documents (up to 10 KB) CD, for middle-sized documents BzipD, and for large documents TD3. The boundary between 'middle-sized' and 'large' documents is in this case dependent on the used language: for Czech it was about 50 KB, for English about 200 KB and for German about 2 MB.

It seems that the success of the TD methods (TD3 inclusive) grows with the average arity of the trie nodes. The syllables are short and the trie representing a set of syllables is typically dense, hence the TD3 method has been always the best.

German language has a lot of different and long word forms, the trie representing such set is quite sparse and therefore the TD3 method outperformed other methods only for set of strings of very large documents.

English typically uses less word forms than Czech and German. These word forms are often shorter than the ones used in Czech and German. The trie is then for smaller documents quite sparse and therefore our compression method outperforms the other ones only for larger documents.

In Czech the documents are typically made form a lot of middle-sized words and the tries (made from set of strings) are therefore quite dense. It is the reason why the method has been so successful for the dictionaries of Czech documents.

We also tested to compression of the set of strings by sharing suffixes instead on prefixes. The results were on our tested documents sets very similar to the prefix-sharing implementation.

7.4.5 Conclusion

We have proposed two universally applicable methods and one method advancing from a specific set of strings type. All the proposed methods represent the set of string by a trie data structure.

The better universal one (TD2) has compressed the set of syllables for given files better than any other general tested methods have. It has been outperformed only by the TD3 method which uses some additional knowledge (specific for document compression) of the set of strings structure. Both methods are also the most successful methods (in their categories) for compression of set of words of large documents.

Such sets are used by many word-based and syllable-based compression algorithms. Improving compression ratio of the set of strings improves (although with smaller impact) the overall compression ratio of these methods.

The goal of this chapter was find some suitable semi-adaptive method for the compression of the alphabet used by syllable-based or word-based compression methods. TD3 method is more successful than other methods commonly used in word-based methods.

Some improvements of TD3 method, called TD4A, ..., TD4H, TDAR, ..., TD4HR are described in [63]. Elias codes are replaced by Huffman coding and nodes with only one son are compress specially. Bitmap compression is used on bits indicate whatever nodes represents elements of the set. Some of these methods achieved compression ratio better than TD3. The improvement for large files is up to 20-30 %. These results we will publish in the future.

7.5 Sets of Characteristic Syllables

Text compression methods are usually applied on large files or collections of large files. We suppose that it is interesting to search also for compression methods effectively applicable on large collections of individually accessible small files (i.e. without using `tar` to convert the collection to a large file or creating "solid" archive). For instance, in the WWW environment the access to small or middle-sized html pages is necessary.

Many compression methods require some minimal file size to be useful (effective enough). In the case of text documents this minimal file size is given by the need of transfer the used alphabet between coder and decoder. We suppose that for processing of a collection of small individually accessible documents it is possible to improve the lower bound of compression methods by a proper initialization of the compression dictionary. In case of syllable-based compression we can choose some syllables which are frequently used in given language and create from those strings *the set of characteristic syllables* of a given language. Similar approach can be made for word-based compression too, there we will have set of characteristic words.

7.5.1 Creating and Criteria

It is not easy to create a set of characteristic syllables of a given language. Such sets may be then used as initial dictionaries for the different compression methods. The set of characteristic syllables can have a crucial impact on the achieved compression ratio: If there are too many syllables in this set, it may happen that most of them are not used during the compression and the codes of useful syllables are therefore unnecessarily big. The second extreme occurs when the set of characteristic syllables is empty. Then each syllable must be on its first occurrence encoded character by character what is rather expensive. An optimal set of characteristic syllables is a compromise between these two approaches.

The sets of characteristic syllables differ for different languages and for different algorithms of partitioning words into syllables. The criteria for inclusion into the set are, however, uniform. The syllable or word included into sets of characteristic syllables should be characteristic for a given language (and a given algorithm of decomposing words into syllables). We have to decide how many syllables we can put into the sets of characteristic syllables. The criteria for syllables to be included into sets of characteristic syllables are important.

The following two criteria seem to be reasonable:

- cumulative criterion – the quotient of the number of syllable occurrences and that number for all syllables; and
- appearance criterion – in how many documents the syllable occurred at least once.
- genetic algorithm criterion – set of characteristic syllables was assembled using a genetic algorithm

For the creation of a set of characteristic syllables it is necessary to have a sufficiently large set of testing documents that are by their contents characteristic for the given language. Both the documents and the collection should be of middle or large size to support better the appearance criterion. When the set of testing documents is selected improperly, it can happen that some rare words or syllables could be included into the set of characteristic syllables what could make the set too large. It can also happen that some words or syllables that are for the given language quite common would not be included into the set of characteristic syllables. The influence of the proper setting of set of characteristic syllables is important especially when compressing small files. We have experimentally verified that there are quite large differences in compression efficiency when different training sets were used.

Using the compressed set of documents for setting up the set of characteristic syllables can improve the compression ratio by up to 10% (comparing the best method (GA) and the worst one (C65) on smallest files). At the same time the set of characteristic syllables are reasonably small, up to the size of 100 KB. However for each combination of language and hyphenation algorithm there has to be one set of characteristic syllables. The creation of the set takes some time, but luckily is performed only once and the resulting set of characteristic syllables file can be distributed along with the compression program.

7.5.2 Cumulative Criterion

In our work [2] the cumulative criterion has been used. The sets of characteristic syllables were initialized with syllables occurring in more than 1/65,000 of all occurrences of all syllables (or words) throughout the whole collection for given a language. Let us call this set C65.

It could happen that the set of characteristic syllables contained syllables that were very frequent in just a few documents and did not occur in the other documents. For example some main character of one book can have a name, which contains some really rare syllables. Although these syllables are

not very common in the given language, and although they have appeared only in one single document of the training set, they would be included in the set of characteristic syllables.

7.5.3 Appearance Criterion

In our work [4] the appearance criterion has been used. We suppose that it is better to use the appearance criterion than cumulative criterion as it saves more syllable definitions throughout the whole collection. We tried to create different sets of characteristic syllables with different initial settings (further called Axx where xx stands for minimal percentage of documents where the syllable occurred). For comparison we have created sets of characteristic syllables occurring in at least 20% (A20), 40% (A40), 60% (A60), or 80% (A80) documents. We have also tested other Axx sets of characteristic syllables (for all multiples of 5), but their behavior was not much different from the main sets of characteristic syllables A20, A40, A60 and A80.

There are sizes of sets of charactersitc syllables and sets of charactersitc words in Table 7.5.

set of	English		Czech		German	
characteristic	words	syllables	words	syllables	words	syllables
A05	458	151	701	114	688	166
A20	174	84	152	64	163	79
A40	80	53	49	40	63	48
A60	41	35	21	26	27	32
A80	17	22	8	15	11	18
A100	1	2	1	2	1	3

Table 7.5: Sizes of sets of characterstic syllables and words (in KB)

7.5.4 Genetic Algorithm Criterion

While the precedent two criteria (cumulative and appearance) were more like heuristics with no solid theoretical grounds, the method described in our article [14] has been based on genetic algorithms.

In 1997, Üçolük and and Toroslu have published an article [110] about use of genetic algorithm in text compression for Turkish. Ideas presented in our paper [14] are strongly influenced by the results of their research, so let us give a brief summary of their method.

Üçolük and Toroslu have studied compression based on Huffman encoding upon mixed alphabet of characters and syllables[1]. This alphabet is apparently a subset of union of all characters and syllables. The issue is, which syllables should be included to ensure the optimal length of the compressed text. Observations suggested, that including nearly all the syllables usually led to the best results. To prove this theory, the whole power set of the set of all syllables had to be examined. A genetic algorithm has been designed for this task.

Nice overview of genetic algorithms can be found in [47]. The general principles are well known: Candidate solutions are encoded into individuals called *chromosomes*. Chromosomes consist of *genes*, each encoding particular attribute of the candidate solution. The values that each gene can have are called *alleles*. The encoding can be done in several different ways: *binary encoding*, *permutational encoding*, *encoding by tree*, and several others. A population of individuals is initiated and then bred to provide an optimal solution. The breeding is performed by two genetic operators – *cross-over*, in which the two selected chromosomes exchange genes, and *mutation*, where the value of a random gene is switched. The quality of a candidate solution is represented by so-called *fitness*. Fitness has influence on the probability, that the chromosome will be selected for mating. The higher the value of the fitness function, the better the solution and the better chance, that genes of the individual will carry over into next generations. After certain amount of generations the algorithm should converge to the optimum.

In this particular case the candidate solution is represented by a binary string, where the value 1 of i-th position means including the i-th syllable in the alphabet and 0 excluding it. The fitness represents the length of the text, if it was coded by Huffman encoding above the candidate alphabet. But performing compression and measuring the compressed text length would be rather expensive; it would require the Huffman tree construction which is known to be of order $O(n \log n)$ with considerably large multiplicative constant. Therefore it was decided rather to estimate this value theoretically. This can be done in linear time.

The approximation is based on two facts. The first fact can be deduced from Shannon's contribution [48]: If the entropy of a given text is H, then the greatest lower bound of the compression coefficient μ for all possible codes is $H/\log m$ where m is the number of different characters of the text.

[1] Note, that this is a slightly different approach, than the one we are using. In their concept, rare syllables are dissolved into characters every time, they occur in the coded message, raising the occurrence of its characters. In contrast, when we come across a new syllable, we encode it character by character and add it into the set of syllables. Next time we read this syllable on input, we treat it just like any other syllable.

Second, the Huffman encoding is optimal. This means the ratio of Huffman compression μ can be well estimated as

$$\mu = -\frac{1}{\log m} \sum_{i=1}^{m} p_i \log p_i \tag{7.1}$$

where p_i is the probability of the i-th character of the alphabet to occur in the text. The probability p_i is calculated as $p_i = n_i/n$, where n_i is the number of occurences of the i-th character of the alphabet in the text. Having the compression ratio makes it easy to compute the final code length l simply by multiplying μ by the bit-length of the uncompressed text, which is $n \log m$. After a little mathematical brushing up we get this formula as the desired approximation:

$$l = n \log n - \sum_{i=1}^{m} n_i \log n_i \tag{7.2}$$

Characteristic syllables and their determination by GA

We have already mentioned, how important the sets of characteristic syllables were for the compression ratio. We have also made clear, that the construction of these sets of characteristic syllables is a difficult issue. In this section we will finally introduce a genetic algorithm designed for this task.

The input of this algorithm is a collection of documents in given language, so-called *training set*. The algorithm returns a file containing the characteristic syllables as its output. The encoding of candidate solutions into chromosomes is again very straightforward; provided that the training set contains a set of N unique syllables, every individual is represented by a binary string of length N, where the value 1 on i-th position means including i-th syllable in the set of characteristic syllables, while 0 means excluding it. The role of the fitness function is played by estimated compressed length of a specimen from the training set. We are breeding the population to find a solution minimizing this value.

Algorithm 7 shows, how the evaluation of characteristic syllables works.

Our fitness function tries to approximate resulting bit length of the text compressed by HuffSyll algorithm. The behaviour of this algorithm allows us to compute this value theoretically and therefore in reasonable time. The resulting set of characteristic syllables should be optimal for use with HuffSyll. It will be interesting to examine, whether this set introduces some improvements of the LZWL effectiveness too.

Algorithm 7 Genetic algorithm for characteristic syllables

1: **input** colection of texts
2: **output** set of characteristic syllables
3: syllable space initialization
4: generate random initial population
5: **while** not last generation **do**
6: select several texts for specimen
7: *new generation* = empty set
8: **while** size of *new generation* $\leq POOLSIZE$ **do**
9: A = random individual from *old generation*
10: B = another random individual from *old generation*
11: C = cross-over(A,B)
12: add C into *new generation*
13: **end while**
14: **if** best individuals of *old generation* are better than worst new individuals **then**
15: replace up to *KEEPRATE* worst new individuals with best old individuals /*application of *elitism**/
16: **end if**
17: switch generations
18: mutate random individual
19: **end while**
20: output(last generation)

Evaluating fitness

The most important part of a genetic algorithm is the fitness function. It has to be accurate enough to provide good ordering on the set of candidate solutions and it has to be efficient, because it is called very often. The requirements concerning speed do not allow us using sophisticated calculations with high complexity.

We have decided not to use the whole training set in the fitness evaluation, but rather it is subset. For each generation we randomly select a specimen and use it for computing the fitness of all individuals. This attitude has two advantages: first, the evaluation needs less time, and second, the appearance of the syllable in the language is taken in concern. It does not only matter, how many occurrence the syllable has in the training set, but also in how many texts it appears at least once, and therefore how big the chance is, that it will appear in the specimen. After experimenting with the specimen size, we agreed on specimen consisting of five documents.

The most accurate way of evaluating fitness would be performing the actual compression and measuring the resulting file size. Again, this would be unacceptably time-consuming. We had to do an approximation similar to the one mentioned in last section.

The contribution of the characteristic syllables to the estimated bit length may be evaluated by a formula very similar to formula 7.2. The only difference is, that we will not only work with syllable frequencies in the file, which compressed bit length we are trying to estimate, but also with their frequencies in the whole training set. We will refer to these global numbers as n'_i for number of occurrences of i-th syllable and n' for the number of all syllables in the training set. Our new formula will be as follows

$$l = n \log n' - \sum_{i=1}^{m} n_i \log n'_i \tag{7.3}$$

The situation will be slightly different with the syllables marked as rare (non-characteristic). These syllables would have to be encoded character by character in the compression. They would be initialized with lower frequency, too. We take this into account in our approximation by adding an estimate of bits necessary for encoding the syllable and by increasing its code bit length by one.

The principals of the fitness evolution are outlined in pseudo code in Algorithm 8.

Algorithm 8 Evaluation of fitness

1: **input** specimen, set of syllables
2: **output** fitness
3: $R = 0$
4: **for all** $file$ in specimen **do**
5: $N' = 0, S = 0, P = 0$
6: **for all** *syllable* in set of syllables **do**
7: V = number of occurrences of *syllable* in $file$
8: V' = number of occurrences of *syllable* in all the files
9: $N' = N' + V'$
10: **if** *syllable* is marked as characteristic **then**
11: $S = S + V * \lg_2(V')$
12: **else if** $V > 0$ **then**
13: $S = S + V * (\lg_2(V') - 1)$
14: $P = P+$ estimated bit length of *syllable*'s code
15: **end if**
16: **end for**
17: N = number of syllables in *file*
18: $R = R + N * \lg_2(N') - S + P$
19: **end for**
20: output(R)

Setting parameters

The behaviour and effectiveness of a genetic algorithm depends on the settings of several parameters. These parameters include size of the population, probability of cross-over, probability of mutation, number of generations, range of elitism and degree of siding with better individuals in selection. There is no general rule for setting these parameters. The situation is even more complicated by the fact, that these parameters often act in a rather antagonistic manner.

Most authors [47] writing about evolutionary computing agree, that among these parameters the one most important is the size of the population. Population too small does not allow the algorithm to sufficiently seek through the whole search space. Inadequately large population leads to consuming too much computational power without much significant improvement in the quality of the solution. Optimal size depends on the *nature* of the problem and on its *size*[2]. Yong Gao insists, that the dependency with size is linear [46]. We have experienced good results with populations of several hundreds individuals.

One thing that is tight very closely to population size is the type of cross-over. In [99] the advantages of different types of cross-overs (one-point, two-point, multi-point and uniform) are discussed. We have decided for multi-point cross-over, because of its positive effect, when used with smaller populations. It prevents the algorithm from creating unproductive clones. We have set the number of cross-over points to the value of 10.

Elitism is an instrument against loosing the best solution found so far. It means, that instead of replacing whole old population with the new one, we keep several members of the old population as long as they are better than the worst members of the new population. Too much elitism may cause *premature convergence*, which is a really unpleasant consequence. To avoid this, we restrict elitism to small number of individuals, about one percent of the population.

In selection, better individuals are treated with favor; better chromosome has higher chance to be chosen, than the one below standard. The probability p, that an individual is chosen, may be formalized by

$$p(x_i) = \frac{k - f(x_i)}{nk - \sum_{j=0}^{n-1} f(x_j)} \tag{7.4}$$

where n stands for population size, f for fitness and constant k is set equal to $max_{x \in P}(f(x)) + min_{x \in P}(f(x))$. P stands for the population.

[2]size of problem is defined as length of candidate solution encoding

7.5.5 Experimental Results

We tested, how the use of different sets of characteristic syllables affects the compression ratio of LZWL and HuffSyllable programs. Results can be seen in Table 7.5.5. GA stands for genetic algorithms, C65 for cumulative criterion and A20 - A80 for several different sets of characteristic syllables assembled by appearance criterion.

We have tested two different languages, Czech (CZ) and English (EN). For each language we used 7000 testing documents of size. Czech ones were from [41] (69 files, size 10 KB - 1 MB) and [81] (6931 files, size 1 - 50 KB), while English documents were from [82] (333 files, size 10 KB - 5 MB) and [30] (6667 files, size 5 KB - 100 KB). The universal middle-left hyphenation algorithm (P_{UML}) was used.

7.5.6 Conclusion

We introduced several different approaches for creating a dictionary of frequent syllables. Cumulative criterion (C65) and several kinds of appearance criterions (A20 – A80) were heuristic based on our assumption only. On the other hand the method using genetic algorithms (GA) was based on the theory.

The measurement performed uncovered several interesting facts.

Good choice of sets of characteristic syllables is the most important for smallest files. When compressing files larger than 200 KB it is insignificant

The set of characteristic syllables based on A20 appearance criterion is nearly as good as the genetic based set of characteristic syllables, while its creation is much less time-consuming.

The applicability of different sets of characteristic syllables slightly differs for different languages.

Huffsyllable and LZWL gave best results when used with different sets of characteristic syllables. LZWL was most efficient with A60 and A80, while HuffSyllable with A20 and GA.

Lang.	Method	Dict.	100B-1KB	1-10KB	10-50KB	50-200KB
CZ	HuffSyll	C65	5.32	4.67	4.10	3.85
CZ	HuffSyll	GA	4.70	4.31	3.99	3.81
CZ	HuffSyll	A20	4.71	4.31	3.97	3.79
CZ	HuffSyll	A40	4.77	4.36	3.99	3.81
CZ	HuffSyll	A60	4.85	4.43	4.05	3.85
CZ	HuffSyll	A80	5.07	4.62	4.18	3.92
CZ	LZWL	C65	6.30	5.19	4.24	3.75
CZ	LZWL	GA	6.15	5.23	4.29	3.76
CZ	LZWL	A20	6.15	5.22	4.28	3.74
CZ	LZWL	A40	5.93	5.03	4.20	3.74
CZ	LZWL	A60	5.74	4.95	4.18	3.76
CZ	LZWL	A80	5.85	5.00	4.24	3.81
EN	HuffSyll	C65	4.84	3.84	3.39	3.24
EN	HuffSyll	GA	4.04	3.46	3.26	3.20
EN	HuffSyll	A20	3.95	3.46	3.27	3.19
EN	HuffSyll	A40	3.95	3.46	3.25	3.18
EN	HuffSyll	A60	3.93	3.45	3.25	3.19
EN	HuffSyll	A80	3.98	3.48	3.27	3.22
EN	LZWL	C65	5.89	3.77	2.88	2.73
EN	LZWL	GA	5.30	3.63	2.87	2.74
EN	LZWL	A20	5.45	3.73	2.91	2.73
EN	LZWL	A40	5.18	3.55	2.84	2.71
EN	LZWL	A60	5.14	3.53	2.80	2.70
EN	LZWL	A80	4.93	3.43	2.77	2.72

Table 7.6: Effect of characteristic syllables in compression of texts. Compression ratio in bits per character.

Chapter 8

Large Text Files Compression

Our motivation for compression of large text files has been fulltext system EGOTHOR [45]. These files have size around 20MB and were formed by concatenation of hundreds of web pages. The cached web pages are mostly textual and contain lots of HTML tags. However the web pages are usually not XML nor HTML well-formed hence special methods for compression of XML can not be used since they require well-formed or even valid documents.

Very common error found in HTML pages is that one tag has more attributes with the same name. Another common problem is inconsistent pairing of opening tag with closing pair. More on this topic can be found in [8].

Our goal was to merge compression methods for text (syllable, word) with methods for the compression of XML and in addition to allow compression of documents that are not well-formed. Furthermore we wanted to take advantage of the fact that these documents contain XML tags even though it need not be well-formed. Important factor has been the size of documents, which has been 15-20 MB. We decided to use Burrows-Wheeler transformation as basis for compression method since it yields very good results when big blocks are compressed.

8.1 Project XBW

Solution of the introduced problems is the project XBW, experimental results of this project were published in [16]. Project's homepage [15] contains executable version, source codes and detailed documentation including the results.

The main idea of the project XBW has been to design a system composed of modules for particulad compression methods (BWT, MTF, RLE, LZC, LZSS, PPM, arithmetic coding, Huffman coding), parser and methods TD1, TD2, TD3 (TDx module) for compression of set of used source units. Connection of modules in describes in Figure 8.1. Some modules are optional and only one module from RLE, LZC, LZSS, and PPM can be selected.

In order to allow easy connectivity data exchange among the most modules is done via simple interface consisting of a data structure array (an array of integers and its size). All implemented compression methods had to be modified for compression using large alphabet since their input has been an array of integers and not an array of characters.

Another goal has been high configurability of system in order to perform wide range of experiments. For example by setting just one parameter, use of BWT or MTF or BWT+MTF before using methods such as RLE, PPM, LZC or LZSS can be tested. How do these results vary for syllables, words and characters? How will the results differ if these methods will use arithmetic or Huffman coding as entropy coder? For many methods there is choice for using adaptive and semi-adaptive variant.

8.2 Parts of Project XBW

The diagram of program XBW is shown in Figure 8.1. An input file is split into source units (characters, syllables, words, XML tags - depending on settings). The output of this parser is an array of these source units, which represent the original file. Another output of the parser is a set of used source units, which is sent into TDx module where is encoded.

Next follow two transformations BWT and MTF, which can be independently turned off. After transformations follow compression methods RLE, LZC, LZSS and PPM which can only be used in final stage.

Compression methods RLE, LZC, LZSS and PPM use along with module TDx also module AC/HC, which performs entropic coding using arithmetic or Huffman coding.

8.2.1 Parser

Parser can read documents in different encodings including UNICODE. It can also use structure of XML even when the document is not well-formed. Output of parser is set of the source units (words, syllables, characters), which are in the input document and array of integers where one integer represents one source unit.

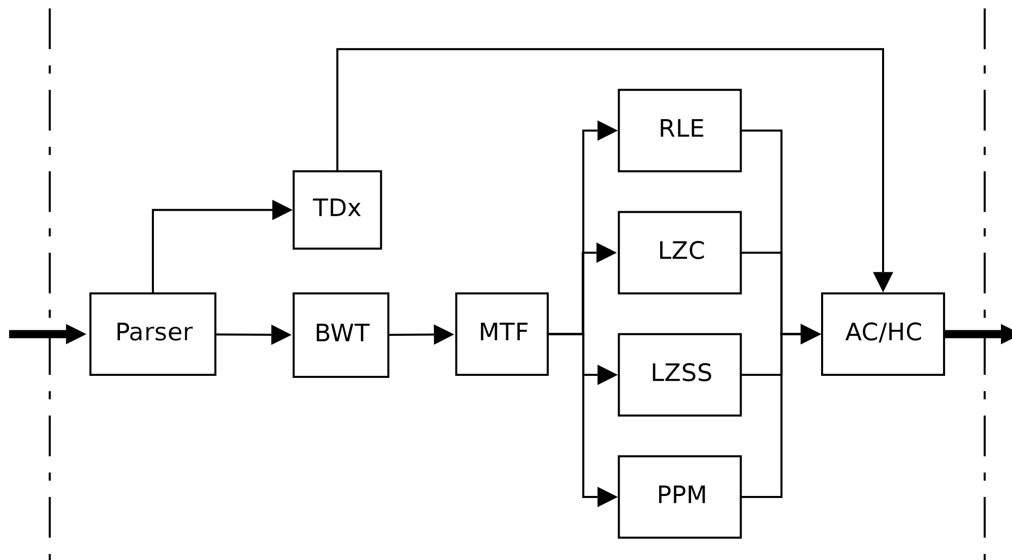


Figure 8.1: Architecture of XBW

The resulting array can be further processed by any compression method, but the set of source units has to be sent to module TDx, which compresses this set using TD1, TD2, or TD3 method.

Processing of XML

The Parser creates two separate dictionaries - one for elements and one for attributes of tags. When new item (element or attribute) is recognised it is searched for in dictionary and in case it is not found it is added into corresponding dictionary and assigned first available number, which will be its identifier. The name of the item is written to output just after the new identifier. This ensures that the dictionaries do not need to be coded explicitly and can be reconstructed during the extraction using already processed part.

We use special symbol EA that is used in three ways. It denotes the end of tag attribute and start of its content (in the example the second occurrence of EA). Second usage is replacing end character for tag (third occurrence of EA). Third way of using it, is separating the end of attribute from the beginning of its value (first occurrence of EA). Parser determines the meaning of the symbol EA by context.

Because we compress also non well-formed documents we need a special symbol EE that indicates that closing tag follows which should not be in that place. After the symbol EE we specify which tag is present.

Example 8.1 *Suppose the input*

```
<book>
  <note importance="high">money</note>
  <note importance="low">your money</note>
</note>
```

Then the dictionaries look like

Element dictionary		*Attribute dictionary*	
E1	*book*	*A1*	*importance*
E2	*note*	*A2*	empty
E3	empty		

and output is the array E1 book E2 note A1 importance EA high EA money EA E2 A1 low EA your money EA EE E2.

8.2.2 Compression of Set of Used Elements

One of the products of parser is set of used source units (words, syllables, characters). This set is stored in data structure trie, which is implemented in module TDx that can be compressed using methods TD1, TD2 and TD3 as described in Chapter 7.4. TD3 is used as default method because it obtained the best results in experiments.

Methods TDx have a very good compression ratio because they compress the set of source units as whole and not separate source units on their own. In addition methods TD2 and TD3 use the fact that syllables and words from natural language are compressed.

8.2.3 BWT

In project XBW a few algorithms for BWT [29] are implemented having different asymptotic complexity. Repetitiveness of file, AML, is defined as average length of common prefixes for all rotations of input string. The fastest algorithm for files that are not too repetitive ($AML < 1000$) is Kao's modification of Itoh's algorithm [57], which has complexity $O(AML{\cdot}n{\cdot}log\, n)$, where n is the length of the input string. For very repetitive files algorithm by Karkkainnen and Sanders [58] with complexity $O(n)$ performs the best. A comparison of different methods for BWT over large alphabet can be found in our work [19].

A very important factor for BWT is the block size, which is transformed at once, because compression methods used after BWT give better compression

ratio for larger blocks. For example, in bzip2 algorithm, the maximum block size is 900 KB. We decided to choose the block size so that any document up to 50MB could be considered as a single block. Again this value can be set as parameter in program XBW.

The choice of the algorithm for BWT has no influence on the compression ratio, it plays a key role in the time performance. Large blocks require more resources, but thanks to using words as source units we can diminish this factor. Compared to bzip2, which uses 900 KB blocks, BWT runs only twice longer with 50MB block size while yielding much better compression ratio.

Module BWT can be turned off and then the transformation is skipped. This is useful for testing compression using methods LZC, LZSS or PPM.

8.2.4 MTF and RLE

Combination of the methods MTF and RLE is the most widely used approach for the second phase in the block compression after BWT. In the project XBW this is only one of the possibilities. MTF is a reversible transformation, which can be turned off. RLE is one of the final methods (other alternatives are LZC, LZSS and PPM).

Method MTF is programmed using splay tree [56], which improves performance especially for large alphabet.

Method RLE is present in three variants: RLE1, RLE2 and RLE3 where RLE is used by default.

In the variant RLE1, repeating sequence of source units is replaced by three symbols. First is an escape sequence, which indicates that following are the symbol and number of occurrences. Compression alphabet in this case is increased only by escape symbol.

For the variant RLE2 the orginal alphabet is doubled; we add one escape symbol for each symbol from alphabet. Repeating sequence of source units is coded by pair of escape symbol for the source unit and number of occurrences.

When variant RLE3 is used we add special symbol for each source unit and number of its occurrences. Then repeating sequence of source units is replaced by special symbol for the source unit and its number of occurrences. This method is useful only for very large files or for very small alphabets setting small maximal length of repeating sequence. Otherwise the compression ratio is bad due to the extent of the alphabet.

8.2.5 Dictionary Methods and PPM

One of the goals of project XBW has been to test use of alternative methods to BWT + MTF + RLE. We chose to use dictionary methods LZSS [100]

and LZC [50] and the statistical method PPM [33]. These methods have also been implemented to support use of large alphabet.

Method LZSS we implemented as opposed to original method uses entropic coder for values D, L and N (see section 2.5.2). Similar idea is used in gzip [44].

Method PPM is configurable by run time parameters. Implemented are variants PPMA, PPMB and PPMC and for all of them the length of context and usage of exclusions can be set. Other variants of PPM (for example PPMD [53] or PPMII [95]) can be added in future.

As part of project XBW these methods have common distinct place as well as RLE that after them no compression method or transformation can be used. This is due to fact that these methods call directly entropic coder (Huffman or arithmetic) and their output is not an array of integers that is input of all other methods.

8.2.6 Entropic Coders

Project XBW implements Huffman coding (HC) and Arithmetic coding (AC), both in static and adaptive version. In addition, versions for improving speed for compression with large alphabet have been implemented. A static version of Huffman coding is implemented as the canonical version [75] and for adaptive arithmetic coding Moffat Tree [74] is used.

The most important property of this part is that the interface for Arithmetic and Huffman coding is identical, so in order to choose coding it is sufficient to change preprocessor definition parameter and recompile program. Arithmetic coder is set by default.

This is the only module that writes data to output file. Hence it is used by compression methods, which can only stand as last in queue of compression methods (RLE, LZC, LZSS, PPM) and it is also used by module TDx that codes the set of used source units.

8.3 Ambiguity of Name XBW

The name XBW has not been chosen very appropriately, because it can be easily mistaken for name "xbw transform" used by the authors of paper [42] from October 2005 for XML transformation into the format more suitable for searching. In another article [43] from May 2006 these authors renamed the "xbw transform" to "XBW transform". Moreover they used it in compression methods called XBzip and XBzipIndex. We use name XBW since July 2006 [7].

Another confusion in naming is also caused by the fact that originally we used XBW in our articles [7, 10, 13, 12] for naming syllable BWT + MTF + RLE for valid XML files. In project XBW which we named the same way, only the main idea for compression of XML using BWT remains, but otherwise it is completely different implementation.

Name XBW is used for naming different things in computer science and elsewhere, for example [34]. Luckily they have nothing to do with XML or compression, hence are not interchanged with our XBW.

8.4 Corpora

Our corpus is formed by three files which come from search engine *EGOTHOR*. The first one is formed by web pages in Czech, the second in English and third in Slovenian of respectives sizes 24MB, 15MB, 21MB. The values of AML, describing their repetitiveness, are approximately 2000. Information about compression ratio of XBW on standard corpora Calgary, Cantebury and Silesia can be found in [15].

bpB		Method			
Parser	File	BWT	LZC	LZSS	PPM
None	xml_cz	0.907	2.217	2.322	1.399
	xml_en	0.886	2.044	2.321	1.292
	xml_sl	0.710	1.982	2.010	1.205
	TOTAL	0.834	2.093	2.213	1.305
Text	xml_cz	0.906	2.206	2.296	1.395
	xml_en	0.887	2.044	2.321	1.292
	xml_sl	0.710	1.979	2.003	1.204
	TOTAL	0.833	2.087	2.200	1.303
XML	xml_cz	**0.894**	2.073	2.098	1.320
	xml_en	**0.874**	1.915	2.115	1.239
	xml_sl	**0.700**	1.850	1.797	1.129
	TOTAL	**0.822**	1.957	1.998	1.234

Table 8.1: Influence of parser on compression ratio for alphabet of characters.

8.5 Results

First we list results of program XBW for various compression methods and the effect of parser on the results. Then we show an influence of alphabet.

MB/s	No Parser				XML Parser - Characters			
	BWT	LZC	LZSS	PPM	BWT	LZC	LZSS	PPM
xml_cz	0.368	3.587	1.498	0.106	0.457	2.668	1.418	0.091
xml_en	0.419	4.028	1.297	0.125	0.544	2.915	1.249	0.104
xml_sl	0.386	4.258	1.638	0.119	0.500	2.915	1.497	0.091
TOTAL	0.386	3.906	1.485	0.115	0.491	2.810	1.397	0.094

Table 8.2: Influence of parser on compression speed

MB/s	No Parser				XML Parser - Characters			
	BWT	LZC	LZSS	PPM	BWT	LZC	LZSS	PPM
xml_cz	4.260	4.724	5.257	0.117	2.577	3.156	3.415	0.096
xml_en	4.417	4.999	5.417	0.142	2.705	3.397	3.606	0.110
xml_sl	4.918	5.236	5.946	0.134	3.012	3.299	3.672	0.097
TOTAL	4.509	4.960	5.519	0.128	2.747	3.263	3.548	0.099

Table 8.3: Influence of parser on decompression speed

At the end we compare results of XBW using optimal parameters with commonly used programs Gzip 1.2.4, Rar 3.7.1, szip 1.12, PPMonstr [96], Bzip2 1.0.4, and ABC 2.4 [22].

All published results have been obtained using arithmetic coder. BWT has been run over whole input at once followed by MTF and RLE (parameter RLE=2). PPM (variant of PPMC) run with parameters PPM_exlusions=off a PPM_order=5. The exclusions has been set off, because the exclusions have improved compression ratio only unsignificantly but they have decreased compression speed significantly.

The size of compressed files includes coded dictionary which is created always when parser is used. Compression ratio is listed in bits per byte.

The run time has been measured under Linux and stands for sum of system and user time. This implies that we list time without waiting for disk. The run time of ABC has been measured under Windows XP, because this program do not support Linux. Measurements has been performed on PC with processor AMD Athlon X2 4200+ with 2GB of RAM. The data is in megabytes per second where the uncompressed size of file is used both for compression and decompression.

We use the abbreviation *Char.* for *Characters* in some tables. Table 8.1 shows the results of compression ratio for various methods for alphabet of characters. These results show effect of XML, which improves the compression ratio by approximately ten percent.

Next in Tables 8.2 and 8.3 we list the speed of program with and without

parser using alphabet of characters. Results show that in almost all cases the parser degrades the speed. The reason is that we have to work with dictionary and the time saved by slightly shortening the input does not compensate for the work with dictionary. The exception is compression using BWT. Here the shortening the input and decreasing its repetitiveness significantly fastens BWT, which is the most demanding part of block compression.

bpB		Method			
XML Parser	File	BWT	LZC	LZSS	PPM
Characters	xml_cz	0.894	2.073	2.098	1.320
	xml_en	0.874	1.915	2.115	1.239
	xml_sl	0.700	1.850	1.797	1.129
	TOTAL	0.822	1.957	1.998	1.234
Syllables	xml_cz	0.854	1.796	1.841	—
	xml_en	0.836	1.626	1.785	—
	xml_sl	0.664	1.559	1.541	—
	TOTAL	0.783	1.672	1.723	—
Words	xml_cz	0.857	1.683	1.654	—
	xml_en	0.830	1.514	1.558	—
	xml_sl	0.668	1.457	1.390	—
	TOTAL	0.785	1.563	1.539	—

Table 8.4: Influence of alphabet on compression ratio

MB/s	XML Parser - Characters				XML Parser - Words			
	BWT	LZC	LZSS	PPM	BWT	LZC	LZSS	PPM
xml_cz	0.457	2.668	1.418	0.091	1.587	0.279	1.477	N/A
xml_en	0.544	2.915	1.249	0.104	2.009	0.920	1.093	N/A
xml_sl	0.500	2.915	1.497	0.091	1.566	0.443	1.349	N/A
TOTAL	0.491	2.810	1.397	0.094	1.666	0.399	1.319	N/A

Table 8.5: Influence of alphabet on compression speed

Method commonly used for text compression is the use of words as source units. In Table 8.4 we show the influence of alphabet on compression ratio. For textual data in English best compression ratio is achieved with using words and method BWT. For Czech and Slovenian the syllables are better, because these languages have rich morphology. One word occurs in text in different forms and each form is added into dictionary. With the use of syllables core of the word is added, which can be formed by more syllables,

MB/s	XML Parser - Characters				XML Parser - Words			
	BWT	LZC	LZSS	PPM	BWT	LZC	LZSS	PPM
xml_cz	2.577	3.156	3.415	0.096	3.986	3.951	3.923	N/A
xml_en	2.705	3.397	3.606	0.110	4.006	4.443	4.523	N/A
xml_sl	3.012	3.299	3.672	0.097	4.241	4.157	4.237	N/A
TOTAL	2.747	3.263	3.548	0.099	4.076	4.135	4.167	N/A

Table 8.6: Influence of alphabet on decompression speed

and the end of word. But these last syllables of words are common for many words and hence there are more occurrences of then in the text. For dictionary methods LZx the words are by far the best choice.

The effect of large alphabet on speed varies and is shown in Tables 8.5 and 8.6. For all algorithms the decompression is faster for words than for characters. On the other hand decompression when parser with words has been used is still slower than decompression without parser see Table 8.1. The use of words increases the speed of compression only when BWT is used. Significant increase in speed for BWT is due to shortening the input and decreasing approximately three times AML. Results for PPM and words are not shown since the program did not finish within hour.

Previous results show that the best compression ratio has the algorithm BWT. Also it is evident that parser improves compression ratio for all algorithms. The fastest compression is achieved using LZC and fastest decompression using LZSS.

Our primary criterion is compression ratio and since method BWT has by far the best compression ratio, we focus mainly on BWT. In case the speed is priority choice of dictionary methods is advisable.

Table 8.7 contains comparison of compression ratios for different choices of parser, which shows that words are best for English and syllables for Czech and Slovenian. The choice of either words or syllables depends on the size of file and on morphology of the language. For languages with rich morphology, and for smaller files the syllables are better. Choice of either words or syllables effects the number of occurrences of symbols from dictionary for the input text. In program XBW we have implemented a few methods for splitting words into syllables. Results have been obtained using the choice *Left*. More details can be found in [15]. Interesting is the fact that the XML mode of parser has small influence on compression ratio. This is not due to incorrect implementation of parser, but due to properties of BWT for large blocks. For example for LZx methods the effect is significant. Again more detailed results are in [15].

Table 8.8 show the influence of parser on the speed of program. The fastest by far is the choice of words as source units for compression. For decompression (see Table 8.9) the differences are small. In order to improve the speed it is better to use parser in text mode instead of XML mode for words.

There are many algorithms for sorting suffixes in BWT. The choice of this algorithm has big impact of overall performance of compression. Without the use of parser, sorting suffixes for big blocks amount up 90% of run time of whole program. More details are in [103]. For all files the fastest is Kao's modification of Itoh's algorithm [57] and it has been used in all measurements when BWT has been used.

Run time of separate parts of program are in Table 8.11. These times show in which parts there is the most room for improvement.

bpB	No Parser	Text Parser			XML Parser		
		Char.	Syllables	Words	Char.	Syllables	Words
xml_cz	0.907	0.906	0.855	0.862	0.894	**0.854**	0.857
xml_en	0.886	0.887	0.836	0.836	0.874	0.834	**0.830**
xml_sl	0.710	0.710	0.666	0.672	0.700	**0.664**	0.668
TOTAL	0.834	0.833	0.789	0.790	0.822	**0.783**	0.785

Table 8.7: Compression ratio for BWT.

MB/s	No Parser	Text Parser			XML Parser		
		Char.	Syllables	Words	Char.	Syllables	Words
xml_cz	0.368	0.324	1.056	**1.767**	0.457	1.073	1.587
xml_en	0.419	0.364	1.225	**2.128**	0.544	1.330	2.009
xml_sl	0.386	0.331	1.102	**1.790**	0.500	1.135	1.566
TOTAL	0.386	0.336	1.110	**1.853**	0.491	1.150	1.666

Table 8.8: Compression speed for BWT

MB/s	No Parser	Text Parser			XML Parser		
		Char.	Syllables	Words	Char.	Syllables	Words
xml_cz	4.260	2.628	4.277	**4.817**	2.577	3.710	3.986
xml_en	4.417	2.494	4.612	**4.764**	2.705	3.981	4.006
xml_sl	4.918	2.639	4.686	**5.442**	3.012	3.685	4.241
TOTAL	4.509	2.598	4.494	**5.002**	2.747	3.765	4.076

Table 8.9: Decompression speed for BWT

8.5.1 Influence of File Size

We compared compression ratio of different methods (LZSS, LZC, BWT + MTF + RLE, and PPM) used in XBW on different sizes of files. We used files files xml_cz, xml_en, and xml_sl and their suffixes of sizes 10^4 B, 10^5 B, 10^6 B, and $5 * 10^6$ B. We set the parser to text mode. Results are shown in Table 8.10. Some results for word-based and syllable-based versions of PPM are not shown, because their compression times are longer than 5 hours.

Both word-based and syllable-based versions of LZSS and LZC have better compression ratio than character-based for all languages and file sizes. Word-based versions are better than syllable-based versions. There are some exceptions for the smallest sizes of files (10^4 B), where character-based or syllable-based versions sometimes outperform word-based versions.

The character-based version of BWT has the best compression ratio for the sizes of files up to (10^6 B), the syllable-based version has the best compression ratio for the larger sizes of files. The difference in compression ratio between the syllable-based and the word-based version is very small and the time of compression and decompression is about 1.5 - 2 times better for the word-based version.

The character-based version of PPM has the best compression ratio for the smallest sizes of files, the syllable-based version has the best compression ratio for the middle sizes of files, and the word-based version has the best compression ratio for the largest sizes of files. PPM has very slow compression time, for example, the character-based version compresses files of the size $5 * 10^6$ B in 1 minute. The syllable-based version compresses this kind of files in 1 hour and the word-based version compresses in 2 hours.

8.5.2 Comparison with Other Programs

For comparison we show the results of programs Gzip, Rar, ABC, szip, PPMonstr, and Bzip2. Programs for compression of XML data such as XMLPPM [32] and Xmill [66] can not cope with non-valid XML files. Hence we could not get their results on our data. For all tested programs we used parameters for the best available compression. XBW was using only text parser (not XML), because the other compared methods are not XML based. In Table 8.12 we list compression ratios. Our program compresses all files the best and is significantly better for files which are not in English.

In Tables 8.13 and 8.14 are the results for speed of compression and decompression. The fastest is Gzip, but it also has the worst compression ratio and hence we compare speed of XBW only with other methods. Compression for XBW takes less twice the minimum of the other methods (except gzip).

Decompression is comparably fast as for Rar and ABC. Decompression speed of Bzip2 is approximately three times faster, szip is twice faster.

The performance of XBW is sufficient for common use, however it is slower than the speed of hard disks, and hence where speed is priority, it is better to use program based on dictionary methods such as Gzip. XBW has the best compression ratio and therefore it is suitable especially for long term archiving.

8.6 Future Work

In future work on XBW we aim to focus on three directions. The first is creation of parser which could be used also on binary data. The later is improving the run time of program where again we see the biggest potential in parser.

The third direction is adding new compression methods. We have only very simple versions of PPM, we plan add some more powerfull versions like PPMonstr [96], there exists some sophisticated alternatives of MTF [22, 90].

bpB		10^4 B	10^5 B	10^6 B	$5 * 10^6$ B	whole
LZSS en	Char.	**3.276**	2.618	2.381	2.313	2.321
	Syllables	3.339	2.208	1.936	1.851	1.849
	Words	3.468	**2.167**	**1.764**	**1.623**	**1.582**
LZSS cz	Char.	**2.900**	2.618	2.381	2.313	2.296
	Syllables	2.973	2.461	2.081	1.948	1.890
	Words	3.014	**2.458**	**1.983**	**1.795**	**1.673**
LZSS sl	Char.	2.140	2.058	2.011	1.998	2.003
	Syllables	1.961	1.770	1.659	1.610	1.584
	Words	**1.946**	**1.750**	**1.580**	**1.485**	**1.403**
LZC en	Char.	4.247	3.178	2.654	2.257	2.044
	Syllables	**3.705**	2.618	2.182	1.846	1.654
	Words	3.749	**2.536**	**2.062**	**1.729**	**1.530**
LZC cz	Char.	3.897	3.435	2.767	2.549	2.206
	Syllables	3.349	2.952	2.398	2.191	1.824
	Words	**3.292**	**2.874**	**2.294**	**2.082**	**1.692**
LZC sl	Char.	3.402	2.833	2.522	2.286	1.979
	Syllables	2.482	2.208	2.057	1.883	1.581
	Words	**2.337**	**2.115**	**1.192**	**1.778**	**1.462**
BWT en	Char.	**3.066**	**1.842**	**1.389**	1.106	0.887
	Syllables	3.388	1.945	1.394	**1.071**	**0.836**
	Words	3.439	1.994	1.421	1.082	**0.836**
BWT cz	Char.	**2.693**	**2.067**	**1.541**	1.335	0.906
	Syllables	3.088	2.259	1.588	**1.334**	**0.855**
	Words	2.997	2.293	1.645	1.375	0.862
BWT sl	Char.	**1.864**	**1.430**	**1.228**	1.063	0.710
	Syllables	1.946	1.503	1.247	**1.051**	**0.666**
	Words	1.871	1.538	1.284	1.083	0.672
PPM en	Char.	**3.353**	2.204	1.742	1.454	1.292
	Syllables	3.358	**2.105**	1.615	1.301	—
	Words	3.486	2.141	**1.611**	**1.283**	—
PPM cz	Char.	**3.024**	2.456	1.906	1.709	1.395
	Syllables	3.090	**2.417**	**1.844**	1.583	—
	Words	3.083	2.439	1.848	**1.573**	—
PPM sl	Char.	2.114	1.824	1.622	1.461	1.204
	Syllables	2.019	**1.701**	**1.498**	1.312	—
	Words	**1.991**	1.723	1.503	**1.311**	—

Table 8.10: Influence of file size on compression ratio

Seconds	Compression				Decompression			
	Parser	BWT	MTF	RLE	Parser	BWT	MTF	RLE
xml_cz	4.668	7.788	0.748	0.720	1.980	0.764	0.868	1.328
xml_en	2.364	3.800	0.388	0.448	1.112	0.716	0.440	0.796
xml_sl	3.352	7.404	0.496	0.504	1.592	0.676	0.556	0.916
TOTAL	10.384	18.992	1.632	1.672	4.684	2.156	1.864	3.04

Parser in text mode using words; BWT using Itoh; RLE - version 3

Table 8.11: Running time for different parts of XBW

bpB	XBW	Gzip	Bzip2	Rar	ABC	szip	PPMonstr
xml_cz	**0.857**	1.697	1.406	1.161	1.196	1.305	1.048
xml_en	**0.830**	1.664	1.299	0.851	0.974	1.139	0.934
xml_sl	**0.668**	1.373	1.126	0.912	0.946	1.057	0.812
TOTAL	**0.785**	1.584	1.275	0.998	1.054	1.178	0.938

XBW: parser in text mode using words, Kao's algoritm for BWT
Gzip: gzip -9; Rar: rar -m5; Bzip2: bzip2 -9; ABC: abc -9; Szip: szip -b41;
PPMonstr ppmonstr -o1 -m256;

Table 8.12: Comparison of compression ratio

MB/s	XBW	Gzip	Bzip2	Rar	ABC	szip	PPMonstr
xml_cz	1.732	**10.320**	3.170	2.708	1.603	4.040	1.033
xml_en	2.058	**11.587**	3.454	2.689	1.034	3.826	1.026
xml_sl	1.758	**13.713**	3.245	3.190	1.645	4.196	1.076
TOTAL	1.812	**11.634**	3.262	2.853	1.422	3.967	1.046

XBW: parser in text mode using words, Kao's algoritm for BWT
Gzip: gzip -9; Rar: rar -m5; Bzip2: bzip2 -9; ABC: abc -9; Szip: szip -b41;
PPMonstr ppmonstr -o1 -m256;

Table 8.13: Comparison of compression speed

MB/s	XBW	Gzip	Bzip2	Rar	ABC	szip	PPMonstr
xml_cz	4.087	**25.004**	9.430	3.955	2.691	5.794	0.988
xml_en	4.309	**46.926**	11.722	6.137	3.407	5.689	0.978
xml_sl	4.614	**46.986**	13.132	4.775	4.101	5.438	1.031
TOTAL	4.313	**34.629**	11.045	4.640	2.363	5.624	1.001

XBW: parser in text mode using words, Kao's algoritm for BWT
Gzip: gzip -9; Rar: rar -m5; Bzip2: bzip2 -9; ABC: abc -9; Szip: szip -b41;
PPMonstr ppmonstr -o1 -m256;

Table 8.14: Comparison of decompression speed

Chapter 9

Conclusion

This thesis is devoted to the text compression, specifically to a new approach called syllable-based compression. Common universal compression methods decompose the text into single characters or n-grams of these characters and use them to compress the file. For the text compression, compression methods often decompose the document into sequences of words (sequences of alphanumerical characters) and non-words (sequences of the remaining characters) and call these new strings an alphabet, over which they perform the compression. In this work we examine the possibility of further dividing the words into syllables and compressing the whole text over the alphabet of syllables. At the first sight, this approach might not seem entirely logical and one reviewer of a prestigious conference has written us that, due to the fact that a syllable is shorter than a word, the compression ratio achieved cannot be better than using word-based compression. It is obviously true, due to the fact, that models used in both categories of methods are different. In practice, the application environment determines which method, or which category of methods is usable. Consequently, it is impossible to call one approach better than the other, since different situations may require both word-based and syllable-based compressions. One of the reasons may be the fact that word-based methods use larger model than syllable-based methods. The biggest part of the model is mostly the set of used words or syllables. We have to encode the set of used words or syllables and transmit this set between the encoder and decoder - in case of word-based compression, the code for this set is greater in size [5].

During our research, we have discovered that the usefulness of word-based or syllable-based compression depends on many factors: the language of the document, its size and also the chosen compression method. In general, it is important whether the document contains a large amount of words or syllables which are used only rarely, such words and syllables disproportion-

ally decrease the efficiency of the compression of the whole alphabet. Both syllable-based and word-based compression is, according to our measurements, usually significantly more efficient than character-based compression for all observed compression methods in all file sizes, with the exception of the smallest files.

Conditions Advantageous for Syllables

Advantageous for syllable-based methods and in turn disadvantageous for word-based compression methods are the following conditions:

1. The language of the document has a rich morphology, its words have many different forms. In the document, there will be a large amount of words that have been used only once and encoding such dictionary would then be inefficient (as in Czech). The opposite is English, where words usually have a small amount of linguistic forms.

2. The language creates new words by compounding (as in German) or by adding prefixes and suffixes (Czech, German), which leads to a large amount of unique words, used only once across the document. The opposite is English, where prepositions are used to change the meaning of words.

3. The file being compressed is small in size. The smaller the size of the file, the greater the probability that a large amount of words will be used only once across the document.

4. The used methods are dictionary compression methods with static initialization (LZWL), statistical methods with a higher order of context (PPM) or BWT. In statistical methods of the first order and dictionary methods without initialization, however, words are more efficient. When we consider compression speed too, words have twice worse compression speed than syllables for PPM and twice better compression speed than syllables for BWT.

Compression methods can be measured according many criteria. The most of which are compression radio, compression and decompression speed, space used during compression and decompression. Compression methods can be tested on various types of files, in case of text compression methods it can be on files of different languages and sizes. We can say that syllable-based versions of the given compression methods are more successful for some criteria and some types of files than word-based or character-based versions.

Syllable-based versions are outperformed by word-based or character-based versions for different combinations of criteria and types of files. The importance of each criterion is changing in time and for each user can be different.

9.1 The Assets of the Thesis

In this thesis, we have defined several compression algorithms that were capable of compression over alphabets of syllables, words and sometimes even n-grams. These methods can be used for compression of pure text, XML and even XML containing errors (*non-well-formed XML*). The following list offers their overview and brief description, including interesting results. Each compression method was tested using vast collections of data (tens of thousands of files) in various languages (English, Czech, German, Slovenian) and of different sizes (5 KB to 20 MB).

LZWL A dictionary compression method based on LZW method. LZWL works over an alphabet of words and syllables. This method shown as the first that, in the case of Czech language, syllable-based version of some method can achieve a much higher compression ratio than word-based version of the same method with the same settings for some type of files.

HuffSyll A statistical compression method which is mostly first order, but use some information from the second order of context. It works over an alphabet of syllables and words and uses static initialization with a characteristic words and syllables sets. For very small files, it can achieve slightly better compression ratio than bzip2.

XMillSyl, XMLSyl Our early developed two methods specializing on a combination of well-formed XML and text compression. These methods were focused on small files, but the results achieved were not too convincing. We have discovered a problem with the necessity of real data compression, which often consists of non-well-formed XML, which is not able to compress by existing XML compression methods.

XBW A very successful method intended for large non-well-formed XML files. Based on Burrows-Wheeler transformation, it combines XML and text compression and works over alphabets of words, characters, and syllables. Syllable- and word-based versions have a compression ratio almost twice better than bzip2 when compressing files 20 MB large. The word-based version also consumes only twice the amount of time bzip2 does in files of this size.

TD1, TD2, TD3 Methods for the compression of a set of strings (syllables, words) based on trie compression, used, e. a., in XBW. In case of syllables, they are more efficient than other methods compared (including bzip2); in case of words, they are more successful for larger files.

9.2 Future Work

The most important findings in the area of syllable-based compression were already discovered and published, but there is a promising field of research for XML and common file compression.

According to our, not yet published, findings, specific algorithms for decomposing words into syllables, using knowledge of a specific language, can slightly improve the compression ratio in very small files. Their use, however, is limited to very large collections of small files, for which the implementation of the algorithm and the creation of its characteristic syllables dictionary would be profitable.

We have managed to slightly improve the HuffSyllable method and its adaptation to arithmetic encoding. The resulting method, called AritSyll, is being prepared for publishing.

For the methods for the compression of a set of strings, we have not yet published a set of improvements called TD4A through TD4H, which replace Elias's codes delta and gamma with static and adaptive Huffman encoding, effectively encode nodes with just one child and also effectively store the bit map for the information whether a certain node represents an element of the dictionary.

XBW is a powerful program. Due to the possibility of combining various compression methods, many compression methods can be added and more interesting results can be published in the future. There is one example: the statistical lossless compression method MSC [20] has the best results when MSC is used the combination BWT + MTF + MSC.

List of Publications

[1] Lánský, J.: Slabiková komprese. Master thesis, Faculty of Mathematics and Physics, Charles University, Prague, (2005). In Czech.

[2] Lánský, J., Žemlička, M.: Text Compression: Syllables. In: Richta, K., Snášel, V., Pokorný, J. (Eds.): Proceedings of the Dateso 2005 Annual International Workshop on DAtabases, TExts, Specifications and Objects. CEUR-WS, Vol. 129, (2005) 32–45.

[3] Lánský, J., Žemlička, M.: Compression of Small Text Files Using Syllables. In: Storer, J. A., Cohn, M. (Eds.): Proceedings of 2006 IEEE Data Compression Conference, IEEE Computer Society Press, Los Alamitos, California, USA (2006) 458.

[4] Lánský, J., Žemlička, M.: Compression of Small Text Files Using Syllables. Technical report no. 2006/1. Department of Software Engeneering, Faculty of Mathematics and Physics, Charles University, Prague (2006).

[5] Lánský, J., Žemlička, M.: Compression of a Dictionary. In: Snášel, V., Richta, K., and Pokorný, J. (Eds.): Proceedings of the Dateso 2006 Annual International Workshop on DAtabases, TExts, Specifications and Objects. CEUR-WS, Vol. 176, (2006) 11-20.

[6] Chernik, K., Lánský, J., Galamboš, L.: Syllable-based compression for XML documents. In: Snášel, V., Richta, K., and Pokorný, J. (Eds.): Proceedings of the Dateso 2006 Annual International Workshop on DAtabases, TExts, Specifications and Objects. CEUR-WS, Vol. 176, (2006) 21-31.

[7] Galamboš, L., Lánský, J., Chernik, K.: Compression of Semistructured Documents. In: International Enformatika Conference IEC 2006, Enformatika, Transactions on Engineering, Computing and Technology, Volume 14, (2006) 222–227.

[8] Lánský, J., Galamboš, L., Chernik, K.: Komprese webového uložiště. In: Vojtáš, P. (Ed.): ITAT 2006, Chata Kosodrevina, Bystrá dolina, Nízké Tatry, Sep 26-Nov 1, (2006) 107–112. In Czech.

[9] Lánský, J.: Slabiková komprese textových dat. In: Vojtáš, P., Skopal, T. (Eds.): Proceedings of the Annual Database Conference DATAKON 2006, Santon Hotel, Brno, (2006) 209–218. In Czech.

[10] Galamboš, L., Lánský, J., Žemlička, M., Chernik, K.: Compression of Semistructured Documents. Internation Journal of Information Technology (IJIT), Volume 4, Number 1, (2007) 11–17.

[11] Lánský, J., Žemlička, M.: Compression of a Set of Strings. In: Storer, J. A., Marcellin, M. W. (Eds.): Proceedings of 2007 IEEE Data Compression Conference, IEEE Computer Society Press, Los Alamitos, California, USA (2007) 390.

[12] Lánský, J., Chernik, K., Vlčková, Z.: Comparison of Text Models for BWT. In: Storer, J. A., Marcellin, M. W. (Eds.): Proceedings of 2007 IEEE Data Compression Conference, IEEE Computer Society Press, Los Alamitos, California, USA (2007) 389.

[13] Lánský, J., Chernik, K., Vlčková, Z.: Syllable-based Burrows-Wheeler Transformation. In: Pokorný, J., Snášel, V., and Richta, K. (Eds.): Proceedings of the Dateso 2007 Annual International Workshop on DAtabases, TExts, Specifications and Objects. CEUR-WS, Vol. 235, (2007) 1–10.

[14] Kuthan, T., Lánský, J., Genetic algorithms in syllable based text compression. In: Pokorný, J., Snášel, V., and Richta, K. (Eds.): Proceedings of the Dateso 2007 Annual International Workshop on DAtabases, TExts, Specifications and Objects. CEUR-WS, Vol. 235, (2007) 21–34.

[15] Lánský, J., Šesták, R., Uzel, P., Kovalčin, S., Kumičák, P., Urban, T., Szabó, M.: XBW - Word-based compression of non-valid XML documents. `http://xbw.sourceforge.net/` as visited on 6th June 2007.

[16] Šesták, R., Lánský, J., Uzel, P.: Kompresia konkatenovaných webových stránok pomocou XBW. In: Vojtáš, P. (Ed.): ITAT 2007, Polana, Slovakia, (2007) 85–90. In Slovak.

[17] Lánský, J., Žemlička, M.: Vliv nastavení slovníku na účinnost komprese malých souborů. In: Vojtáš, P. (Ed.): ITAT 2007, Polana, Slovakia, (2007) 75–81. In Czech.

[18] Šesták, R., Lánský, J.: Compression of Concatenated Web Pages Using XBW. In: Geffert, V. et al. (Eds.): SOFSEM 2008, LNCS 4910, Springer-Verlag Berlin, Heidelberg, Germany, (2008) pg. 743-754.

[19] Šesták, R., Lánský, J., Žemlička, M.: Suffix Array for Large Alphabet. In: Storer, J. A., Marcellin, M. W. (Eds.): Proceedings of 2008 Data Compression Conference, IEEE Computer Society Press, Los Alamitos, California, USA, (2008) pg. 543.

[20] Kochánek, J., Lánský, J., Uzel, P., Žemlička, M.: Multistream Compression. In: Storer, J. A., Marcellin, M. W. (Eds.): Proceedings of 2008 Data Compression Conference, IEEE Computer Society Press, Los Alamitos, California, USA, (2008) pg. 527.

[21] Kochánek, J., Lánský, J., Uzel, P., Žemlička, M.: The New Statistical Compression Method: Multistream Compression. In: Proceedings of 2008 International Conference on the Applications of Digital Information and Web Technologies (ICADIWT), IEEE Computer Society Press, Los Alamitos, California, USA, (2008). In press.

Bibliography

[22] Abel, J.: ABC http://www.data-compression.info/ABC as visited on 14th June 2008.

[23] Abel, J., Teahan, W.: Universal Text Preprocessing for Data Compression. In: IEEE Transactions on Computers, 54 (5), pp. 497-507, (2005)

[24] Adiego, J., Fuente, P.: On the Use of Words as Source Alphabet Symbols in PPM. In: Storer, J. A., Cohn, M. (Eds.): Proceedings of 2006 IEEE Data Compression Conference, IEEE Computer Society Press, Los Alamitos, California, USA (2006) 435.

[25] Aoe, J., Morimoto, K., Shishibori, M., Park, and K. H.: A trie compaction algorithm for a large set of keys. In: IEEE Transactions on Knowledge and Data Engineering, 08(3), 476–491, (1996).

[26] Bell, T., Powell, M.: The Canterbury corpus. http://corpus.canterbury.ac.nz as visited on 26th February 2007.

[27] Bell, T., Witten, I. H., Cleary, J.G.: Modeling for Text Compression. In: ACM Computing Surveys, Vol. 21, No. 4, (1989) 557–591.

[28] Bentley, J. L., Sleator, D. D., Tarjan, R. E., Wei, V. K.: A locally adaptive data compression scheme. In: Communications of the ACM, 29(4), (1986) 320–330.

[29] Burrows, M., Wheeler, D. J.: A Block Sorting Lossless Data Compression Algorithm. Technical report, Digital Equipment Corporation, Palo Alto, CA, U.S.A (2003).

[30] California Law. http://www.leginfo.ca.gov/calaw.html as visited on 2nd February 2007.

[31] Chapin, B., Tate, S.R.: Higher Compression from the BurrowsWheeler Transform by Modified Sorting. In: Storer, J. A., Cohn, M. (Eds.):

Proceedings of 1998 IEEE Data Compression Conference, IEEE Computer Society Press, Los Alamitos, California, USA (1998) 532.

[32] Cheney, J.: Compressing XML with Multiplexed Hierarchical PPM Models. In: Storer, J. A., Cohn, M. (Eds.): Proceedings of 2001 IEEE Data Compression Conference, IEEE Computer Society Press, Los Alamitos, California, USA (2001) 163.

[33] Cleary, J.G., Witten, I.H.: Data compression using adaptive coding and partial string matching. In: IEEE Transactions on Communications, COM-32 (4), (1984) 396-402.

[34] Claesson, V. Poledna, S. Soderberg, J.: The XBW model for dependable real-time systems. In: Proceedings of the Parallel and Distributed Systems Conference (1998) 130–138.

[35] Cook, R. P.: Heuristic compression of an English word list. In: Software - Practice and Experience 35(6), (2005) 577–581.

[36] Cover, T., M., Thomas, J., A.: Elements of Information Theory. 2nd edition, Wiley, (2006).

[37] Ciura, M., Deorowicz, S.: How to squeeze a lexicon. In: Software - Practice and Experience 31(11), (2001) 1077–1090.

[38] Dvorský, J. Word-Based Compression Methods for Information Retrieval Systems. Doctoral Thesis, Faculty of Mathematics and Physics, Charles University, Prague (2003).

[39] Dvorský, J., Pokorný, J., Snášel, V.: Word-based Compression Methods for Large Text Documents. In: Storer, J. A., Cohn, M. (Eds.): Proceedings of 1999 IEEE Data Compression Conference, IEEE Computer Society Press, Los Alamitos, California, USA (1999) 523.

[40] Elias., P.: Universal codeword sets and representation of the integers. In: IEEE Trans. on Information Theory, 21(2), (1975) 194–203.

[41] eKnihy: `http://go.to/eknihy` as visited on 2nd February 2007.

[42] Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Structuring labeled trees for optimal succinctness, and beyond. In: Proc. 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05), (2005) 184-193.

[43] Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and searching XML data via two zips. In: Proceedings of the 15th International Conference on World Wide Web (2006): 751–760.

[44] Gailly, J. L.: The gzip home page. `http://www.gzip.org/` as visited on 26th February 2007.

[45] Galamboš, L.: EGOTHOR. `http://www.egothor.org/` as visited on 6th February 2007.

[46] Gao, Y.: Population Size and Sampling Complexity in Genetic Algorithms. In: Proceedings of the Bird of a Feather Workshops (GECCO) – Learning, Adaptation and Approximation in Evolutionary Computation, (2003).

[47] Goldberg, D. E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Pub. Co. (1989).

[48] Hamming, R. W.: Coding and Information Theory. Prentice-Hall, Englewood Cliffs, NJ, (1986).

[49] Heinz, S., Zobel J., Williams, H.E.: Burst tries: a fast, efficient data structure for string keys. In: ACM Trans. Inf. Syst. 20(2), (2002) 192–223.

[50] Horspool, R. N.: Improving LZW. In: Storer, J. A., Reif, J., H. (Eds.): Proceedings of 1991 IEEE Data Compression Conference, IEEE Computer Society Press, Los Alamitos, California, USA (1991) 332–341.

[51] Horspool, R. N., Cormack, G. V.: A general purpose data compression technique with practical applications. Proceedings of the CIPS Session 84, (1984) 138-141.

[52] Horspool, R. N., Cormack, G. V.: Constructing WordBased Text Compression Algorithms. In: Storer, J. A., Cohn, M. (Eds.): Proceedings of 1992 IEEE Data Compression Conference, IEEE Computer Society Press, Los Alamitos, California, USA (1992) 62–71.

[53] Howard, P. G.: The design and analysis of efficient lossless data compression systems. Technical Report CS9328, Brown University, Providence, Rhode Island, (1993).

[54] Huffman, D. A.: A method for the construction of minimum-redundancy codes. In: Proceedings of the Institute of Radio Engineers, 40 (9), (1952) 1098-1101.

[55] Isal, R. Y. K., Moffat, A.: Word-based Block-sorting Text Compression. In: Proceedings of the 24th Australasian conference on Computer science, Gold Coast, Queensland, Australia, (2001) 92–99.

[56] Jones, D. W.: Application of splay trees to data compression. In: Communications of the ACM, vol. 31, number 8, ACM Press (1988) 996-1007.

[57] Kao, T. H.: Improving suffix-array construction algorithms with applications. Master Thesis. Gunma University, Japan, (2001).

[58] Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Proceedings of 13th International Conference on Automata, Languages and Programming. Springer, (2003) 943–955.

[59] Knuth, D. E.: Dynamic Huffman Coding. In: Journal of Algorithms, Vol. 6, (1985) 163–180.

[60] Knuth, D. E.: The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley, third edition, (1997).

[61] Kosek, J.: Inteligentní podpora navigace na WWW s využitím XML. `http://www.kosek.cz/diplomka/` as visited on 16th January 2007.

[62] Korodi, G., Rissanen, J., Tabus, I.: Lossless Data Compression Using Optimal Tree Machines. In: Storer, J. A., Cohn, M. (Eds.): Proceedings of 2005 IEEE Data Compression Conference, IEEE Computer Society Press, Los Alamitos, California, USA (2005) 348–357.

[63] Kuřík, O.: Komprese slovníku. (2007). Bachelor thesis, Faculty of Mathematics and Physics, Charles University, Prague, (2007). In Czech.

[64] League, C., Eng, K.: Type-Based Compression of XML Data. In: Storer, J. A., Marcellin, M. W. (Eds.): Proceedings of 2007 IEEE Data Compression Conference, IEEE Computer Society Press, Los Alamitos, California, USA (2007) 273–282.

[65] Lelewer, D. A., Hirschberg, D. S.: Data Compression. ACM Computing Surveys , Vol. 19, No. 3, (1987).

[66] Liefke, H., Suciu, D.: XMill: an Efficient Compressor for XML Data. In: Proceedings of ACM SIGMOD Conference (2000) 153–164.

[67] Mahoney, M. V., Ratushnyak, A.: PAQAR 4.0 2004. `http://www.cs.fit.edu/~mmahoney/compression/` as visited on 22nd May 2008.

[68] Maly, K.: Compressed tries. In: Commun. ACM, 19(7): 409–415, (1976).

[69] Megginson, D.: SAX: A Simple API for XML. `http://www.saxproject.org` as visited on 22nd January 2006.

[70] Melichar, B., Pokorný, J.: Data Compression. Revised version of the report DC-92-04. Department of Computers, Czech Technical University, (1994).

[71] Min, J. K., Park, M. J., Chung, C. W.: XPRESS: A Queriable Compression for XML Data. In: Proceedings of ACM SIGMOD Conference 2003, San Diego, CA, USA (2003) 122–133.

[72] Moffat, A.: Implementing the PPM data compression scheme. In: Proceedings of the IEEE Transactions on Communications, 38 (11), (1990) 1917-1921.

[73] Moffat, A.: Word based text compression. In: Software-Practice and Experience 19 (2), (1989) 185–198.

[74] Moffat, A., Neal, R. M., Witten, I. H.: Arithmetic Coding Revisited. In: ACM Transactions on Information Systems, Vol. 16, (1998) 256-294.

[75] Moffat, A., Turpin, A.: On the implementation of minimum redundancy prefix codes. in: IEEE Trans.Comm. 45(1997), 1200–1207

[76] Moura, E. S., Navarro, G., Ziviani, N., Baeza-Yates, R.: Fast searching on compressed text allowing errors. In: Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval. ACM Press, New York, (1998) 298-306.

[77] Ng., W., Lam, W. Y., Cheng, J.: Comparative Analysis of XML Compression Technologies. In: Word Wide Web: Internet nad Web Information Systems, Vol. 9, (2006) 5–33.

[78] O'Neill, E. T., Lavoie, B. F., Bennett, R.: Trends in the Evolution of the Public Web: 1998–2002. D-Lib Magazine, Vol. 9, No. 4 (2003).

[79] Nelson, M., Gailly, J.: The Data Compression Book. M&T Books, New York, NY (1995).

[80] Pilgrim, M.: What Is RSS. http://www.xml.com/pub/a/2002/12/18/dive-into-xml.html as visited on 16th January 2007.

[81] The Prague Dependency Treebank. http://ufal.mff.cuni.cz/pdt/ as visited on 16th January 2007.

[82] Project Gutenberg: http://www.promo.net/pg as visited on 17th January 2006.

[83] Rein, S., Gühmann, C., Fitzek, F.: Compression of Short Text on Embedded Systems. Journal of Computers, Vol. 1, No. 6, (2006).

[84] Rissanen, J.: A Lossless Data Compression System. IEEE Transactions on Information Theory, Vol. IT-29, No. 5, 656-664, (1983).

[85] Ristov, S.: LZ trie and dictionary compression. In: Software - Practice and Experience, 35(5), (2005) 445–465.

[86] Rodeh, M., Pratt, V. R., Even, S: Linear algorithm for data compression via string matching. In: Journal ACM 28, 1 (Jan.), (1981) 16–24.

[87] Ryabko, B. Y.: Data compression by means of a book stack. Prob. Inf. Transm., 16(4), (1980). In Russian.

[88] Salomon, D.: Data Compression - The Complete Reference. Springer-Verlag, 2nd edition, (2002).

[89] Sayood, K.: Introduction to Data Compression. Morgan Kaufmann, (1996).

[90] Schindler, M.: szip homepage http://www.compressconsult.com/szip/ as visited on 13rd June 2007.

[91] Seward, J.: The bzip2 and libbzip2 official home page. http://sources.redhat.com/bzip2/ as visited on 3rd February 2007.

[92] Seward, J.: On the Performance of BWT Sorting Algorithms. In: Storer, J. A., Cohn, M. (Eds.): Proceedings of 2000 IEEE Data Compression Conference, IEEE Computer Society Press, Los Alamitos, California, USA (2000) 173.

[93] Shannon, C. E.: A Mathematical Theory of Communication. In: Bell System Technical Journal, 27:3 (1948), 623–656.

[94] Shkarin, D.: Durilca Light and Durilca 0.4b. http://compression.graphicon.ru/ds/ as visited on 3rd February 2007.

[95] Shkarin, D.: PPM: one step to practicality. In: Storer, J. A., Cohn, M. (Eds.): Proceedings of 2002 IEEE Data Compression Conference, IEEE Computer Society Press, Los Alamitos, California, USA (2002) 202-211.

[96] Shkarin, D.: PPMd and PPMonstr, var. I `http://compression.graphicon.ru/ds/` as visited on 3rd February 2007.

[97] Skibinski, P.: Reversible data transforms that improve effectiveness of universal lossless data compression. Doctor of Philosophy Dissertation, University of Wroclaw, (2006).

[98] Skibinski, P.: Two-level directory based compression. In: Storer, J. A., Cohn, M. (Eds.): Proceedings of 2005 IEEE Data Compression Conference, IEEE Computer Society Press, Los Alamitos, California, USA (2005) 481.

[99] Spears W. M., De Jong K. A.: An Analysis of Multi-Point Crossover. In: FGA, (1991) 301–315.

[100] Storer, J., Szymanski, T. G.: Data compression via textual substitution. Journal of the ACM 29, (1982) 928-951.

[101] Sueddeutsche. `http://www.sueddeutsche.de` as visited on 3rd February 2007.

[102] Sundaresan, N., Moussa, R.: Algorithms and Programming Models for Ecient Representation of XML for Internet Applications. In: Proceedings of the 10th International WWW Conference, (2001) 366–375.

[103] Šesták, R.: Suffix Arrays for Large Alphabet. Master Thesis, Charles University in Prague (2007)

[104] Teahan, W.: Modelling English text. Ph.D. dissertation, University of Waikato, New Zealand, (1998).

[105] Thai Open Source Software Center Ltd.: Expat XML Parser. `http://expat.sourceforge.net` as visited on 16th January 2007.

[106] The American Heritage® Dictionary of the English Language, Fourth Edition. Houghton Mifflin Company, 2004. `http://dictionary.reference.com/browse/syllable` as visited on 9th January 2007.

[107] Thomas, S. W., McKie, J., Davies, S., Turkowski, K, Woods, J. A., Orost, J. W. Compress (version 4.0) program and documentation, (1985).

[108] Tolani, P., Haritsa, J.R.: XGrind: A Query-friendly XML Compressor. In: Proceedings of the 18th International Conference on Data Engineering (2002) 225.

[109] Toman, V.: Compression of XML Data. Master thesis, Faculty of Mathematics and Physics, Charles University, Prague (2003).

[110] Üçolük G., Toroslu H.: A Genetic Algorithm Approach for Verification of the Syllable Based Text Compression Technique. Journal of Information Science, Vol. 23, No. 5, (1997) 365–372.

[111] Walsh, N., Muellner, L.: DocBook: The Definitive Guide. `http://www.docbook.org/` as visited on 17th February 2007.

[112] Welch, T. A.: A technique for high performance data compression. In: IEEE Computer, 17:6 (1984), 819.

[113] Witten, I. H., Bell, T. C.: Canterbury text compression corpus. `http://corpus.canterbury.ac.nz` as visited on 29th January 2007.

[114] Witten, I. H., Moffat, A., Bell, T. C.: Managing Gigabytes: Compressing and Indexing Documents and Images. Van Nostrand Reinhold, (1994).

[115] World Wide Web Consorcium: Extensive Markup Language (XML). `http://www.w3.org/XML/` as visited on 6th February 2007.

[116] World Wide Web Consorcium: HyperText Markup Language (HTML). `http://www.w3.org/MarkUp/` as visited on 6th February 2007.

[117] Ziv, J., Lempel, A.: A Universal Algorithm for Sequential Data Compression. In: IEEE Transactions on Information Theory 23(3), (1977) 337-342.

[118] Ziv, J., Lempel, A.: Compression of Individual Sequences via Variable-Rate Coding. In: IEEE Transactions on Information Theory 24 (5), (1978) 530-536.