University of Wrocław
Department of Mathematics and Computer Science
Institute of Computer Science

Przemysław Skibiński

Doctor of Philosophy Dissertation

Reversible data transforms that improve effectiveness of universal lossless data compression

Supervisor: Dr hab. Marek Piotrów

Wrocław, 2006

dedykowane moim rodzicom,
Mirosławie i Januszowi Skibińskim

(dedicated to my parents,
Mirosława and Janusz Skibiński)

Abstract

The subject of this dissertation are universal lossless data compression algorithms. It is a very important field of research as the data compression allows to reduce the amount of space needed to store data or to reduce the amount of time needed to transmit data. This dissertation presents lossless data compression algorithms as well as most of well-known nowadays reversible data transforms that improve effectiveness of lossless data compression algorithms.

The main contribution of this dissertation are two word-based textual preprocessing algorithms, which significantly improve the compression effectiveness of universal lossless data compression schemes on textual files. These algorithms have very high encoding and decoding speed, which is amortized by a better compression effectiveness. The computational complexity remains the same as these algorithms work in a linear time. Moreover, in practice they require less than 10 MB of memory.

Contents

List of Figures

List of Tables

1 Introduction

Data compression refers to reducing the amount of space needed to store data or reducing the amount of time needed to transmit data. The size of data is reduced by removing the excessive information. A reverse process to the compression is called the decompression.

Most of people are not aware of a fact that data compression is widely used in the contemporary world. There are many applications of data compression, starting from modem transmission, across internet HTTP protocol, mobile telephony (GSM), internet telephony (Voice-over-IP), ending on hardware implemented image formats (JPEG), audio formats (MP3, AC-3), and video formats (VideoCD, DVD Video, DivX).

Data compression can be *lossless*, what means that the compression processes is fully reversible and decompressed data is identical to the original data. Another family of compression algorithms is called *lossy* as these algorithms irreversibly remove some parts of data and only an approximation of the original data can be reconstructed. Lossy algorithms achieve better compression effectiveness than lossless algorithms, but lossy compression is limited to audio, images, and video, where some loss is acceptable.

The performance of data compression is measured with the use of three main criteria: compression effectiveness, complexity (hardware requirements, algorithmic complexity), and speed (time to compress/decompress). Data compression algorithms are instantly improved, and with growing computational power of computers and amount of available memory, more complicated and more effective data compression algorithms take place of old methods.

Textual compression is a subset of lossless data compression, which deals with texts in natural languages, for example, English. Texts in natural languages have specific structure. They can be divided into sentences, finished by a period, a question mark, or an exclamation mark. Each sentence consists of words that are separated from the neighboring words by space and/or punctuation marks. This property of texts is exploited by word-based textual compression, where alphabet consists of words in natural language.

The *preprocessing* (preliminary processing) is a process, which reversibly transforms a data into some intermediate form, which can be compressed more efficiently. Transformed data can be compressed with most of existing lossless data compression algorithms, with better compression effectiveness than achieved using an untransformed data. The reverse process has two stages: decompression using given compressor and a reverse preprocessing (*postprocessing*) transform.

The contribution of this dissertation:

- introducing two predictive compression algorithms, which improve state-of-art predictive compression methods,
- presenting a mode for non-textual data, which protects textual preprocessing algorithms from a significant loss on non-textual data,
- introducing a method of surrounding words with spaces, which improves compression effectiveness of predictive and BWT-based compression schemes,
- presenting two-level dictionaries, which improve compression effectiveness of lossless compression methods,
- introducing a method for recognizing multilingual text files,
- creating separate optimization for main classes of lossless data compression algorithms (LZ, BWT-based, PPM, and PAQ),

- creating multilingual corpus, which contains English, German, Polish, Russian, and French text files, for evaluating compression performance of lossless data compression algorithms.

This dissertation has a following structure. Chapter 2 is an introduction to lossless data compression. This chapter is created mainly for people unfamiliar with lossless data compression field. Although, in Chapter 2 experts can find description of PPMZ, PPMII, PPM with built-in models, and PAQ algorithms, which are not described or vaguely described in the literature. Chapter 2 presents universal lossless compression methods. These schemes are used in our experiments in Chapter 5 of this dissertation. Chapter 2 also introduces a PPM with built-in models and word-based algorithms, which are based on universal compression methods. These methods are direct competitors for our word-based textual preprocessing methods presented in Chapter 5. At the end of this chapter, we also provide short comparison of practical lossless compression schemes.

Chapter 3 introduces our two predictive compression algorithms, which improve state-of-art predictive methods presented in Chapter 2. We have designed two algorithms that improve compression effectiveness on highly redundant data. The first method (PPMEA [Sk06]) extends PPM alphabet to long repeated strings. The second method (PPMVC [SG04]) extends the PPM algorithm with string matching abilities similar to the one used by the LZ77 algorithm. The experiments show that PPMEA and PPMVC improve the average compression effectiveness for about 1% and 3% respectively. Some parts of this chapter were published by the author in References [Sk06, SG04].

Chapter 4 presents most of well-known nowadays reversible data transforms that improve effectiveness of universal data compression algorithms. The strongest emphasis is placed on texts in natural languages as this field is most developed and an improvement in the compression effectiveness is the biggest. For each kind of data, this kind of data is introduced, then specialized compression methods are briefly presented and finally, preprocessing methods are described.

Chapter 5 contains the main contribution of the dissertation. It presents our two preprocessing algorithms: Word Replacing Transform (WRT [SG$^+$05]) and Two-level Word Replacing Transform (TWRT [Sk05b]). WRT is a word-based textual preprocessing algorithm. WRT concerns on English language as it is the most popular language in the computer science and most of texts in natural languages are written in English. TWRT is an expansion of WRT. Comparing to its predecessor, TWRT uses several dictionaries. It divides files on various kinds and chooses for each file combination of two best suitable dictionaries, what improves the compression effectiveness in a latter stage. Moreover, TWRT automatically recognizes multilingual text files. Some parts of this chapter were published by the author in References [SG$^+$05, Sk05b].

At the end of Chapter 5, we present comparison of TWRT to our direct competitors—StarNT and PPM with built-in models. We also show results of our experiments with TWRT on main classes of lossless data compression algorithms (LZ, BWT-based, PPM, and PAQ). The experiments are performed on well-known corpuses: the Calgary corpus, the Canterbury corpus, and the large Canterbury corpus. As there is no well-known corpus with multilingual text files, therefore we have created our own multilingual corpus, which was used in the experiments. The obtained results are fully commented. Concluding, TWRT significantly improves the compression effectiveness of universal lossless data compression algorithms (from 5% to over 23%, depending on a kind of the compression algorithm). Moreover, TWRT has a very high encoding and decoding speed, which is amortized (especially on larger files) by a better compression effectiveness. It is confirmed using a coefficient of the transmission acceleration. The computational complexity remains the same as TWRT works in a linear time. Furthermore, TWRT requires only about 10 MB of memory, which is allocated before actual compression or after actual decompression. Our final conclusion is that TWRT significantly improves compression effectiveness of lossless data compression algorithms, while the compression and the decompression speed as well as the complexity remain almost the same.

Recently, TWRT was successfully joined with the newest PAQ version (PAQAR 4.0 [MR04a]) into PASQDA [MR$^+$06]. According to many independent sources (Lossless data compression software benchmarks [Be05], Ultimate Command Line Compressors [Bo05]), it is one of the best currently available compressors.

Chapter 6 contains a discussion of obtained results and interesting possibilities for the further research. Some interesting details are presented at the end of this dissertation. Appendix A contains precise specification of files in the Calgary corpus. Precise specification of files in the Canterbury corpus and the large Canterbury corpus can be found in Appendix B. Appendix C contains precise specification of files in the multilingual corpus. Appendix D contains technical information about the usage of PPMVC computer program that implements the algorithm presented in this dissertation. Appendix E contains technical information about the usage of TWRT program that implements the proposed preprocessing algorithms. The detailed information of the compression programs used for the comparison in Chapter 5 can be found in Appendix F. Finally, Appendix G contains contents of CD included to this dissertation.

2 Lossless data compression

Data compression is a process by which data is changed into a more compact form. The size of data is reduced by removing the excessive information. In the compressed form, the data can be more efficiently stored or transmitted. A reverse process to the compression is called the decompression. A special computer program, which performs the compression and the decompression process, is called a *compressor* or an *archiver* (if it can compress more than one input file to single output file—*archive*).

Data compression methods can be classified in several ways. One of the most important criteria of classification is division into a lossless and a lossy compression. Lossless compression means that the compression process is fully reversible and decompressed data are identical to the original data. The compression algorithms that remove irreversibly some parts of data are called lossy as they can only reconstruct an approximation of the original data. This kind of algorithms is used in audio, pictures, and video data compression, when some loss is acceptable or even sometimes undetectable by human ear or eye. This dissertation deals only with lossless data compression.

Some lossless data compression methods belong to a group of *universal* compression algorithms, if these techniques are used to compress all kinds of data. There are many algorithms specialized only for one kind of data, for example GIF or PNG destined for images. In their field, specialized methods usually perform better than universal techniques. This dissertation presents data transform techniques that improve effectiveness of universal compression algorithms, so these algorithms are more competitive to specialized methods.

Data to be compressed consist of *symbols* over some *alphabet*. The input symbols may be bits, bytes (characters), words, pixels, etc. The compression effectiveness can be expressed in many units of measure. For example, the *compression ratio* is a ratio of the size of compressed data to the original size of data. To express the compression effectiveness in this dissertation we use well known unit of measure—*bits per character* (bpc). To get results in bpc, the formula

$$res = \frac{8 \cdot ods}{ids} \tag{2.1}$$

is calculated, where *ods* is the output (compressed) data size and *ifs* is the input data size. Lower bpc means a better compression effectiveness.

The compression effectiveness is not only one important criterion for a comparison of compression algorithms. The compression and decompression speed are also very important. A comparison of algorithms that take into consideration both, the compression effectiveness and the compression speed, is not a trivial task. Moreover, the compression and the decompression speed depend on a computer used to make experiments. To solve this problem Yoockin [Yo03] has defined a *total transmission time* (TTT):

$$TTT = t_c + t_t + t_d, \tag{2.2}$$

where: t_c – the compression time, t_t – the transmission time of the compressed file, t_d – the decompression time. A unit of measure for TTT is a second (s). A value of TTT is interpreted as an overall transmission time using the compression. The main disadvantage of this measure is a dependency from the size of a file. Swacha has proposed another measure, called a *coefficient of transmission acceleration* (CTA [Sw04]):

$$CTA = \frac{t_u}{TTT}, \tag{2.3}$$

where: t_u – the transmission time of the uncompressed file. A value of *CTA* is interpreted as how many times faster the data is transmitted with compression comparing to the transmission without the compression.

According to Rissanen and Langdon [RL81], we can distinguish two stages in lossless data compression: ***modeling*** and ***coding***. The modeling stage tries to find regularities, similarities, redundancies and other correlations. It transforms the input data into some intermediate structure or structures, which remove correlations. The modeling method is hard to choose as it depends on type of data to compress and is often different for lossless and lossy compression. In general, the coding stage is simpler than the first stage. It encodes structures obtained in the modeling stage. From a scientific point of view, a field of the coding is almost closed as the lower bound for coding methods has been already achieved. Nevertheless, there is—for example—possibility to improve the encoding and the decoding speed. Statistical coders are usually used as the coding methods. For example, Prediction by Partial Matching (PPM) algorithm in the modeling stage predicts probability distribution for a new symbol from the input data. In the coding stage, a new symbol is encoded with a predicted probability by an arithmetic encoder.

This chapter presents universal lossless compression methods, which are used in our experiments in this dissertation. It also presents PPM with built-in models and word-based algorithms, which are based on universal compression methods, and which are direct competitors for our word-based preprocessing methods presented in this dissertation. At the end, we provide short comparison of practical lossless compression schemes.

2.1 Statistical compression techniques

In the statistical coding, each symbol is encoded according to the probability that the symbol will occur with. Symbols that occur more frequently have assigned shorter codes, and less frequently used symbols have assigned longer codes. The statistical methods are rarely used as independent compression methods. They are usually used as the coding methods in two-stage lossless data compression (modeling and coding).

2.1.1 Entropy

Statistical coders in a modeling stage assign probabilities to input symbols and translate these probabilities to a sequence of bits in a coding stage. Shannon in 1948 [Sh48] established a source coding theorem, which describes relationship between the probabilities and output codes. This theorem claims that a symbol expected to occur with probability $P(x)$ is best represented in

$$\log_2 \frac{1}{P(x)} = -\log_2 P(x) \tag{2.4}$$

bits. According to this theorem, symbols that occur more frequently have assigned shorter codes, and less frequently used symbols have assigned longer codes.

The ***entropy*** of the probability distribution is a weighted average over all possible symbols, and therefore it is an expected length of an output code. It is defined as

$$H(X) = -\sum_{i=1}^{k} P(x_i)\ \log_2\ P(x_i)\,, \tag{2.5}$$

where k is count of all possible symbols and $P(x_i)$ is a probability of occurring of symbol x_i.

The entropy is a property of a model. For example, let us assume we have an alphabet {a, b, c, d} and symbols occur with the frequencies 4, 1, 3, and 2, respectively. The entropy H for this source equals $-0.4 \cdot \log_2(0.4) - 0.1 \cdot \log_2(0.1) - 0.3 \cdot \log_2(0.3) - 0.2 \cdot \log_2(0.2) = 0.529 + 0.332 + 0.521 + 0.464 = 1.8464$ bits/char. If we change the frequencies to a=7, b=1, c=1, and d=1, the entropy for this source changes to $-0.7 \cdot \log_2(0.7) - 0.1 \cdot \log_2(0.1) - 0.1 \cdot \log_2(0.1) - 0.1 \cdot \log_2(0.1) = 0.360 + 0.332 + 0.332 + 0.332 = 1.3567$ bits/char.

For a given probability distribution of the characters in English texts (based on a set of English textual files) the entropy equals about 4.5 bits/char. It is called *unconditional entropy* of the English language, as a probability of each symbol is independent of a previous symbol. When some advantage of relationships among adjacent or nearby symbols is taken, the entropy of English language is estimated to only 1.3 bits/char or even lower [Sh51].

2.1.2 Semi-adaptive Huffman coding

A *prefix code* has a property that code for no symbol is a prefix of the code for another symbol. The Huffman coding algorithm generates, from a set of probabilities, optimal prefix codes, which belongs to a family of codes with a variable codeword length. Prefix property of Huffman codes assures us that they can be correctly decoded despite being variable length. The Huffman coding algorithm [Hu52] is named after its inventor, David Huffman, who developed this algorithm as a student in a class on information theory at MIT in 1950 [Sa05].

The Huffman algorithm builds a prefix code on the binary alphabet {0,1}, which corresponds to a binary tree in which each inner node has a left and a right child, labeled '0' and '1' respectively. Leaves of the code tree are labeled by the input symbols. Each node has a weight, which is the frequency of the symbol's appearance. A path from a root of tree to each leave corresponds to Huffman code for each symbol. Picture 4) on Figure 2.1 presents binary tree created by Huffman algorithm. For example, the symbol 'd' corresponds to Huffman code '001' (left, left, right).

The Huffman algorithm is simple and can be described in terms of creating a Huffman code tree. The procedure for building this tree is following:

a) Start with a list of free nodes, where each node corresponds to each symbol.
b) Select two free nodes with the lowest frequency from the list.
c) Created a parent node for two nodes found in b) with a frequency equal to the sum of the two child nodes.
d) Remove the two nodes found in b) from the list of free nodes, and add the parent node created in c) to the list.
e) Repeat the process starting from b) until only a single tree remains.

Figure 2.1 illustrates an example of building the Huffman tree. The algorithm starts with a list of nodes 'a', 'b', 'c', and 'd', with frequencies 4, 1, 3, and 2 respectively, what one can see on the picture 1). In the next step, the algorithm selects two free nodes with the lowest frequency, this is, 'b' and 'd'. Then, it creates a parent node for these nodes, with cumulative frequency 3, what illustrates the picture 2). In the following step, there are free nodes with frequency 3, 3, and 4, so the algorithm selects two first nodes, what can be observed on picture 3). In the last step, there are two free nodes, which have a frequency 6 and 4. They are joined into a root of tree, which has cumulative frequency equal to 10.

For an alphabet of k symbols procedure of building the Huffman tree requires $k-1$ steps as complete binary tree with k leaves has $k-1$ inner nodes, and each step creates one inner node. If we use a heap as the list of free nodes, we can select nodes with the lowest frequency in $O(\log_2 k)$ time and the algorithm will run in $O(k \log_2 k)$ time.

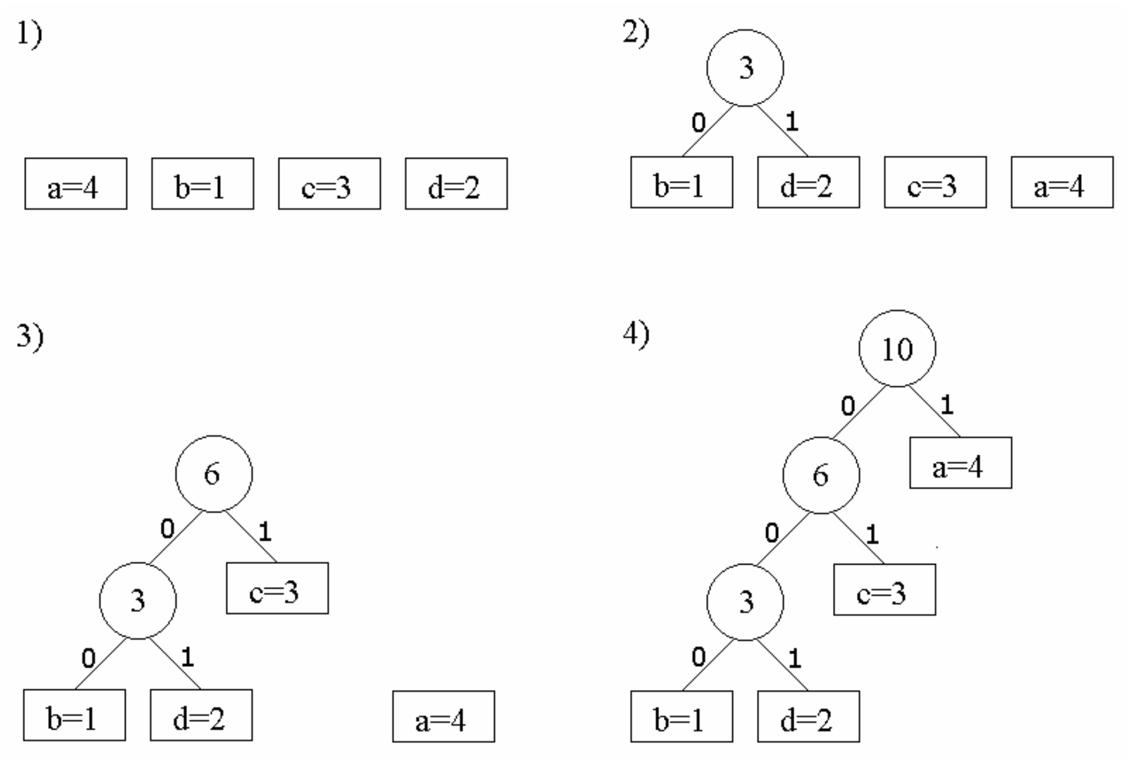

Figure 2.1: Example of building Huffman code tree

After building the Huffman tree, the algorithm creates a prefix code for each symbol from the alphabet by traversing the binary tree from the root to the node, which corresponds to the symbol. It records 0 for a left branch and 1 for a right branch. The Huffman algorithm, using the Huffman tree from Figure 2.1, assigns codes '1', '000', '01', and '001' to symbols 'a', 'b', 'c', and 'd' respectively.

Having Huffman codes for our model, we can encode, for example, the string 'acbdd'. It is encoded using 1 + 2 + 3 + 3 + 3 = 12 bits, what gives 2.4 bits/char. From previous subsection, we know that entropy for this model is equal to 1.84644 bits/char. Ineffectiveness comes from the fact that usually symbol x cannot be encoded in exactly $-\log_2 P(x)$ bits, unless $P(x)$ is a negative power of 2. In other words, the entropy for most symbols is usually a non-integer, but the Huffman coding uses codes of integer length.

Above-presented algorithm is called an ***semi-adaptive*** (or an *semi-static*) Huffman coding as it requires knowledge of frequencies for each symbol from alphabet. Moreover, the Huffman tree with the Huffman codes for symbols (or just frequencies of symbols, which can be used to create the Huffman tree) must be stored together with the compressed output. This information is needed by the decoder and it is usually placed in the header of a compressed file.

2.1.3 Adaptive Huffman coding

The semi-adaptive Huffman coding is not suitable to situations when probabilities of the input symbols are changing. It is very ineffective to use Huffman algorithm for building the tree and generating prefix codes after encoding each symbol from the input. In 1973 Faller has presented modified version of Huffman algorithm [Fa73], which manages with this problem. This algorithm is nowadays known as an *adaptive* Huffman coding.

The adaptive Huffman algorithm presents a different approach to building a Huffman tree, which introduces a concept known as the sibling property. This algorithm starts with an empty tree or a standard distribution. It adds to the tree a special control symbols identifying new symbols, which currently are not a part of the tree. The adaptive Huffman algorithm allows modifying the Huffman tree after encoding each symbol from the input. In this way, Huffman codes can be dynamically changed according to a change of probabilities of the symbols. Of course, the encoder and decoder must maintain the same Huffman tree.

Usually the adaptive Huffman algorithm produces code that is more effective then the semi-adaptive Huffman code. There is also no need to store the Huffman tree with the Huffman codes for symbols with the compressed output. On the other hand, adaptive Huffman algorithm has to update the Huffman tree after encoding each symbol, therefore is slower than semi-adaptive version. Moreover, the compression effectiveness at the beginning of the coding or for small files is low.

The semi-adaptive and the adaptive Huffman coding decrease redundancy in a data by the fact that distinct symbols have distinct probabilities of occurrence. Symbols with higher probabilities of occurrence have assigned shorter codes, while symbols with lower probabilities are encoded with longer codes. In practical applications, however, the adaptive Huffman coding is often replaced by easier and more effective arithmetic coding, described in the next subsection.

The Huffman coding is rarely used as an independent compression method. It is usually used as the coding method in the last stage of lossless data compression. Huffman compression is used in a connection with, for example, LZSS algorithm (used in gzip, PKZip, ARJ, and LHArc), BWT-based algorithms, JPEG [Wa91] and MPEG compression, Run-Length Encoding, Move-To-Front coding. Most of these algorithms are described in this dissertation.

2.1.4 Arithmetic coding

It is proven that the Huffman coding generates optimal codes of integer length. It means that this algorithm achieves the theoretical entropy bound if all symbol probabilities are negative powers of 2. The entropy for most of symbols is, however, usually a non-integer (more exact, $-\log_2 P(x)$ bits, where $P(x)$ is a probability of occurring of symbol x). This fractional length must be approximated by the Huffman algorithm with an integer number of bits. For example, if a probability of a symbol is 1/3, the optimal number of bits to code this symbol is about 1.585. The Huffman coding has to assign either 1 or 2 bit code for this symbol. In both cases, an average length of the output code is higher than theoretical entropy bound.

The arithmetic coding dispenses with the restriction that each symbol is encoded in an integer number of bits. It completely bypasses the idea of replacing an input symbol with a corresponding output code. Instead, it replaces a sequence of input symbols with a single output floating-point number.

The basic concept of the arithmetic coding was invented by Elias in the early 1960s [Ab63]. Elias, however, did not solve a problem with an arithmetic accuracy, which needs to be increased with the length of a sequence of input symbols. In 1976 Pasco [Pa76] and Rissanen [Ri76] proved simultaneously that finite-precision arithmetic could be used, without any loss of accuracy, to represent output data of the arithmetic coding scheme. The idea of the arithmetic coding that is generally known nowadays was independently presented in 1979 and 1980 by Rissanen and Langdon [RL79], Rubin [Ru79], and Guazzo [Gu80]. A summary of Rissanen and Langdon's work on field of the arithmetic coding can be found in the work of Langdon [La84]. A modern approach to the arithmetic coding is described in the work of Moffat et al. [MN$^+$98].

The basic idea of arithmetic coding is to represent input symbols by a single floating-point number. The algorithm starts with an interval of 0.0 (lower bound) and 1.0 (upper bound). The interval is divided into parts. Each symbol from the alphabet has assigned non-overlapping subinterval with a size proportional to its probability of occurrence. The order of assigning subintervals to symbols does not matter as long as it is done in the same manner by both, the encoder and the decoder. In the following step, an input symbol is encoded by selecting its subinterval. This subinterval becomes new interval, and it is divided into parts according to probability of symbols from the input alphabet. This process is repeated for each input symbol. At the end, any floating-point number within a final interval uniquely determines the input data. The pseudo code below illustrates the arithmetic coding process:

```
lowerBound = 0.0
upperBound = 1.0
while there are still symbols to encode
      get an input symbol
      currentRange = upperBound – lowerBound
      lowerBound = lowerBound + (currentRange * lowerBound of new symbol)
      upperBound = lowerBound + (currentRange * upperBound of new symbol)
end of while
output lowerBound
```

Figure 2.2 illustrates an example of the arithmetic encoding process. Our model uses an alphabet consist of symbols ‘a’, ‘b’, ‘c’, and ‘d’, with frequencies 4, 1, 3, and 2 respectively. We are encoding the string ‘acbdd’. At the beginning we have the interval [0.0, 1.0) (lower bound, upper bound). Next, we have the symbol ‘a’ to encode, and we select its subinterval. A new interval is [0.0, 0.4). After encoding the symbol ‘c’, this interval is narrowed to [0.0 + 0.4·0.5, 0.0 + 0.4·0.8] = [0.2, 0.32). Similarly, the next three symbols ‘b’, ‘d’, and ‘d’ narrow the interval to [0.248, 2.6), [0.2576, 2.6), and [0.25952, 2.6), respectively. In this way, we have encoded the string ‘acbdd’ into [0.25952, 2.6). Any number within this interval uniquely determines the input string, and we have selected lower bound, that is, 0.25952. As the count of encoded symbols grows, the interval needed to represent it becomes smaller, and the number of bits needed to specify the interval grows. It is worth to mention that the more likely symbols reduce the size of the interval by less than the unlikely symbols, therefore they add fewer bits to the output floating-point number.

The decoding process is basically the same as the encoding process, but instead of using the symbols to narrow the interval, we use given interval to select a symbol, and then narrow it. The main problem with decoding is that we do not know how many symbols are to decode or when to stop the decoder. This can be handled by encoding a special symbol EOF (End-of-File), which has very low probability and occurs only once in the file, or including the count of input symbols with the output number. The same problem applies to the Huffman coding, but to simplify, we passed over it.

The output from the arithmetic encoder is a very long floating-point number. From practical reasons, we must work on fixed-precision numbers, and the process of arithmetic coding seems completely impractical. Fortunately, there is technique for progressively transmitting an interval, which corresponds to input data, using fixed-precision arithmetic. Moreover, it turns out that, if we accept some effectiveness loss, we can even use fixed-precision integers for the arithmetic coding. The loss in the integer implementation of the arithmetic coding comes from round-off errors in division. The integer version is, however, faster than floating-point arithmetic coding. On the other hand, the loss is small and can be accepted.

Theoretically, the arithmetic coding reaches the same compression ratio as the unconditional entropy of the input data. In practice, however, using an integer arithmetic and scaling, to prevent overflow of the variables that store the symbol frequencies, makes the coding less efficient. Despite of this, the arithmetic coding achieves better compression effectiveness than the Huffman coding, but it requires more computation and it is slower. The arithmetic coding is most useful for adaptive compression, in which probabilities of input symbols may be different at each step. For a static and a semi-static compression, in which the probabilities of the input symbol are fixed, the Huffman coding is usually preferable to the arithmetic coding [BK93, MT97].

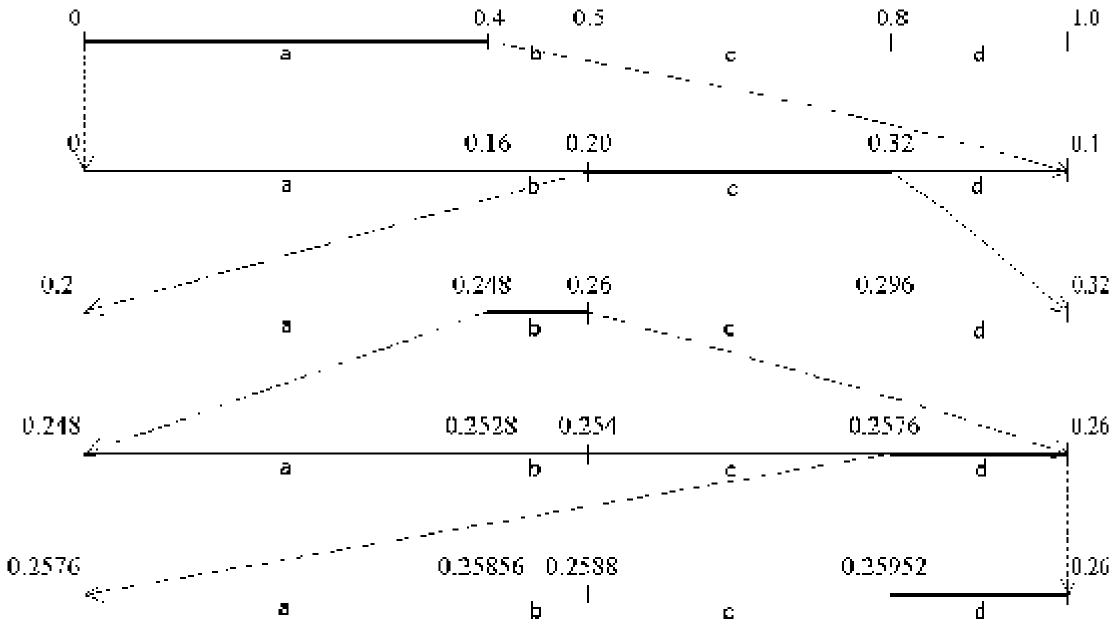

Figure 2.2: Example of arithmetic encoding process

The arithmetic coding, like the Huffman coding, is rarely used as an independent compression method. It is usually used as the coding method in the last stage of lossless data compression. The arithmetic coding is used in a connection with, for example, a PPM algorithm, BWT-based algorithms, JPEG-LS [WS$^+$00], JBIG, Run-Length Encoding, Move-To-Front coding. It is also used in the hardware implemented fax protocols CCITT Group 3 [IT80] and CCITT Group 4 [IT88], which use a small alphabet with an unevenly distributed probability. Most of these algorithms are described in this dissertation.

2.2 Dictionary-based compression techniques

In a dictionary compression, we make use of the fact that certain groups of consecutive characters occur more than once. These characters are replaced by a code, which points to some kind of dictionary, what reduces the size of data. For example, in the LZ77 algorithm, the dictionary is build as part of previously seen data.

In 1987, Bell [Be87] showed that there is a general algorithm for converting a dictionary method to a statistical one, and any practical dictionary compression scheme can be outperformed by a related statistical compression scheme. Dictionary-based compression techniques, however, are very fast at average compression ratio, what makes them very attractive from a practical point of view.

2.2.1 RLE

The Run-Length Encoding (RLE) is a very simple compression technique created especially for data with strings of repeated symbols (the length of the string is called a *run*). The main idea behind RLE-1 is to encode repeated symbols as a pair: the length of the string and the symbol. As one can see in Figure 2.3, the string 'abbaaaaabaabbba' of length 15 bytes (characters) is presented as 7 integers plus 7 characters, which can be easily encoded on 14 bytes (as for example '1a2b5a1b2a3b1a'). The biggest problem with RLE-1 is that in the worst case (for example random data) the size of output data can be two times longer than size of input data. To eliminate this problem, each pair can be later encoded (the lengths and the strings separately) with, for example, the Huffman coding.

RLE-2 is designed not to expand the size of data on data that does not contain strings of repeated symbols. It uses a special symbol to signal the start of the RLE-2 encoded sequence. Only strings of a length longer or equal to some fixed number, usually 3, are replaced by the special symbol and a pair (the length of the string and the repeated symbol). The special symbol can be selected as one of unused characters, but this idea needs additional preprocessing of data to find unused characters. Another idea is to use a fixed character. If the fixed characters occur in data, it is encoded with added flag (for example the length of the string equal to 0). As one can see in Figure 2.3, the string 'abbaaaaabaabbba' of length 15 bytes (characters) is presented as 7 characters plus 2 special symbols and 2 pairs, which can be easily encoded on 13 bytes (as for example 'abbX5abaaX3ba').

Method	Data to encode	Encoded data
RLE-1	abbaaaaabaabbba	<1,a>,<2,b>,<5,a>,<1,b>,<2,a>,<3,b>,<1,a>
RLE-2	abbaaaaabaabbba	abbX<5,a>baaX<3,b>a

Figure 2.3: Run-Length Encoding (RLE) example

RLE with its modifications are widely used, especially in well-known graphics files like: BMP images, PCX images, and TIFF images. RLE is often combined with other compression algorithms. RLE is used to reduce a number of zeros after Move-to-Front (MTF) encoding in some BWT-based (Burrows–Wheeler Transform) algorithms (for example, in Reference [Fe96]) and after quantization in JPEG [PM92] image compression.

2.2.2 LZ77

In 1977 Jacob Ziv and Abraham Lempel have presented their dictionary-based scheme [ZL77] for lossless data compression. This algorithm is nowadays known as LZ77 in honor to the authors and the publishing date. The reversal of the initials in the abbreviation is a historical mistake.

The LZ77 idea is to build a dictionary (a *sliding window*) from a part of previously seen data and to encode a remaining data (a *look-ahead buffer*) as a reference to the dictionary. The algorithm searches the sliding window for the longest match with the beginning of the look-ahead buffer and outputs a reference (a pointer) to that match. It is possible that there is no match at all, so the output cannot contain just pointers. In LZ77 the reference is always outputted as a triple: an offset to the match, a match length, and the next symbol after the match. If there is no match, the algorithm outputs a null-pointer (both the offset and the match length equal to 0) and the first symbol in the look-ahead buffer.

Figure 2.4 presents an example of the LZ77 encoding process. Let us assume that we have already encoded some part of input data, and there is only the string 'caaababbcabcc' left to encode. The algorithm searches the sliding window for a match, but there is no match, so LZ77 outputs a triple <0,0,c>. The longest match for the string ‘aaababbcabcc’ is ‘aa’, and the encoder outputs <7,2,a>. For the remaining string ‘babbcabcc’, there is the match ‘babbca’ outputted as <8,6,b>. Finally, we have the string ‘cc’, matched as ‘c’ and encoded as a triple <3,1,c>.

The data already encoded	The data to encode	The data encoded as
(sliding window)	(look-ahead buffer)	<offset, match length, next symbol>
…aababb	caaababbcabcc	<0,0,c>
…aababbc	aaababbcabcc	<7,2,a>
…aababbcaaa	babbcabcc	<8,6,b>
…aababbcaaababbcab	cc	<3,1,c>
…aababbcaaababbcabcc		

Figure 2.4: LZ77 encoding example
The longest match is underlined.

The values of an offset to a match and a match length must be limited to some maximum constants. The compression performance of LZ77 depends mainly on these constants. The offset is usually encoded on 12–16 bits, so it is limited from 0 to 4095–65535 symbols. Thus there is no need to remember more than 65535 last seen symbols in the sliding window. The match length is usually encoded on 8 bits, what gives maximum match length equal to 255.

The LZ77 encoding process is very fast, because usually one reference (a triple) is transmitted for several input symbols. Another important property of LZ77 is that the decoding is much faster than the encoding. Other algorithms need a similar time comparing compression and decompression time. Most of the LZ77 compression time is, however, used by searching the sliding window for the longest match, what is not performed during the decompression. In the LZ77 algorithm decompression is trivial as each reference is simply replaced with the string, which it points to.

2.2.3 LZSS

In 1982 James Storer and Thomas Szymanski have presented their LZSS algorithm [SS82], which is based on LZ77. LZSS is intended that the dictionary reference should be always shorter than the string it replaces. That was not always the case in the LZ77 algorithm. Sometimes, particularly at the beginning of the encoding, no match will be found for the symbol being encoded, and LZ77 solves this problem very ineffectively by outputting a null-pointer (both the offset and the match length equal to 0) and the first symbol in the look-ahead buffer. Moreover, the match may be short and the reference can consume more space than the string it replaces.

LZ77 requires that the next symbol after a matched string be always transmitted with each pointer and a match length. Storer and Szymanski observed that a better compression could be achieved by outputting this symbol only when necessary as it can be encoded as a part of the next match. They proposed to include an extra bit (a *bit flag*) at each coding step to indicate whether the outputted code represents a pair (a pointer and a match length) or a single symbol. This modification solves also a problem with ineffectively encoded null-pointers.

Storer and Szymanski also observed that matches shorter than h, where h is usually equal to 3, are more effectively encoded as explicit symbols. Moreover, unused match lengths from 0 to h–1 can increase the maximum match length. Therefore, if a match length is encoded for example on 8 bits, LZSS can use values 0–255 as equivalent of match length from *h* to 255+*h*.

Figure 2.5 presents an example of the LZSS encoding process. Let us assume that we have already encoded some part of input data, and there is only the string ‘caaababbcabcc’ left to encode. We can also assume that a bit flag 0 means that following code represents a single symbol and 1 represents a pair (a pointer and a match length). The algorithm searches the sliding window for a match, but there is no match, so LZSS outputs the bit flag 0 and the symbol ‘c’. The longest match for the string ‘aaababbcabcc’ is ‘aa’, which is shorter than the maximum match length and the encoder outputs the bit flag 0 and the symbol ‘a’. Then, the longest match for the string ‘aababbcabcc’ is ‘aababbca’ outputted as the bit flag 1 and the pair <8,8>. The remaining symbols are encoded separately using the bit flag 0.

Data already encoded	Data to encode	Encoded data
(sliding window)	(look-ahead buffer)	<bit flag, symbol or <offset, match length>>
…aababb	caaababbcabcc	<0,c>
…aababbc	aaababbcabcc	<0,a>
…aababbca	aababbcabcc	<1,<8,8>>
…aababbcaaababbca	bcc	<0,b>
…aababbcaaababbcab	cc	<0,c>
…aababbcaaababbcabc	c	<0,c>
…aababbcaaababbcabcc		

Figure 2.5: LZSS encoding example
The longest match is underlined.

In a popular compressor—gzip [Ga93]—an offset is encoded on 15 bits, a match length is encoded on 8 bits, and the minimum match length is equal to 3. With these assumptions, for encoding the string ‘aaababbcabcc’ from Figure 2.5, LZSS needs $5 \cdot 9$ bits (single symbols) plus $1 \cdot 24$ bits (the pair), what gives 69 bits. The same string encoded using the same assumptions and LZ77 needs $4 \cdot 31$ bits (triples), what is equal to as many as 124 bits.

LZSS is often combined with the statistical coding method. The symbols, the offsets, and the match lengths are effectively encoded using statistical techniques. For example, gzip, PKZip, ARJ, and LHArc apply the Huffman encoding, and LZARI uses the arithmetic coding. Of course, LZSS inherits from LZ77 high compression speed and very high decompression speed, what makes this scheme the most popular compression algorithm.

2.2.4 LZ78

In 1978 Jacob Ziv and Abraham Lempel have presented their second most-known dictionary-based scheme [ZL78], which is nowadays known as LZ78. Distinct from the LZ77 algorithm, LZ78 method does not build a dictionary from a part of previously seen data (a sliding window), but the dictionary is an independent structure, build adaptively during the encoding. Of course, both the encoder and the decoder must follow the same rules to ensure that they use an identical dictionary.

The LZ78 idea is very similar to LZ77. In each step the algorithm searches the dictionary for the longest match with the data to encode. LZ78 always outputs a pair: index addressing an entry of the dictionary and the next symbol after the match. There is no need to encode the match length, like in the LZ77 algorithm, as decoder knows it. Then, the dictionary is updated. The new dictionary entry, consisting of the match and the next symbol after the match, is added to the dictionary. It means that when a new entry is added, the dictionary already contains all the prefixes of the current match. The new dictionary entry gets the index, which is equal to number of entries in the dictionary. This entry will be available to the encoder at any time in the future, not just for the next few thousand symbols like in LZ77.

At the beginning of the encoding, the LZ78 dictionary contains only a single entry—the null-string. If there is no match (even a single symbol), what—for example—always happens at the beginning of the encoding, the algorithm outputs the null-string and the first symbol to encode. This is the reason, why the output must contain a symbol after an index to a dictionary entry, what is analogous to LZ77.

In LZ77 the value of the offset to a match is encoded on constant number of bits. LZ78 encodes indexes, whose maximal value grows with the size of the dictionary. When indexes become larger numbers, they require more bits to be encoded. It is unprofitable to let the dictionary to grow indefinitely, so the dictionary is periodically purged and encoding continues as if starting on a new text. It will be better explained in the next subsection.

Figure 2.6 presents an example of the LZ78 encoding process. The algorithm starts only with the null-string in the dictionary. It has the string ‘abaaabababb’ to encode, and there is no match, so the pair <0,a> (where 0 represents a null-string) is outputted. In the following step, there is also no match for the string ‘baaabababb’, and the pair <0,b> is encoded. The longest match for the remaining string ‘aaabababb’ is ‘a’, so the encoder outputs the pair <1,a> (where 1 represents ‘a’). Then, the longest match for the string ‘abababb’ is ‘a’, outputted as the pair <1,b>. In the next step, there is the string ‘ababb’ matched to ‘ab’, and encoded as <4,a> (where 4 represents ‘ab’). Finally, the longest match for the remaining string ‘bb’ is ‘b’, outputted as the pair <2,b> (where 2 represents ‘b’).

The dictionary	Data to encode	The data encoded as <index, next symbol>
0=null-string	abaaabababb	<0,a>
0=null-string; 1=a	baaabababb	<0,b>
0=null-string; 1=a; 2=b	aaabababb	<1,a>
0=null-string; 1=a; 2=b; 3=aa	abababb	<1,b>
0=null-string; 1=a; 2=b; 3=aa; 4=ab	ababb	<4,a>
0=null-string; 1=a; 2=b; 3=aa; 4=ab; 5=aba	bb	<2,b>
0=null-string; 1=a; 2=b; 3=aa; 4=ab; 5=aba; 6=bb		

Figure 2.6: LZ78 encoding example
The longest match is underlined.

The LZ78 encoding is fast, comparable to LZ77, what is the main advantage of dictionary-based compression. The LZ78 algorithm preserves important property of LZ77 that the decoding is faster than the encoding. The decompression in LZ78 is faster than the compression as there is no need to find the match with the data to decode and the dictionary. Each index is simply replaced with the string, which it points to, and the dictionary is updated.

2.2.5 LZW

In 1984 Terry Welch has presented his LZW (Lempel–Ziv–Welch) algorithm [We84], which is based on LZ78. Welch observed that in LZ78 there is no need to encode the next symbol after the match, if we initialize the dictionary with all possible symbols from the input alphabet. It guarantees that a match will always be found. LZW always outputs only an index addressing an entry of the dictionary. This solution simplifies the algorithm and improves the compression effectiveness as the next symbol after the match can be encoded as a part of the next match (the next index). This modification is analogous to transition from LZ77 to LZSS.

The LZW encoding is based on the LZ78 encoding. In each step the algorithm searches the dictionary for the longest match with the data to encode. LZW always outputs only an index addressing an entry of the dictionary and updates the dictionary. The new dictionary entry, consisting of the match and the next symbol after the match, is added to the dictionary. The new dictionary entry gets the index, which is equal to number of entries in the dictionary. The main difference between LZW and LZ78 is that the next symbol after the match becomes the beginning of the data to encode and will be a part of the next match.

In LZ78 and LZW, the maximal value of an index grows with a size of the dictionary. When indexes become larger numbers, they require more bits to be encoded. It is unprofitable to let the dictionary to grow indefinitely so the dictionary is periodically purged. There are several methods of purging the dictionary:

- Remove all dictionary entries except all symbols from the input alphabet when the dictionary reaches a certain size (GIF).
- Remove all dictionary entries except all symbols from the input alphabet when compression is not effective (Unix compress).
- Remove only least recently used entry when the dictionary reaches a certain size (BTLZ [LH^{+}94] – British Telecom Lempel Ziv).

Figure 2.7 presents an example of the LZW encoding process. Let us assume that we use character-based LZW. The dictionary is initialized with 256-character alphabet (values from 0 to 255). The algorithm has the string ‘abaaabababb’ to encode, and ‘a’ is encoded as 97. In the following step, the string ‘baaabababb’ in matched to ‘b’, outputted as 98. The longest match for the remaining string ‘aaabababb’ is ‘a’, and the encoder outputs 97. Then, the longest match for the string ‘aabababb’ is ‘aa’, outputted as the entry 258 (created in the previous step). In the next step, there is the string ‘bababb’ matched to ‘ba’, and encoded as 257. For the string ‘babb’, the longest match is ‘bab’, and the encoder outputs 260. Finally, the remaining string ‘b’ is outputted as the entry 98.

The dictionary	Data to encode	Encoded index
0–255=8 bit ASCII (particularly, a=97; b=98)	abaaabababb	97
0–255=8 bit ASCII (particularly, a=97; b=98); 256=ab	baaabababb	98
0–255=8 bit ASCII (particularly, a=97; b=98); 256=ab; 257=ba	aaabababb	97
0–255=8 bit ASCII; 256=ab; 257=ba; 258=aa	aabababb	258
0–255=8 bit ASCII; 256=ab; 257=ba; 258=aa; 259=aab	bababb	257
0–255=8 bit ASCII; 256=ab; 257=ba; 258=aa; 259=aab; 260=bab	babb	260
0–255=…; 256=ab; 257=ba; 258=aa; 259=aab; 260=bab; 261=babb	b	98

Figure 2.7: LZW encoding example
The longest match is underlined.

If we assume that LZW indexes use always 12 bit codes, then the maximal available value of an index is 4095. We must remember that values 0–255 are reserved for all symbols from the input alphabet. With these assumptions, LZW encodes the string ‘abaaabababb’ from Figure 2.7 on 7·12 bits (indexes), what gives 84 bits. The same string encoded using LZ78 needs 6·20 bits (pairs), what is equal to as many as 120 bits.

LZW is an important part of various data formats. The LZW algorithm is used by a well-known Unix compress program, ARC (archiver for MS-DOS), GIF images, TIFF images (optional), and Postscript (optional). LZW encoding is fast, comparable to LZSS, but LZSS combined with a statistical coding gives a better compression effectiveness. LZW is, however, a simple algorithm that can be easily implemented in hardware. It was originally developed for high-performance disk controllers.

2.3 Block-sorting compression algorithm

The block-sorting compression algorithm was developed by Michael Burrows and David J. Wheeler in 1994 [BW94]. The authors stated that this scheme does not belong to either dictionary or statistical methods, but we can assume that it is related to statistical methods.

There are several variations of block-sorting compression algorithm [Fe96, BK^{+}99, De00, De02, De03a], but the original version consists of three stages: the Burrows–Wheeler Transform (BWT), the Move-to-Front encoding (MTF) and the statistical encoding. The first stage exhibits the property that a particular symbol is likely to reappear in similar contexts (the context is a finite sequence of symbols preceding the current symbol). BWT permutes input data and returns a data with more and longer runs (strings of identical symbols) than found in the original data. The second stage transforms the permuted data into a sequence of integers, which represent the rank of recency. In the last stage, these integers are effectively encoded using the Huffman coding or the arithmetic coding. A good compression effectiveness is achieved mainly by the fact that the number of runs and their lengths are increased in BWT stage.

Distinct from most other compression algorithms, the block-sorting compression algorithm does not process the input data sequentially; therefore, the block sorting is an off-line algorithm. The encoder divides the input data into the blocks as all the data and additional data structures required for sorting it, might not fit in a memory. The encoder performs the compression separately for each block. Usually, larger blocks give a better compression effectiveness at the expense of compression speed and memory requirements. The size of the block for widely used bzip2 [Se02] is 900 kB.

BWT-based archivers achieve good compression effectiveness, significantly better than compressors from Lempel–Ziv family. Nevertheless, the compression speed, and particularly the decompression speed is lower than archivers from Lempel–Ziv family. BWT class is between dictionary-based compressors from Lempel–Ziv family and predictive compressors, described in the next section. It is applied to compression speed as well as compression effectiveness and memory requirements.

2.3.1 BWT

The Burrows–Wheeler Transform (BWT) is not actually a compression scheme. BWT is a reversible transform that converts the data into a format that is generally more compressible. The Burrows–Wheeler Transform forms an $n \cdot n$ matrix of all possible cyclic shifts of the input data (the block). Left matrix in Figure 2.8 is formed by the BWT algorithm for the input data ‘minimum’. Then, the matrix is sorted lexicographically and the last column of sorted matrix is taken as the output. Right matrix in Figure 2.8 is the same as left, but lexicographically sorted. Row 3 of this matrix, which is underlined, contains the input data. The BWT transformed version of the string ‘minimum’ corresponds to the last, bolded column of this matrix. To make the reverse transform possible, the number of row, where is the original data in sorted matrix, must be transmitted with the output string. This makes the transformed data even larger than its original form, but the transformed data is more compressible.

m	i	n	i	m	u	m
m	m	i	n	i	m	u
u	m	m	i	n	i	m
m	u	m	m	i	n	i
i	m	u	m	m	i	n
n	i	m	u	m	m	i
i	n	i	m	u	m	m

i	m	u	m	m	i	**n**
i	n	i	m	u	m	**m**
m	i	n	i	m	u	**m**
m	m	i	n	i	m	**u**
m	u	m	m	i	n	**i**
n	i	m	u	m	m	**i**
u	m	m	i	n	i	**m**

Figure 2.8: Burrows–Wheeler Transform example
The data to encode is underlined.

This algorithm was developed by David J. Wheeler in 1983. It was presented publicly by Michael Burrows and David J. Wheeler in 1994 as a part of block-sorting compression algorithm [BW94].

2.3.2 MTF

Move-to-Front (MTF [BS+86]) is a self-organization heuristic for lists. In data compression field, MTF is used to convert the data into a sequence of integers, with hope that values of integers are low and could be effectively encoded using a statistical coding.

The MTF encoder maintains list of symbols, called a MTF list, which is initialized with all the symbols that occur in the data to be compressed. Then, for each symbol from the data, the encoder outputs its position on MTF list (as an integer) and updates the list. A currently encoded symbol is moved from the current position in the list to the front of the list (the first position). The most important property of this technique is that recently used symbols are near to the front of the list. We hope that equal symbols will often appear close to each other in the data and therefore these symbols will be converted to small integers. Usually small integers appear more frequent so they are encoded in fewer bits than larger integers using a statistical coding, for example, the Huffman or the arithmetic coding.

As one can see in Figure 2.9, MTF list is initialized with input alphabet <a, b, c>. In the next step, the encoder processes ‘c’ by checking its position (counting from 0) on the list and outputs 2. The MTF list is updated, and ‘c’ is moved to the front of the list. The remaining symbols are moved one position further from the front of the list. Then, the encoder processes ‘a’, which is at the position number 1. The list is updated to <a, c, b> and the encoder processes the second ‘a’ and so on.

MTF is used as a part of several other compression algorithms including the Burrows–Wheeler Transform (BWT). There are many MTF variations, for example MTF1 [BK+99]. When MTF1 encounters a symbol that is at the position 1 (counting from 0) of the list, then it moves the symbol to the front. Otherwise, it moves encountered symbol to the position 1. Of course, if the symbol is on the front of the list, its position is not changed.

An MTF list	Data to encode	Encoded data
a b $\underline{c}$	$\underline{c}$aaababbcabcc	2
c $\underline{a}$ b	$\underline{a}$aababbcabcc	1
$\underline{a}$ c b	$\underline{a}$ababbcabcc	0
$\underline{a}$ c b	$\underline{a}$babbcabcc	0
a c $\underline{b}$	$\underline{b}$abbcabcc	2
b $\underline{a}$ c	$\underline{a}$bbcabcc	1
a $\underline{b}$ c	$\underline{b}$bcabcc	1
$\underline{b}$ a c	$\underline{b}$cabcc	0
b a $\underline{c}$	$\underline{c}$abcc	2
c b $\underline{a}$	$\underline{a}$bcc	2
a c $\underline{b}$	$\underline{b}$cc	2
b a $\underline{c}$	$\underline{c}$c	2
$\underline{c}$ b a	$\underline{c}$	0

Figure 2.9: Move-to-Front (MTF) encoding example
The currently encoded character is underlined.

2.4 Predictive compression techniques

The main idea of predictive compression is to take advantage of the previous *r* characters to generate a conditional probability of the current symbol. The predictive coding techniques can be considered as a subset of a statistical coding. They use an arithmetic encoder in a coding stage of two-stage lossless data compression. The predictive methods achieve much higher effectiveness than unconditional entropy as they use conditional probabilities for symbols.

2.4.1 PPM

Prediction by Partial Matching (PPM) is a well-established adaptive statistical lossless compression algorithm, originally developed by Cleary and Witten [CW84]. Nowadays, its advanced variations offer one of the best compression ratios. Unfortunately, the memory requirements for PPM are high and the compression speed is relatively low.

PPM is an adaptive statistical compression method. A statistical model accumulates count of symbols (usually characters) seen so far in the input data. Thanks to that, an encoder can predict probability distribution for new symbols from the input data. Then a new symbol is encoded with a predicted probability by an arithmetic encoder. As higher the probability, as fewer bits are needed by the arithmetic encoder to encode the symbol, and the compression effectiveness is better.

The statistical PPM model is based on contexts. The *context* is a finite sequence of symbols preceding the current symbol. The length of the sequence is called an *order* of the context. The *context model* keeps information about count of symbols' appearances for the context. *PPM model* encompasses all context models. Maximum allowable order of the context is called a PPM model's order or shorter a *PPM order*. The contexts are adaptively built from scratch. They are created during the compression process. While encoding a new symbol from the input data, the algorithm is in some context. This context is called *active*. An order of this context is between zero and *r*, where *r* is a PPM model's order. After encoding the new symbol, the PPM model is updated. The counters of the symbol for contexts models from active order until *r* are updated. This technique is called an *update exclusion*. If some contexts do not exist, then they are created. Updating contexts of lower order may distort probabilities, which may results in worse compression effectiveness. The PPM model has also a special order –1, which contains all symbols from the alphabet with equal probability.

Several factors influence on selection of a PPM order. A higher order is associated with larger memory requirements but also with a better estimation of a probability, that is, also with a better compression effectiveness. Sometimes a symbol to encode has not appeared in the context, and the probability for this symbol equals zero. It happens especially in higher orders. Therefore, a new symbol called *escape* is added to each context. This symbol switches the PPM model to a shorter context. The estimation of probability for the escape symbol is a very important and difficult task. There are many various methods of selection of the escape symbol's probability: PPMA [CW84], PPMB [CW84], PPMC [Mo90], PPMD [Ho93], PPMZ [Bl98], and PPMII [Sh02a], to name the most important ones only. The most often applied order for widely used PPMD is five as higher orders cause deterioration in the compression effectiveness. This is caused by a frequent occurrence of the escape symbol and its bad estimation. Nevertheless, an estimation of the escape symbol's probability for PPMII is much better. The cPPMII algorithm—the complicated version of PPMII [Sh02a]—uses orders even up to 64, but the main reason allowing using such high orders is much better estimation of ordinary symbols' probability (thanks to using an auxiliary model together with the standard PPM model). On the other hand, high orders are rarely used in practice as they have much higher memory requirements.

PPMA, PPMB, PPMC, and PPMD methods have fixed assumptions about a probability of the escape symbol. For example, an escape frequency in PPMD for the context c is equal to $(u/2)/a$, where u is the number of unique symbols seen so far in the context, and a is the number of all symbols seen so far in the context. An escape estimation in PPMII and PPMZ is adaptive. It uses a Secondary Escape Estimation (SEE [AS$^+$97]) model. SEE is a special, separate model used only for better evaluation of a probability of escape symbol. Most of PPM models use statistics from the longest matching context. PPMII inherits the statistics of shorter contexts when a longer context is encountered for the first time. The shorter (the last longest matching) context's statistics are used to estimate statistics of the longer context.

[encoded data] and data to encode	Order 0		Order 1		Order 2	
	context	char	context	char	context	char
abaaababab	null	a=1				
[a]baaababab	null	b=1	a	b=1		
[ab]aaababab	null	a=2	b	a=1	ab	a=1
[ba]aababab	null	a=3	a	a=1	ba	a=1
[aa]ababab	null	a=4	a	a=2	aa	a=1
[aa]babab	null	b=2	a	b=2	aa	b=1
[ab]abab	null	a=5	b	a=2	ab	a=2
[ba]bab	null	b=3	a	b=3	ba	b=1
[ab]ab	null	a=6	b	a=3	ab	a=3
[ba]b	null	b=4	a	b=4	ba	b=2

Figure 2.10: PPMC encoding example (without update exclusions) for PPM order 2
The longest context used in model updating is underlined.

Figure 2.10 presents an example of the PPMC encoding process. To simplify, we are using PPMC without update exclusions, so PPM updates always all orders. At the beginning the PPM model is empty. In each step, the algorithm encodes one symbol and updates the model (orders 0, 1 and 2) according to Figure 2.10. PPMC model after encoding the string ‘abaaababab’ is presented in Figure 2.11. As one can observe, the string ‘abaaababab’ contains of six ‘a’ symbols and four ‘b’ symbols, what agrees with the order 0 statistics in Figure 2.11. The string ‘abaaababab’ contains, for example, two overlapping substrings ‘aa’. It matches with the context ‘a’ and the char ‘a’ in the order 1 of Figure 2.11.

We have used PPMC as for this PPM variation a value of the escape symbol is the number of unique symbols seen so far in the context. The last symbol from the string ‘abaaababab’, that is ‘b’, was encoded in order 2, in the context ‘ba’. There were three symbols in this context ‘a’, ‘b’, and ‘esc’, with probabilities 1, 1, and 2 respectively, that is four in total. The symbol ‘b’ was encoded with the estimated probability equal to ¼, which is encoded on $-\log_2 ¼$ bits = 2 bits. Then, the context is updated to ‘a=1’, ‘b=2’, and ‘esc=2’ as in Figure 2.11.

Order 0		Order 1		Order 2	
context	char	context	char	context	char
null	a=6 b=4 esc=2	a	a=2 b=4 esc=2	aa	a=1 b=1 esc=2
				ab	a=3 esc=1
		b	a=3 esc=1	ba	a=1 b=2 esc=2

Figure 2.11: PPMC model after encoding ‘abaaababab’

The main disadvantages of the PPM algorithm are high memory requirements and relatively low compression speed. Generally, those drawbacks are becoming more acute with increasing the maximum context order. Higher orders provide more information necessary for reliable prediction but also increase the computational requirements. It should also be stressed that making effective use of high order information is not a trivial task.

2.4.2 PPM*

In most PPM schemes, the maximum context order is finite. Increasing the maximum order may lead to better compression, as higher order contexts are expected to be more specific in their predictions, but practitioners are well aware that this is not always the case. For example, on textual data most available PPM implementations achieve best results with the maximum order length set to five and deteriorate slowly with increasing that value. It happens because high order contexts usually do not contain many occurrences of symbols, which implies that the escapes to lower contexts are frequent, hence increasing the size of the encoded data.

On the other hand, it is obvious that for some files contexts much longer than, let us say, five are useful. Moreover, the “optimal” context lengths may vary within a file. For example, within a large collection of English texts the sequence ‘to be or not to ‘ is very likely to be followed with symbol ‘b’, while truncating this context to order 5, that is to ‘t to ’, makes the prediction much more obscure.

Cleary et al. described the idea of using *unbounded length contexts* [CT$^+$95] in an algorithm named PPM*. This algorithm extends PPM with possibility of referring to arbitrarily long contexts, which may prove very useful on very redundant data, and is quite safe from a significant loss on data that do not require such long contexts.

To explain how PPM* works, one simple definition is needed. A context is called ***deterministic*** if only one symbol has appeared in it so far. There may be many occurrences of a given context, but as long as it predicts the same symbol, it remains deterministic. PPM* algorithm keeps a pointer to the last context's appearance (a pointer to the input data) for each deterministic context. Cleary et al. noticed that in such contexts the probability of a new symbol is much lower than expected, based on uniform prior distribution. In other words, earlier PPM algorithms, handling all contexts in the same manner, had underestimated the prediction ability of deterministic contexts.

While encoding, the algorithm uses the shortest deterministic context from the list of available contexts (ancestors). This context contains a pointer to a reference context (the last context's appearance). At the beginning a predicted symbol is the first symbol after the reference context. If the prediction is correct (the predicted symbol is the same as the symbol to encode), then PPM* encodes the predicted symbol and tries to encode a new predicted symbol, that is the second symbol after reference context and so on. The sequence of successive correctly predicted symbols can be arbitrarily long, therefore contexts have unbounded length. If the prediction fails, then the escape symbol is emitted. If there is no deterministic context for encoding or the escape symbol was emitted, then the longest PPM context is used instead. It means that in such cases PPM* behaves like an ordinary PPM coder.

Order 0		Order 1		Order 2	
context	char	context	char	context	char
null	a=6 b=4 esc=2	a	a=2 b=4 esc=2	aa	a=1 b=1 esc=2
				ab	a=3 esc=1 reference context='…abaaabab' (followed by 'ab…')
		b	a=3 esc=1 reference context='…abaaabab' (followed by 'ab…')	ba	a=1 b=2 esc=2

Figure 2.12: PPMC* model after encoding 'abaaababab'

Figure 2.12 presents PPMC* model after encoding the string 'abaaababab'. The model contains only two deterministic contexts 'b' and 'ab'. Let suppose that the next symbol to encode after the string 'abaaababab' is 'a'. PPM* chooses the shortest deterministic contexts for the string 'abaaababab', that is 'b'. The reference context for the deterministic context 'b' is '…abaaabab', which is followed by the string 'ab…'. PPM* compares 'a' with the predicted symbol (the first symbol after the reference context), that is 'a'. The prediction is correct, so PPM* compares the next character to encode with 'b' (the second character from 'ab…') and so on. As can be seen, while using a pointer a virtual deterministic context (with a predicted symbol and an escape symbol) is created. This is the main advantage over the PPM algorithm, which may use non-deterministic contexts to encode the same symbol. The symbol in non-deterministic contexts is usually encoded with lower probability than in deterministic contexts.

2.4.3 PPMZ

PPMZ algorithm is a PPM variation developed by Bloom [Bl98]. It uses several important techniques to boost compression: a heuristic for choosing the context order to start encoding from (called Local Order Estimation or shorter LOE), an adaptive mechanism for escape probability estimation (called Secondary Escape probability Estimation or shorter SEE), and unbounded length deterministic contexts.

SEE is designed to improve the probability estimation of escape symbols. It works by joining statistics of occurrences of escape symbols for similar PPM contexts. The SEE model consist of order 2 escape contexts, independent from PPM model. The escape context is created as concatenation of various components, like the PPM order, the number of escapes and the number of correct predictions. All components are quantized to a few bits.

The unbounded length deterministic contexts in PPMZ is based on PPM*, but it has some modifications. PPMZ uses only very long unbounded contexts (order 12 or higher) and imposes a minimum match length bound (for each unbounded context separately) to eliminate errors when the length of predicted symbols is too short. The unbounded contexts are always deterministic.

The unbounded contexts in PPMZ are additional to the standard PPM model (where the maximum order is less than 12) and in a way independent of it. Each unbounded context holds a pointer to the last context's appearance (a pointer to the input data), the minimum match length, and the match frequency. PPMZ constructs unbounded context models adaptively. They are built after encoding each symbol and indexed by a hashed value of the last 12 appeared symbols.

While encoding, the algorithm finds an appropriate unbounded deterministic context for an active context (the same index value for the active context and the deterministic context). The unbounded context contains a pointer to a reference context (the last context's appearance). Then PPMZ calculates an order of common context for the active context and the reference context. In the other words, PPMZ counts maximum length of the match between currently encoded symbols and symbols that built the reference context. The unbounded context and the active context have the same index value so, using a good hash function, minimum length of the match equals 12. PPMZ algorithm checks if the match length is equal to or greater than the minimum match length (which initially equals 12). When an unbounded deterministic context is not found or the match length is too small, then PPMZ uses the ordinary PPM encoding. It means that in such cases PPMZ behaves like an ordinary PPM coder.

If the length is equal to or exceeds the minimum match length for this context, then the PPMZ encoding is attempted. The predicted symbol is the one (from definition) symbol in the unbounded deterministic context. If the prediction is correct (the predicted symbol is the same as the symbol to encode), then PPMZ encodes the predicted symbol and updates the match frequency in the unbounded deterministic context. Then PPMZ starts from the beginning and searches for the next unbounded deterministic context that matches with a new active context. If the prediction fails, the escape symbol is emitted and the symbol is encoded using the ordinary PPM encoding. Moreover the minimum match length in the unbounded context is changed to a calculated match length plus one. It assures that this error will never occur again as similar contexts (an active context and a reference context) will be distinguished. In this manner, PPMZ forces all unbounded contexts to be deterministic.

2.4.4 PPMII

In 2002 Shkarin has presented a paper [Sh02a] on his PPM with Information Inheritance (PPMII) algorithm. Most of the ideas described in the paper could earlier been known to the compression community from the public sources of his own implementation, called quite confusingly, PPMd (PPM by Dmitry [Sh02b]). The algorithm offers a very attractive trade-off between compression effectiveness and speed, and thanks to that it had been chosen for implementation in several third-party archivers, including the well-known RAR [Ro04]. Shkarin's PPM is a complex scheme, and it is hard to describe it here in details, so we present below only its basic ideas.

The main novelty of PPMII (reflected in its name) is passing gathered information (statistics) from parent to child contexts. When a longer (child) context appears for the first time, it already contains some "knowledge" (probability estimation) inherited from its parent. In this way, the PPM model may learn faster, and the problem with sparse statistics in high orders is mitigated.

PPMII uses a well-known adaptive mechanism for the probability estimation of the escape symbol (called Secondary Escape probability Estimation or shorter SEE). Nevertheless, this technique is significantly modified. It divides the contexts into three classes to more efficiently estimate escape probability, which also changes probabilities of remaining symbols in the contexts. In addition, important practical gains are achieved via rescaling statistics and thus exploiting the recency (in non-stationary data).

A complicated version of PPMII (cPPMII [Sh02a]) uses orders even up to 64, but the main reason allowing using such high orders is much better estimation of ordinary symbols' probability using Secondary Symbol probability Estimation (SSE [Sh02a]) mechanism. SSE is an auxiliary adaptive model built similarly to the model for the escape symbol (SEE). It also divides the contexts into three classes to better estimate the probabilities of ordinary symbols in the contexts.

Together with some other minor improvements and very efficient data structures, on textual files PPMII can achieve very good compression ratios at the compression speed comparable to gzip in the best compression mode (see PPMII, order 8, results on the Calgary corpus in Reference [Sh02a]). The main drawbacks of this scheme are quite significant memory requirements in high orders.

2.4.5 PPM with built-in models

Teahan and Cleary [TC96] experimented with an estimation of entropy of the English language. They trained a PPM compressor on English texts and experimented with such a pre-loaded PPM model. In this scheme, the compressor starts compression with lots of useful knowledge and gives more reliable estimation of entropy of the English language.

Shkarin used the pre-loaded PPM model idea to improve the compression effectiveness of his PPM compressors (PPMd [Sh02b] and PPMonstr [Sh02b]), and he created Durilca [Sh04]. The most important thing is that a built-in PPM model is suitable only for the same kind of data that it was trained on. When a built-in model is used on different kind of data, it will deteriorate the compression effectiveness, instead of improving. Durilca incorporates several built-in models, selected for given data via a data detection routine. It employs the PPM engine directly, without any modification, what makes this scheme similar to the preprocessing approach. Durilca, together with its faster version, Durilca Light, achieve excellent compression performance, but the main drawback of this approach is limitation to narrow set of files (it includes built-in models only for English texts, Russian texts and 32-bit

Intel executable files). It is worth to mention that built-in models are stored in a file, which occupy more that 1.7 MB.

2.4.6 PAQ

PAQ [Ma02] is a family of compressors, originally developed by Matthew Mahoney, based on the PPM idea. The main difference is that PAQ uses a binary alphabet, what highly decreases the compression (and the decompression) speed, but it gives additional opportunities. PPM uses a character-based alphabet and a new symbol in a context must be encoded in lower orders using an escape mechanism. PAQ does not use the escape symbol at all as in each step it must encode only 0 or 1. The binary alphabet causes that a new character in a context can be distinguished after first unseen bit, what is not possible in the case of PPM. It is the next improvement to the PPM algorithm. In the PAQ's coding stage a binary symbol is encoded with a predicted probability by an arithmetic encoder, like in the PPM algorithm.

The biggest improvement to the PPM algorithm is that PAQ uses several models (bit-level predictors) comparing to a single PPM model. PAQ1 combines following bit-level predictors [Ma02]:

- A bland model with 0 or 1 equally likely,
- A set of order 1 through 8 nonstationary *q*-gram models,
- A string matching model for *q*-grams longer than 8,
- A nonstationary word unigram and bigram model for English text,
- A positional context model for data with fixed length records.

A probability for '0' or '1' symbol can be equal to zero, and the bland model ensures that this does not happen. PPM uses a stationary model, in the sense that the statistics used to estimate the probability (and therefore the code length) of a symbol are independent of the age of the data. Nevertheless, in many types of data newer statistics are more reliable [CR97] and PAQ1 uses this idea in the set of n-gram models and also in the word model.

PAQ6 [Ma03] and its modifications are compressors that currently give the best compression effectiveness (at the slowest compression speed). PAQ6 is much more sophisticated than its predecessors. It has eight independent bit-level predictors and requires up to 1626 MB of memory. Comparing to PAQ1, it uses three additional models:

- A sparse model for contexts with gaps,
- An analog model for audio and graphics data,
- An exe model for 32-bit Intel executable files.

In PAQ6 the bit-level predictors (models) are weighted adaptively by a *mixer* to favor those making the best predictions. There are two independent mixers, which use sets of weights selected by different contexts. An output of two independent mixers is averaged and adjusted by Secondary Symbol probability Estimation (SSE), known from the PPMII algorithm. SSE updates the probability considering previous experience and the current context. Figure 2.13 presents PAQ6 encoding process.

The main disadvantages of PAQ algorithm are very high memory requirements and very low compression speed, what makes this algorithm unattractive from a practical point of view.

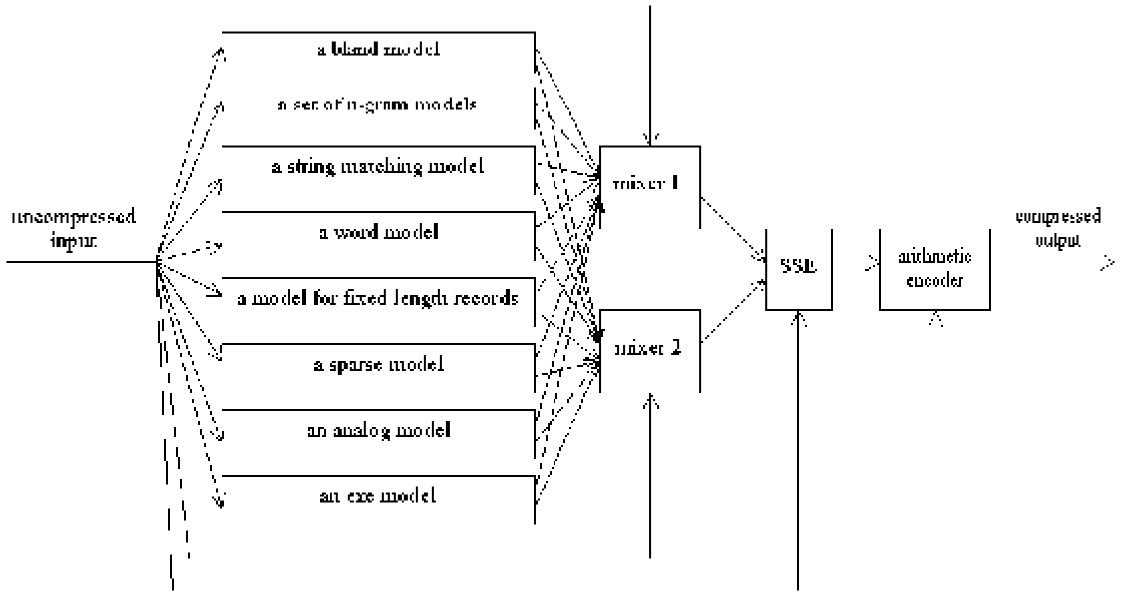

Figure 2.13: PAQ6 encoding process

2.5 Word-based compression techniques

All above-mentioned compression techniques (statistical, dictionary-based, BWT-based and predictive) are usually implemented using 256 characters (8 bit ASCII) alphabet. There are, however, a few attempts to develop word-based compression techniques, where alphabet consists of an unknown number of words in a natural language. Word-based versions of well-known compression algorithms are the word-based Huffman encoding [HC84], the word-based LZW [HC92], the word-based BWT [IM+02], and the word-based PPM [Mo89]. Recently, syllable-based algorithms LZWL [LZ05] and HuffSyllabe [LZ05] have appeared, in which an alphabet consist of syllables. Word-based and syllable-based algorithms usually perform better than their characters-based equivalents, but they are limited only to textual files, they have a higher memory requirements and a lower compression/decompression speed. From a practical point of view, an improvement in the compression effectiveness is relatively small and word-based compression techniques are rarely used. Much more useful seems a word-based textual preprocessing, described in the Chapter 4, which also divides input data into words in natural language.

2.5.1 Encoding words as integers

Sentences in most natural languages consist of words, which can be defined as sequence of letters over some alphabet, and separators, like spaces or punctuation marks. In 1972 Hagamen et al. have presented one of the first word-based compression algorithms [HL72], which divides input data into words in a natural language. In this scheme, the words were encoded using fixed-point numbers.

In Figure 2.14 we present an example of encoding words as integers. It is one of the simplest variations. In this scheme each word is transmitted as a string of letters just once. The next occurrences of each word are encoded as integers (fixed or variable-length). As one can see in Figure 2.14, we are encoding the string ‘the car hit the other car’. The first three words are encoded in their original form, because they appear for the first time. They are placed in a dictionary. Then, there is the remaining string ‘the other car’ to encode. The word ‘the’ is encoded as the number 1 in the dictionary. The word ‘other’ has appeared for the first time, and it is placed in a dictionary as the number 4. The last word, ‘car’, is encoded as the number 2 from the dictionary.

This scheme has property that each new word corresponds to the integer, which is one greater than the total number of distinct words that have appeared so far. Integers that correspond with words, must be suitably encoded to be distinguished from an ordinary data by a decoder.

the dictionary	data to encode	encoded data
	the car hit the other car	the
1=the	car hit the other car	car
1=the, 2=car	hit the other car	hit
1=the, 2=cat, 3=hit	the other car	1
1=the, 2=car, 3=hit	other car	other
1=the, 2=car, 3=hit, 4=other	car	2

Figure 2.14: Example of encoding words as integers
The currently encoded word is underlined.

2.5.2 Word-based Huffman coding

In early and mid-1980s, Ryabko [Ry80], Horspool and Cormack [HC84], and Bentley et al. [BS$^+$86] have presented independently an idea of word-based Huffman compression.

In the scheme developed by Bentley et al., words were defined as the longest sequences of alphanumeric and non-alphanumeric characters. According to this definition, the input data is divided into two disjoint word classes, which are compressed separately by the adaptive Huffman compression. The words alternate, so there is no need to include additional information about a class of word in the output data. If a word appears for the first time, it is signaled using the escape code, analogous to the escape symbol used in the PPM compression. Next, the word is encoded by the character-based Huffman coding, using separately models for alphanumeric and non-alphanumeric characters. Thus, there are four different models, two models for words and two models for characters.

The compression effectiveness of the word-based adaptive Huffman coding is about 10% better than the character-based adaptive Huffman coding. On the other hand, as the alphabet of words grows, the compression speed becomes lower. Moreover, according to Horspool and Cormack [RH$^+$87], the compression effectiveness of word-based adaptive Huffman is worse than the LZ77 algorithm, which is simpler to implement and executes much faster.

The word-based semi-adaptive Huffman coding is faster than the adaptive version. It makes two passes over the input data. The first pass obtains the frequency of each word, and the second pass performs the actual compression. Nowadays, the word-based semi-adaptive Huffman coding is successfully used in text retrieval applications [BM$^+$93, MN$^+$00], where the decompression speed is very important. Moreover, the semi-adaptive Huffman coding allows searching on compressed data, without having to decompress the whole data as in adaptive method.

2.5.3 Word-based LZW

The word-based version of the LZW algorithm was independently presented in 1992 by Horspool and Cormack [HC92], and Jiang and Jones [JJ92].

The word-based adaptive LZW algorithm developed by Horspool and Cormack has a very similar idea to the word-based adaptive Huffman coding, presented in the previous subsection. The input data is divided into two disjoint word classes (consisting of alphanumeric and non-alphanumeric characters), which are compressed separately by the word-based LZW algorithm. There is no need to include additional information about a class of word in the output data as the words alternate. If a word appears for the first time, it is signaled using an escape dictionary entry. Again, it is possible to apply the same basis algorithm for the character-based and the word-based encoding. Thus, the word is encoded by character-based LZW compression, using separately models for alphanumeric and non-alphanumeric characters. As in the word-based Huffman coding, there are four different models, two models for words and two models for characters. There is, however, a need to take note of a one important detail. During the encoding using character-based LZW, we must treat the end of each new word as equivalent to the End-of-File (EOF). Otherwise, the next character after the new word will be added with this word as a new dictionary entry, what is undesirable.

The word-based adaptive LZW algorithm achieves a similar compression effectiveness to the word-based adaptive Huffman coding. Nevertheless, the compression and the decompression speed are better, especially when dictionaries are implemented as large hash tables.

Jiang and Jones [JJ92] a developed word-based adaptive LZW variation called WDLZW. In this scheme, input data is divided into words, which are defined as sequence of letters, and separators, like spaces or punctuation marks. This algorithm introduces a *spaceless* model, which assumes that there is a space character between every two words and there is no need to encode this space character. Most of words fulfill this assumption, and remaining words are treated in a special way. Words are encoded using the LZW algorithm. Separators and new words are encoded in their original form, in blocks, which are preceded by a length of block. To distinguish words and separators/new words, the non-interfering codes are used. To improve the compression effectiveness, WDLZW uses four dictionaries of different size (with indexes encoded using 7, 9, 11, and 13 bits). The dictionaries are identified using a combination of 2-bit codes. The weighting system places the most frequently used strings of words in smaller dictionaries, which require fewer bits to encode the corresponding string of words. This scheme is designed to minimize the average number of bits required to code a string of words comparing with a single dictionary LZW algorithm. Moreover, the WDLZW algorithm initializes dictionaries with 9572 predefined English words, what improves the compression effectiveness on English textual files.

Dvorský et al. [DP+99] have presented the word-based semi-adaptive LZW algorithm, called WLZW, based on a work of Horspool and Cormack. Their scheme is designed for text retrieval applications, when two passes over the input data do not interfere. On the other hand, the semi-adaptive algorithm gives a possibility to create a full-text index, and it is faster than the adaptive version, what is very important in the text retrieval (especially the decompression speed).

2.5.4 Word-based PPM

The word-based adaptive Huffman coding, presented by Bentley et al. [BS+86], can be easily modified to the word-based arithmetic coding, or in other words to an zero-order word-based

PPM algorithm. The arithmetic coding can be used for both an encoding words and an encoding characters, when a new word has appeared. This idea was used and extended in a work of Moffat [Mo89] from 1989.

Following Bentley et al., in the word-based adaptive PPM words were separated into two classes, defined as the longest sequences of alphanumeric and non-alphanumeric characters. The words alternate, so there is no need to include additional information about a class of word in the output data. Moffat has extended the word model to the first-order, thus words are predicted by preceding words. If a word has not appeared in the first-order word context, then it is encoded in the zero-order word context. If a word has appeared for the first time, its length is encoded using the special zero-order model to encoding the words lengths (numbers from 0 to 20). Then, the word is encoded using the first-order character-based PPM. The method B is used for estimating probabilities in both, word and character models, as it gives better compression effectiveness using large alphabets. In total, there are five separate models: two (the first and zero) word models, two (the first and zero) character models and the zero-order model for words lengths. As there are two classes of words and words that belong to one class are predicted by preceding words from the same class, therefore the number of models is doubled. Moffat experimented also with the second-order word-based adaptive PPM compression, but the gain accrued from the change from the first-order to the second-order was very small.

Horspool and Cormack [HC92] have proposed to group similar words in PPM model. Words are divided into groups by their parts of speech in the English language. Horspool and Cormack have assumed, for simplicity, that there are only five parts of speech named ***article***, ***noun***, ***adjective***, ***verb*** and *other*. Knowing the part of speech tag of a word helps in predicting this word, and this tag is predicted by a tag of the previous word. This idea can improve the compression effectiveness, but we must remember that tags must be encoded within output data. Moreover, the words must have assigned tags to them. Horspool and Cormack have solved this problem by assigning to a new word a tag that is most likely to follow a tag for the preceding word. For each word there are statistics on how well this word fits into its tag. If a better compression would have been achieved with a different tag, this word is dynamically reassigned to the other tag.

Teahan and Cleary [TC97] have continued work started by Moffat and Horspool and Cormack. They assumed that the words were already tagged using a much more comprehensive tag set than used by Horspool and Cormack. Teahan and Cleary have experimented with different ways of prediction of words. For example, their “TTT” model predicts using tags of two previous words. If this prediction is unsuccessful, the algorithm escapes and predicts using only tag of preceding word, and finally the algorithm escapes again and predicts without any context. The experiments have leaded the authors to the conclusion that modeling by parts of speech is just comparable to the pure word-based compression as it requires that the tags must be encoded with the words.

Experiments carried out by Teahan and Cleary have showed that the word-based adaptive PPM is about 4% better than the character-based order 5 PPMD algorithm. The word-based approach is, however, limited to textual files, and requires more memory than PPMD. Nowadays, the word-based adaptive PPM is not competitive to recent character-based PPM variations like PPMII, which uses very high orders.

The word-based PPM uses only the first-order model, and we believe that a better effectiveness can be achieved when the character-based PPM and the word-based PPM will be joined into one model, instead of using separate models. This can be done by extending a character-based alphabet with words. A similar idea was successfully implemented in the PPMEA algorithm (described in the next chapter), which adds to the alphabet long repeated strings of characters instead of words.

2.5.5 Word-based BWT

The most recent and the most successful algorithm from a family of word-based algorithms is semi-adaptive word-based BWT, developed by Sadakane [Sa99] in 1999.

The semi-adaptive word-based BWT makes two passes over input data. Following Bentley et al., words are defined as the longest sequences of alphanumeric and non-alphanumeric characters, and divided into two disjoint word classes. The words from these two classes, however, are compressed together, with the same model. In the first pass word-based BWT creates a dictionary (a table), in which unique words have assigned consecutive numbers. This dictionary is outputted and used in the second pass as an input alphabet. The second pass performs the ordinary BWT compression, but on an extended alphabet, which is much bigger than the ordinary alphabet.

A work of Sadakane was continued by Isal and Moffat [IM01, IM$^+$02]. They used the semi-adaptive word-based BWT with a spaceless model [JJ92], which assumes that there is the space symbol after each word and there is no need to encode this space symbol. Experiment carried out by Isal and Moffat have showed that 75% words consisting of non-alphanumeric characters are omitted in the spaceless model.

Isal and Moffat have suggested that the cost of storing the dictionary can be reduced by sorting it, front-code the words by eliminating the common prefixes [WM$^+$99, page 160], and coding the stream of remaining characters with the character-based BWT algorithm. Moreover, they optimized the MTF transform to the extended alphabet and introduced a new heuristic. According to this heuristic, words of frequency one should never be promoted at all in MTF list, since they only occur once in input data and have just occurred.

On large textual files, the semi-adaptive word-based BWT, performs better than the word-based PPM, even slightly better than PPMII (but not better than cPPMII), which is a recent character-based PPM variation. The compression effectiveness of the word-based BWT is comparative to the PPMEA algorithm, but worse than achieved with the PPMVC scheme (both algorithms are presented in the next chapter). The best results on large textual files are, however, achieved with a word-based textual preprocessing, described in Chapter 4.

2.6 Putting lossless data compression into practice

From a practical point of view, we can divide lossless data compression algorithms into four classes, presented in Figure 2.15.

Class	Compression effectiveness	Compression speed	Decompression speed	Memory requirements
LZ	Average	High	Very high	Low
BWT	Good	Average	Average	Average
PPM	Very good	Low	Low	High
PAQ	The best	Very low	Very low	Very high

Figure 2.15: Comparison of practical lossless data compression algorithms

LZ class can be characterized as a high compression speed, a very high decompression speed at average compression effectiveness. It is probably the main reason why LZ-based archivers (PKZip, gzip, WinZip) are currently the most popular and widely used. BWT class is slower in the compression and the decompression, but achieves a better compression effectiveness. BWT-based archivers (bzip2) follow after LZ-based archivers and gain popularity. PPM class was for a long time unpractical, but its value grows with a progress in electronics and with faster computers. There is no PPM-based archiver that is a standard in its class as gzip and bzip2 for LZ and BWT, respectively. Nevertheless, archivers developed by Shkarin (PPMd, PPMonstr, Durilca, and Durilca Light) can be taken into consideration as strong candidates. Some of these programs achieve very good compression effectiveness at a speed comparable to bzip2. The last class is PAQ, which gives the best compression effectiveness, but with an unacceptable compression and a decompression speed. In a few years, however, situation on a lossless data compression field can be changed.

3 Improved predictive compression techniques

There are some kinds of data, which contains long repeated strings. These data are compressed with PPM in a way far from optimal, and similar compression effectiveness can be achieved with LZ77-based compressors. The PPM* and PPMZ algorithms use arbitrarily long contexts, which improve predictions on data that contains long repeated strings. Nevertheless, these algorithms do not fit for modern PPM variations like PPMII as they can deteriorate the compression effectiveness, instead of improving.

This chapter presents our two predictive compression algorithms, which improve state-of-art predictive methods introduced in the previous chapter. We have designed two algorithms that improve compression effectiveness on highly redundant data. The first method extends PPM alphabet to long repeated strings. The second method extends the PPM algorithm with string matching abilities, similar to the one used by the LZ77 algorithm. Both algorithms were tested in a connection with PPMII.

3.1 PPMEA

In this subsection we present a universal modification of any algorithm from the PPM family, which improves the compression effectiveness. This scheme is based on a Bentley–McIlroy's algorithm to finding long repeated strings.

Low orders in the PPM algorithm are often used to limit memory requirements. In this modification better compression effectiveness can be achieved in the same memory requirements. The compression and the decompression speed for the ordinary PPM algorithm is almost identical, since the encoder and the decoder perform similar tasks. The proposed modification decreases the compression speed, as we have to find long repeated strings before starting of the PPM compression. The decompression speed is, however, higher because strings from the extended alphabet are copied from the dictionary, and there is no need to modify a context tree for each character from this string. This approach is especially desirable for the off-line compression [AL00, AL98, LM99, TS02], where only the decompression time is significant.

3.1.1 An idea of PPM with extended alphabet

Let us assume that we have a file, in which we can find a long repeated (at a random distance) string with the length of *m* bytes (characters). PPM has to encode each character of this string separately (but with very high probability). PPM always must have a possibility to encode the escape symbol, when a new symbol in context has appeared. In this case, it is probable that the LZ77 algorithm would compress better, as it encodes only the offset, the length, and the next character. But if we extend the PPM alphabet with a new symbol, which is equal to our string (with the length *m*), then we can encode this string much more efficiently. Let us assume that a dictionary is a set of all long repeated strings. In this case, our alphabet consists of 256 characters (the ordinary PPM alphabet), the escape symbol, and symbols, which are equal to long repeated strings from the dictionary. We have to pre-pass over data to calculate statistics and to find repeated strings to construct the alphabet before starting the PPM compression. This algorithm is called PPM with extended alphabet (PPMEA [Sk06]).

Using other alphabet than 256 characters in the PPM algorithm is not a new idea. For example, Moffat used it in the word-based PPM algorithm [Mo89], which is also described in this dissertation.

3.1.2 Finding long repeated strings

In the paper of Bentley and McIlroy [BM01], we can find description of an interesting algorithm, which has been created to find long repeated strings. This is a preprocessing algorithm, which can interact with many known compression algorithms (the authors use it in combination with LZ77). It is based on the Karp–Rabin's algorithm [KR87] to find pattern in a text (with using of a fingerprint). Data are divided into parts with the length *b*. Each part has its own fingerprint. Fingerprints are stored in a hash table. In the following step, we are scanning the data, computing the fingerprints for each string of the length *b*. When the fingerprint of a string can be found in the hash table, we have a very high chance that we found the repeated string. If the strings are equal, then we can extend them to all common characters before and after these strings.

The Karp and Rabin's algorithm can find all the occurrences of a string *s* of the length *m* in the input text of the length *n*. A fingerprint is made from *m* characters of the string *s* as a polynomial modulo of a large prime number. The Karp and Rabin's algorithm scans the input text computing the fingerprint for each of $n–m+1$ substrings of the length *m*. If the fingerprint of the string *s* and the fingerprint of a substring are not equal, then we are certain that the substring does not match the string *s*. When fingerprints are equal, we have to check if the string *s* does match the substring. Karp and Rabin proved that a fingerprint can be computed in $O(m)$ time and updated, while scanning an input text, in $O(1)$ time. The worst-case complexity of this algorithm is $O(m \cdot n)$. For ordinary data, however, it runs close to linear time.

The Bentley and McIlroy's algorithm divides the input data into blocks of the size *b*. For each block a fingerprint is created to give n/b fingerprints. The fingerprints are stored in a hash table. When we scan input data and evaluate the fingerprints, we can find equal fingerprints in the hash table. When fingerprints do match and blocks of the size *b* do match, we can extend these blocks to all common blocks and characters before and after the blocks. The Bentley and McIlroy's algorithm detects all repeated strings with the length at least $2b–1$, may detect repeated strings with the length from *b* through $2b–2$, and will not detect repeated strings with the length less than *b*. If the strings do overlap, then it must be cut (shortened) not to overlap. Bentley and McIlroy claim that the worst-case complexity of this algorithm is $O(n^{3/2})$, but for "realistic" data, it runs close to linear time. This algorithm is constructed especially for long repeated strings, and nowadays it is the fastest known algorithm solving problem of finding long repeated strings.

3.1.3 Which repeated strings should be added to the dictionary

The extended alphabet idea can even slightly worsen the compression effectiveness, instead of improving, if we add to many symbols into the alphabet or if we choose too short repeated strings. So it is very important how we choose the dictionary, and we suggests the following gain function,

$$gain(s) = m \cdot (x–1), \tag{3.1}$$

where *m* is the length of the string *s*, and *x* is the count of strings in compressed data.

This is a very simple function, which describes how many characters from input will be replaced by symbols from the dictionary. We may say that this is the amount of bytes that are saved. We do not take into consideration the escape symbol (because usually to use a symbol from the dictionary, we have to switch to the order –1 or 0) and the fact that a replaced string would be compressed with the ordinary PPM alphabet.

Having the gain function, we can add to the alphabet only the strings *s*, for which gain(*s*)>=*z*, where *z* is a boundary of profitability. An optimal value of this parameter is different for each file (even for different compression algorithms, for example, PPMD and PPMII), therefore we must select *z* as constant that gives the best compression effectiveness results on average.

Now we can evaluate value of *b* from the Bentley and McIlroy’s algorithm. In order to find all the repeated strings with the length *m*, we select *m*=2*b*–1 and we get *b*=(*m*+1)/2.

3.1.4 Efficient storage of the dictionary in the compressed data

Before start of actual compression we know how many strings, which meet inequality gain(*s*)>=*z*, were found by Bentley and McIlroy's algorithm. We should reserve for each string symbol in the order –1. These symbols form the extended alphabet and the number of extended symbols needs to be encoded first.

At the first appearance of a string we encode the string using the ordinary PPM alphabet. After encoding of the string, we encode its symbol (from the extended alphabet) and the length of the string. Thanks to that, we can evaluate the string’s start position. We also know the length of the string and we have the unique symbol, which is equal to this string. While the next appearances of the same string, we encode only the symbol of this string (which will be replaced with this string while decoding). We do not need any additional information. There are only two variants of symbol appearance—the first appearance of the symbol followed by the length of the string and the next appearances of the symbol without the length of the string.

3.1.5 Memory requirements

The Karp–Rabin’s algorithm splits a file into blocks of the size *b*. For each block we need four bytes for the fingerprint and four bytes for the block position (or the number). We can estimate the memory required for the Karp–Rabin’s algorithm (without the size of the created dictionary) with the equation

$$mem = n + \frac{8 \cdot n}{b}. \tag{3.2}$$

When we select *b* = (*m*+1)/2 then we get the equation

$$mem = n + \frac{16 \cdot n}{m+1}, \tag{3.3}$$

where *mem* is the required memory, *n* is the length of the file and *m* is the minimal length of the repeated string *s*.

When the file is very large, we can split it into parts (of size, for example, 5 MB) and find repeated strings for each part. In the following step we can join the same strings from different parts of the file. If we split the file into the parts, then the compression effectiveness may be decreased, but in most of cases only slightly. If we know the amount of memory (allocated buffer) that will be used by the PPM algorithm, then we can create the extended dictionary using the same memory. It can be done by splitting the file into the parts, in which the required memory will be less than size of the allocated buffer.

3.1.6 Complex gain function

Although the gain function is evaluated in a very simple way, it gives quite good results. We can try to estimate real gain from adding a string to the extended alphabet to get better compression effectiveness. This idea leaded us to develop a new gain function. Lets look on our gain function, represented by Equation 3.1. The parameters *m* and *x* are obtained from the Bentley and McIlroy's algorithm. We must only select a parameter *z*, which is a boundary of profitability, from inequality *gain(s)*>=z. We can set this parameters to the constant 0, and get inequality *gain(s)*>=0. This should worsen the compression using our gain function, but we have developed the new complex gain function

$$gain_{clx}(\mathrm{s}) = \frac{5 \cdot m \cdot (x-1)}{order_{max}} - 2 \cdot order_{max} \cdot (x-1) + 8 \cdot order_{max} - 120\,. \qquad (3.4)$$

This function depends on $order_{max}$, which is a maximum PPM order. The constant $order_{max}$ lowers the gain function value for higher PPM orders. For example, when maximum PPM order is equal to 5 we get:

$$gain_{clx}(s) = m \cdot (x-1) - 10 \cdot (x-1) - 80. \qquad (3.5)$$

The first two components of the equation ("$m \cdot (x-1) - 10 \cdot (x-1)$") are used to prefer a few longer strings than many short strings, which can have the same gain value using previous gain function. The second component tries to approximate the loss from encoding the symbol from the extended alphabet. The last part of the equation, that is "$8 \cdot order_{max} - 120$" evaluated to "–80", is selected empirically to give the best compression effectiveness with inequality *gain(s)*>=0.

3.2 PPMVC

There are some kinds of data, which contains long repeated strings (e.g., redundant XML files). As we said earlier, these data are compressed with PPM in a way far from optimal, and similar compression effectiveness can be achieved with LZ77-based compressors. The highest order used with PPMD is only five, as the higher order usually causes deterioration in the compression effectiveness. In PPMII compression we can use a higher order, but then the algorithm has much higher memory requirements. Additionally, many data types do not seem to ever benefit from high order dependencies in the PPM compression, so the best we could do for those files is just not to deteriorate the compression effectiveness in comparison with a low-order modeling.

The PPM* algorithm was created to solve above-mentioned problem, as it uses arbitrarily long contexts and is quite safe from a significant loss on data that do not require such long contexts. Nevertheless, PPM* does not fit for modern PPM variations like PPMII, as it can deteriorate the compression effectiveness, instead of improving. That is the reason why Skibiński and Grabowski have presented in 2004 a technique based on the idea of unbounded length contexts (PPM*)—variable-length contexts (VC [SG04]). This technique improves even the state-of-art PPM variations like PPMII but also works with for example

PPMD. This algorithm, comparing to an extended alphabet idea (PPMEA), achieves better compression effectiveness, it is much faster, and requires only one pass over input data.

The PPM with variable-length contexts (PPMVC [SG04]) algorithm virtually increases orders of maximum order contexts. One can say that PPMVC extends the ordinary, character-based PPM with string matching similar to the one used by the LZ77 algorithm. As there are several versions of the PPMVC algorithm, this subsection is divided into parts. We start from explaining the simplest version, which is later extended with new features. The implementation of the PPMVC algorithm is publicly available at Reference [Sk03].

3.2.1 PPMVC1 – the simplest and the fastest version

The PPMVC mechanism uses PPM contexts of maximum order only. The contexts do not have to be deterministic. If an order of context is lower, then the VC technique is not used, and the current symbol is encoded with a ordinary PPM model instead.

In PPMVC each maximum order context contains a pointer to a reference context (the last context's appearance). The right match length (RML) is defined as the length of the matching sequence between symbols to encode and symbols followed by the reference context. The pointer is updated after each appearance of the context, no matter if the context was successful or not in its prediction. This assumption goes in accordance with Mahoney's experiments [Ma02] intended to show that *q*-grams (that are contiguous sequences of *q* symbols) tend to reappear more often in close distances.

The PPMVC1 encoding process is simple. While the encoder uses a maximum order context, it evaluates RML and encodes its value using an additional global RML model (order 0). This is a significant difference to the coding variant from PPM* or PPMZ, as in PPMVC there is no need to encode all correctly predicted symbols one by one (with a possibility of appearance of escape symbol). The minimum length of the right match may be zero.

There are two more ideas in PPMVC1 intended to improve the compression effectiveness. First is the minimum right match length (*minRML*). If the current right match length is below the *minRML* threshold, then PPMVC1 sets RML to 0. This assures that short matches are not used, and they are encoded with a minimal loss. At the beginning, the *minRML* threshold is set to default value, dependent on the order of PPM model (it will be explained later). The *minRML* threshold is the same for all contexts of maximum order.

The second modification is to divide matched strings into groups and encoding only the group number. Groups are created by dividing RML by integer. For example, if the divisor is 3, then the first group contains matched strings of length 0, 1 or 2, and the second group contains strings of length 3, 4 or 5, and so on. In other words, the RML values are linearly quantized. Encoded RML must be bounded (e.g., the length of 255), but if the match length exceeds this bound, then we can encode the maximum RML and the remaining part as the next RML (possibly continuing this process).

The decompression is unambiguous as the decoder holds the same structures as the encoder and follows the steps of the encoder. Only the RML values have to be encoded explicitly, as the decoder does not know them.

The following pseudo code implements the main encoding loop. It takes as an input data to encode (**`char* data`**) and the size of data to encode (**`int len`**). It is worth to mention that the PPM algorithm in one step can encode only one symbol while the PPMVC algorithm can encode one or more symbols. As you can see, switching between PPM and PPMVC is unambiguous (for the encoder and the decoder) so there is no need to encode an additional information about switching.

```
void encode_data(char* data, int len)
{
     int enc_count;

     while (len>0)
     {
          active_context = find appropriate context in PPM model

          if (active_context->getOrder() < maximum_order)
          {
               enc_count=PPM_encode(active_context, data, len);
               update PPM model for active_context
          }
          else
          {
               enc_count=PPMVC_encode(active_context, data, len);
               update PPM model for all encoded contexts
          }

          data = data + enc_count;

          len = len – enc_count;
     }
}
```

The following pseudo code implements the PPMVC1 algorithm. It takes as an input the active context (**`Context* active_context`**), data to encode (**`char* data`**) and the size of data to encode (**`int len`**). Comparing to PPM, class **`Context`** consist of additional field **`pointer`**. PPMVC1 algorithm in one step can encode only one symbol (using **`PPM_encode()`**, when RML is equal to 0) or more symbols. As PPMVC doesn't encode an additional information about switching, so RML values must be explicitly encoded, even if RML is equal to 0 (but using RML model they are encoded with a minimal loss).

```
int PPMVC1_encode(Context* active_context, char* data, int len)
{
     reference_context = active_context->pointer;

     active_context->pointer = data;

     RML = the length of the matching sequence between symbols to
            encode and symbols followed by the reference context

     if (RML < minRML)
          RML = 0;

     RML = RML/d;

     Encode RML using an additional global RML model (order 0)

     if (RML == 0)
          return PPM_encode(active_context, data, len);

     return RML*d;
}
```

3.2.2 PPMVC2 – added minimal left match length

The left match length (LML) is defined as the length of the matching sequence between already encoded symbols and symbols that build the reference context (the last appearance of the active context). In the other words, this is an order of common context for the active context and the reference context. The minimum LML is always equal to the maximum PPM order, because this is the order of the reference and the active context.

PPMVC2 extends the previous version with a minimum left match length, which is based on a corresponding idea from PPMZ. For each context of the maximum order PPMVC2 keeps, besides a pointer to the reference context (the last context's appearance), the minimum left match length.

The PPMVC2 encoding is following. While the encoder uses a context of the maximum order, it evaluates LML using the active context and the reference context. If LML is below the minimal left match length (*minLML*) threshold defined for this context, then the encoder uses an ordinary PPM encoding (without emitting any escape symbol). If LML is equal to or greater then *minLML*, PPMVC2 evaluates and encodes RML like in the PPMVC1 algorithm.

Moreover, if RML is zero, then we are sure that the reference context was bad prediction for the active context. In this case we can increase value of *minLML* to better distinguish similar contexts in further predictions, what minimizes prediction errors. In PPMVC2 we propose to update *minLML* with the formula *minLML*=*minLML*+1. We can think about this formula as a very slow m*inLML* adaptation, when error occurs (RML is equal to 0). Experiments have showed that for PPMVC this approach gives better results than Bloom's instant error-defending formula (*minLML*=LML+1 for PPMZ, where LML is the currently evaluated left match length). At the beginning, the *minLML* value is set to default value, dependent on the maximum order (it will be explained later).

The decompression is unambiguous as the decoder holds the same structures as the encoder and follows the steps of the encoder. As the symbols that build the reference context are already encoded, both the encoder and the decoder can evaluate LML without additional information. Only the RML values have to be encoded explicitly, as the decoder does not know them.

The following pseudo code implements the PPMVC2 algorithm. It takes as an input the active context (**`Context* active_context`**), data to encode (**`char* data`**) and the size of data to encode (**`int len`**). Comparing to PPM, class **`Context`** consist of additional fields **`pointer`** and **`minLML`**.

```
int PPMVC2_encode(Context* active_context, char* data, int len)
{
     reference_context = active_context->pointer

     active_context->pointer = data;

     LML = the length of the matching sequence between already
             encoded symbols (active context) and symbols that build
             the reference context

     if (LML < active_context->minLML)
               return PPM_encode(active_context, data, len);
```

```
    RML = the length of the matching sequence between symbols to
            encode and symbols followed by the reference context

    if (RML < minRML)
         RML = 0;

    RML = RML/d;

    Encode RML using an additional global RML model (order 0)

    if (RML == 0)
    {
         active_context->minLML = active_context->minLML + 1;
         return PPM_encode(active_context, data, len);
    }

    return RML*d;
}
```

3.2.3 PPMVC3 – the most complex version with the best compression

PPMVC2 remembers for each context of maximum order only the last context's appearance (a pointer to a reference context). The experiments have showed that this approach gives good results, but sometimes using a more distant reference can give a better prediction (a longer right match).

There are several possibilities of taking into account not only the last context's appearance. One is to find the previous context's appearance with a maximum RML and to encode RML. But in this case we must also explicitly encode an index (a number) of the previous context's appearance to enable decompression. The encoding of the index must be effective enough not to deteriorate the compression effectiveness. The experiments have showed that in lots of cases the last (the most recent) context's appearance gives the best prediction (the longest right match). The skew distribution of indexes obviously implies their efficient encoding. The indexes can be efficiently encoded in the global index model (order 0), similar to the RML model. The last context's appearance has the index 0, the penultimate one has the index 1, and so on. In overall, this approach decreases the compression speed (finding of a pointer with a maximum RML) comparing to PPMVC2, but the decompression speed is unaffected.

Another possibility is to select a reference context that will be also known to the decoder. Good results should be achieved when we choose the previous context's appearance with the longest left match length (LML). The index encoding is not necessary when we choose the index of the previous context's appearance with the longest LML. The decoder can evaluate LML for each previous context's appearance without additional information, as the symbols that build the previous context's appearance are already encoded. Therefore the decoder can find the index of the previous context's appearance with the longest LML.

At the beginning the encoder has to find the previous context's appearance with the longest LML. Then the encoder works like PPMVC2 variation, that is, it has to check if the match length is equal to or exceeds *minLML*. If so, the encoder uses this context to find the RML and encodes the data. The decompression is unambiguous as the decoder holds the same structures as the encoder and follows the steps of the encoder. But the compression and the decompression speed is hampered in almost the same degree.

The main problem with the second approach (that is selecting of a previous context's appearance with the longest LML) is that the longest LML not always predicts the longest right match. But it turns out that this idea gives better results than a variant, in which the encoder would explicitly encode the index of the previous context's appearance with a longer (or equal) length of the right match.

The following pseudo code implements the PPMVC3 algorithm. It takes as an input the active context (**`Context* active_context`**), data to encode (**`char* data`**) and the size of data to encode (**`int len`**). Comparing to PPM, class **`Context`** consist of additional fields **`list of pointers`** and **`minLML`**.

```
int PPMVC3_encode(Context* active_context, char* data, int len)
{
     reference_context = select the context with the longest LML
             from the list of previous context's appearances

     Add active_context context to the list of previous context's
             appearances

     LML = the length of the matching sequence between already
             encoded symbols (active context) and symbols that build
             the reference context

     if (LML < active_context->minLML)
               return PPM_encode(active_context, data, len);

     RML = the length of the matching sequence between symbols to
             encode and symbols followed by the reference context

     if (RML < minRML)
          RML = 0;

     RML = RML/d;

     Encode RML using an additional global RML model (order 0)

     if (RML == 0)
     {
          active_context->minLML = active_context->minLML + 1;
          return PPM_encode(active_context, data, len);
     }

     return RML*d;
}
```

3.2.4 Selecting parameters

There are four parameters in PPMVC: the divisor *d* for a match length quantization, *minRML*, *minLML*, and the number of pointers (previous context's appearances) per context (only for PPMVC3). These parameters mostly depend on the maximum PPM order (*maxOrder*). As higher the order, as better the PPM effectiveness in deterministic contexts (e.g., about 1 bpc in the order 8 (exclusively) for textual data in the PPMII compression), so the parameters ***d***, *minLML* and *minRML* must respectively be greater.

Our experiments with PPMVC based on PPMII have showed that the best compression effectiveness on average can be obtained if *d* is set to *maxOrder*+1. The right match length depends on the divisor *d*, and *minRML* was set to 2*d*. *MinLML* was chosen experimentally as 2·(*maxOrder*–1).

The maximum number of pointers (previous context's appearances) depends also on the maximum PPM order. In higher orders contexts are better distinguished, and there is no need to hold as many pointers as in lower orders. There are files for which the best compression ratios are obtained having from 32 to 64 pointers for the order 4 (e.g., *nci* from Silesia corpus [De03b]), but we suggest to hold 8 pointers, which results in the best compression performance on average (tradeoff between compression speed and compression effectiveness). For the order 8 we use 4 pointers and only 2 pointers for the order 16.

3.2.5 Differences between PPMZ and PPMVC

The memory requirements for the PPMVC algorithms are not very high. PPMVC1 needs only 4 bytes (for the pointer) for each context of maximum order. PPMVC2 uses 5 bytes (the pointer and 1 byte for *minLML* value) here, while PPMVC3 requires 4*p*+1 bytes (where *p* is the count of pointers).

At the end of this subsection we would like to point out the main differences between PPMZ and PPMVC (PPMVC2 and PPMVC3):

- PPMVC simulates unbounded contexts using deterministic and non-deterministic contexts of maximum order; PPMZ simulates unbounded contexts using additional, deterministic contexts of constant (equal to 12) order,
- PPMVC encodes only the length of well-predicted symbols; PPMZ encodes all predicted symbols one by one,
- PPMVC and PPMZ update the minimum left match length in a different manner.

3.3 Results of experiments

In this section, we present comparison of our two predictive compression algorithms (PPMEA and PPMVC) and PPMII, which is a base for these methods. PPMEA method extends PPM alphabet to long repeated strings. PPMVC method extends the PPM algorithm with string matching abilities, similar to the one used by the LZ77 algorithm.

3.3.1 Experiments with the Calgary corpus

At the beginning, we have carried out our first experiments on a well-known set of files—the Calgary corpus [BC+90], fully described in Appendix A.

Table 3.1 shows results of experiments with the extended alphabet algorithm. The experiments were carried out using author's PPMD implementation of the PPMD algorithm as described in [Ho93]. We also included results of PPMII, based on the author's implementation of the PPMII algorithm (that has a little worse compression performance than Shkarin's implementation [Sh02b]) as described in [Sh02a]. The count of symbols added into extended alphabet and an appearance of added symbols for both PPMD and PPMII are the same, because these are evaluated before starting of the PPM compression.

File	PPMD order 5	PPMD order 5 with extended alphabet	PPMII order 5	PPMII order 5 with extended alphabet	Count of symbols added into alphabet	Appearance of added symbols
bib	1.876	1.860	1.785	1.772	28	213
book1	2.275	2.275	2.214	2.214	1	2
book2	1.952	1.949	1.898	1.895	27	248
geo	4.832	4.829	4.510	4.506	3	45
news	2.368	2.343	2.280	2.259	100	775
obj1	3.785	3.791	3.656	3.663	5	17
obj2	2.429	2.393	2.337	2.305	85	505
paper1	2.335	2.336	2.238	2.239	2	7
paper2	2.303	2.301	2.210	2.209	2	7
pic	0.807	0.811	0.792	0.786	113	5488
progc	2.385	2.382	2.268	2.266	2	5
progl	1.682	1.630	1.603	1.557	34	182
progp	1.717	1.638	1.613	1.547	30	71
trans	1.496	1.420	1.381	1.331	77	352
average	2.303	2.283	2.199	2.182	36	565

Table 3.1: Results of experiments with PPMD, PPMII and PPMD, PPMII with extended alphabet on the Calgary corpus. Results are given in bits per character (bpc).

As can be seen in Table 3.1, alphabet extensions in order 5 affect much only the three last files (*progl*, *progp*, *trans*). The extended alphabet algorithm improves the average compression effectiveness for about 1% for both PPMD and PPMII. In lower orders improvement is much bigger, but in higher orders improvement is lower.

File	Shkarin’s PPMII order 5	PPMVC1 order 5	PPMVC2 order 5	PPMVC3 order 5
bib	1.752	1.743	1.731	1.730
book1	2.190	2.191	2.190	2.190
book2	1.875	1.869	1.863	1.863
geo	4.346	4.346	4.343	4.344
news	2.243	2.221	2.208	2.205
obj1	3.537	3.551	3.539	3.540
obj2	2.279	2.217	2.217	2.212
paper1	2.219	2.208	2.206	2.205
paper2	2.186	2.185	2.182	2.182
pic	0.767	0.755	0.750	0.749
progc	2.250	2.240	2.233	2.230
progl	1.576	1.513	1.493	1.490
progp	1.604	1.493	1.481	1.473
trans	1.365	1.274	1.265	1.255
average	2.156	2.129	2.121	2.119
ctime	1.92s	2.25s	2.50s	3.20s

Table 3.2: Results of experiments with PPMII and PPMVC1, PPMVC2, PPMVC3 on the Calgary corpus. Results are given in bits per character (bpc). The compression time (ctime) is given in seconds.

Table 3.2 shows results of experiments with PPMII and PPMVC algorithms. The experiments were carried out using Shkarin's implementation of the PPMII algorithm, which is confusingly called PPMd [Sh02b]. PPMVC was implemented as extension of Shkarin's

PPMII. This implementation is publicly available at Reference [Sk03] and described in Appendix D of this dissertation.

The results presented in Table 3.2 indicate that, as we expected, PPMVC3 gives the best compression, PPMVC1 is the fastest but also has worst compression of the three variants, and PPMVC2 is in between. From the point of practical use (also taking into account the memory requirements), PPMVC2 seems to be the best choice. The PPMVC2 algorithm improves the average compression effectiveness of PPMII for about 2%.

We believe that the Calgary Corpus files are too small to get a full potential from the extended alphabet and the PPMVC algorithm, so we decided to perform similar experiments on the Canterbury corpus.

PPM order	Shkarin's PPMII compression effectiveness	Shkarin's PPMII memory usage	Shkarin's PPMII compression time	PPMVC2 compression effectiveness	PPMVC2 memory usage	PPMVC2 compression time
order 2	2.760	2 MB	1.28s	2.530	3 MB	2.04s
order 3	2.366	3 MB	1.42s	2.254	4 MB	2.19s
order 4	2.210	5 MB	1.67s	2.150	6 MB	2.36s
order 5	2.156	7 MB	1.92s	2.119	8 MB	2.50s
order 6	2.134	10 MB	2.19s	2.109	11 MB	2.66s
order 8	2.114	17 MB	2.42s	2.101	18 MB	2.79s
order 10	2.107	19 MB	2.54s	2.099	20 MB	2.83s
order 12	2.103	21 MB	2.67s	2.098	22 MB	2.88s
order 16	2.100	23 MB	2.79s	2.097	24 MB	2.93s

Table 3.3 Results of experiments with PPMII and PPMVC algorithms on the Calgary corpus. Results are given in bits per character (bpc).

In the following experiments, we have experimented only with PPMVC2. As can be seen in Table 3.3, the compression speed is slightly worse that of Shkarin's PPMII. The compression speed is lower especially in lower orders, where an improvement in the compression performance is higher. Memory usage is almost the same (always about 1 MB higher than Shkarin's PPMII). The most interesting thing is that PPMVC2 order 8 achieves almost the same compression performance that Shkarin's PPMII in order 16, at the same compression speed. Nevertheless, PPMVC2 requires 5 MB of memory less.

3.3.2 Experiments with the Canterbury corpus and the large Canterbury corpus

The next experiments were carried out on a combination of the Canterbury corpus and the large Canterbury corpus, which are fully described in Appendix B.

As can be seen in Table 3.4 author's implementation of PPMII is about 2% worse than Shkarin's. This difference comes from different PPM model initialization (not described by Shkarin in [Sh02a]) and improvements added by Shkarin for binary files (the biggest difference can be seen on the file *kennedy.xls*).

PPMEA improvement over the PPMII algorithm is similar to the results achieved in the previous experiments. PPMEA improves the average compression effectiveness for over 1% in comparison to PPMII. PPMVC also improves the average compression effectiveness of Shkarin's PPMII for about 1%. But if we take into consideration only files from the large Canterbury corpus (*bible.txt*, *E.coli* and *world192.txt*) PPMEA improvement over the PPMII algorithm is equal over 3% and PPMVC improvement over Shkarin's PPMII is about 2%.

File	PPMII order 5	PPMII order 5 with extended alphabet	Shkarin's PPMII order 5	PPMVC2 order 5
alice29.txt	2.079	2.081	2.054	**2.049**
asyoulik.txt	2.346	2.345	2.314	**2.310**
bible.txt	1.564	1.547	1.529	**1.492**
cp.html	2.176	2.197	**2.151**	2.170
E.coli	1.944	**1.918**	1.961	1.958
fields.c	1.927	**1.904**	1.944	1.908
grammar.lsp	2.288	**2.285**	2.328	2.326
kennedy.xls	1.420	1.420	**0.987**	**0.987**
lcet10.txt	1.850	1.845	1.825	**1.815**
plrabn12.txt	2.220	2.219	2.196	**2.193**
ptt5	0.793	0.790	0.767	**0.750**
sum	2.566	2.490	2.470	**2.401**
world192.txt	1.460	**1.342**	1.436	1.387
xargs.1	2.852	**2.852**	2.865	2.869
average	1.963	1.945	1.916	**1.901**

Table 3.4 Results of experiments with PPMII, PPMEA and PPMVC algorithms on the Canterbury and the large Canterbury corpus. Results are given in bits per character (bpc).

4 Reversible data transforms that improve compression effectiveness

The preprocessing (preliminary processing) can be considered as a part of the modeling stage in lossless data compression. It can be divided into two stages: *analysis* and *transform*. The analysis stage determines kind of data or gathers some information that is helpful in a further modeling. If we know kind of data, we can use specialized methods, which usually perform better than universal techniques. The second stage is optional; it reversibly transforms a data into some intermediate form, which can be compressed more efficiently. In this case, the reverse process has two stages, the decompression using given compressor and the reverse preprocessing (postprocessing) transform. Sometimes the transform stage is an integral and inseparable part of the compression algorithm (for example, the DCT stage in the JPEG [Wa91] image compression).

From a scientific point of view, a field of the universal compression is almost closed, and nowadays researches are focused on methods specialized only for one kind of data. Thus, many papers about specialized compression algorithms have appeared. They explore specific properties of, for example, text [FK$^+$00, Gr99, SM$^+$03], XML [AN$^+$04], platform-dependent executable files [DK02], and record-aligned data [VV04]. Specialized compressors are potentially more powerful, both from the viewpoint of compression effectiveness and compression speed, as there are virtually no restrictions imposed on the implemented ideas. Nevertheless, preprocessing is more flexible, as the transformed data can be compressed with most of existing universal compressors, so the compression effectiveness can be improved without knowing any details about a compressor used in the compression stage. Moreover, in a field of the universal data compression, preprocessing methods can be separated from a compressor or an archiver. They are usually quite simple to implement and can be combined into a *preprocessor*—a standalone computer program. In this dissertation we deal with this case as we believe that this approach is more flexible and universal than building preprocessing methods into a compressor. There is also a possibility to implement specialized methods into a single compressor, what is inflexible and much more complicated than creating a preprocessor.

In this chapter, we present most of well-known nowadays reversible data transforms that improve effectiveness of universal data compression algorithms. The biggest emphasis is placed on texts in natural languages as this field is most developed and an improvement in the compression effectiveness is the biggest. For each kind of data, we try to introduce this kind of data, then we briefly present specialized methods and finally we describe preprocessing methods.

4.1 Fixed-length record aligned data preprocessing

Many kinds of files consist of continuous sequence of records with a fixed length. The simplest example is a 24-bit image, which consist of sequence of RGB (red, green, and blue) values “r_1 g_1 b_1 r_2 g_2 b_2 r_3 g_3 b_3 … r_n g_n b_n”. Fixed-length record aligned data can be seen as a table or a two-dimensional array of bytes. In record-based data, a dependency between sequences of bytes in columns is usually higher than dependency between sequences of bytes in rows. Fixed-length record aligned data preprocessing utilizes this property and transposes data by the record length. Using our previous example, the record length is equal to three bytes and transposed data (RGB values in this case) are “r_1 r_2 r_3 … r_n g_1 g_2 g_3 … g_n b_1 b_2 b_3 …

b_n". If above-mentioned property is fulfilled, then transposed data is compressed more effectively with universal lossless compressors.

In a field of database management an idea of transposed files is known since over 30 years [Ho75]. An *attribute-transposed file* [Ba79] is a collection of files, called subfiles, in which each subfile contains selected attribute data for all records. In this way the contents of each record are distributed over all subfiles. Transposed files are designed to reduce data transfer costs. One of the subfiles corresponds to a primary record. While processing queries, the primary record is accessed as the first, and other subfiles are accessed only if necessary. Transposed files give additional side effect, which is a significant effectiveness improvement of query processing.

In 1998 Bloom in his paper about PPMZ [Bl98] noticed that the file *geo* from the Calgary corpus [BC+90] contains a list of 32-bit floating-point numbers, in which the first byte of each number is always zero. This file can be compressed more effectively when it will be transposed by the record length, which is equal to four. Nevertheless, Bloom did not present an algorithm to detect a record length and to decide if a file should be transposed.

Buchsbaum et al. [BC+00] have observed that a significant compression improvement could be achieved by partitioning the table (seen as two-dimensional array of bytes) into disjoint and contiguous intervals of columns and by compressing each interval separately. The partition was generated by a preprocessing training procedure, and the resulting compression strategy was applied to a universal lossless compressor (gzip was the preferred choice). Buchsbaum et al. also observed that rearrangement of the columns by grouping dependent columns improves the compression effectiveness even more. This work was extended in 2002 by Buchsbaum et al. [BF+02].

Vo and Vo in their paper [VV04] have continued work of Buchsbaum et al. They developed the *RecordLength* heuristic to automatically deduce the table structure of a dataset by computing its number of columns (which is equal to the record length). They also presented a preprocessing technique to automatically learn the dependency among the columns of a table, called 2–transform. This scheme transforms the data to a form, which is more compressible with universal lossless compression methods. Vo and Vo have combined it with the predictive Move-to-Front transform, the Run-Length Encoder and the Huffman coder.

Abel, simultaneously to Vo and Vo, has presented his record preprocessing technique [Ab04] predestined to the universal lossless compression. This record preprocessing method consists of three stages: a detection of a record length, a determination of transposition, and a transposition by the record length. The second stage, counting the variance of the most used trigram frequencies, determines if the transposed file is better suitable for the compression than the raw file, so we do not need to assume that the input is always fixed-length record aligned data as in above-presented schemes. On the other hand, the record preprocessing scheme does not take advantage of dependency among columns and does not rearrange the columns.

The detection of a record length presented by Abel is simpler and more elegant than the scheme presented by Vo and Vo, but it is designed especially for fixed-length record aligned data with a very short record length. It is based on an idea that symbols repeat at the same position within the succeeding records. This heuristic detects the most frequent distance between the same symbols. The following C++ function implements this algorithm. It takes as an input the length of a file (**`int fileLen`**) and the content of the file as an array of bytes (**`char* m`**). The function returns **`fc_max`**, which is a detected length of the record in the input file.

```
int recordLength(char* m, int fileLen)
{
     int i, fc[MAX_RECORD_LEN], fc_max, lastPosition[256];

     for (i=0; i < 256; i++)
          lastPosition[i]=0;
     for (i=0; i < MAX_RECORD_LEN; i++)
          fc[i] = 0;
     fc_max = 0;

     for (i=1; i < fileLen; i++)
     {
          if (m[i-1] != m[i])
          {
               int distance = i - lastPosition[m[i]];
               if (distance < MAX_RECORD_LEN)
               {
                    fc[distance]++;
                    if (fc[distance] > fc[fc_max])
                         fc_max = distance;
               }
               lastPosition[m[i]] = i;
          }
     }
     return fc_max;
}
```

This scheme remembers for each symbol the last occurrence of a symbol (**`lastPosition[]`**) and calculates the distance between the current position and the last occurrence (**`distance`**). If this distance exceeds two, then the frequency counter **`fc[distance]`** for this distance is increased. The distance for which **`fc[distance]`** reaches its maximum value represents the detected record length of the transposition.

The record preprocessing method presented by Abel is designed especially for fixed-length record aligned data with a very short record length as uncompressed TIFF images, uncompressed BMP images, the file *geo* from the Calgary corpus. Of course, this technique can be combined with every universal lossless compression algorithm.

4.2 Audio data preprocessing

Most specialized schemes for audio compression, for example MPEG-1 Layer 3 (MP3 [BS94]), MPEG-2 Advanced Audio Coding (AAC [BB$^+$96]), Dolby AC-3 [TD$^+$94], are lossy. They exploit limitation of a human ear perception and attempt to evaluate the irrelevant components of the audio signal that fall outside the hearing threshold. These techniques are referred as the *perceptual coding*. Lossless schemes for the audio compression are not so popular. The main drawback of these schemes is lower compression ratio compared to the lossy audio compression. Lossy compressors achieve compression factors ranging from 2 to 24 or even higher (depending on the audio quality), while lossless compressors usually achieve compression factors from 2 to 3.

Most of audio compression algorithms, regardless of lossy or lossless, have a preprocessing stage prior to a statistical coding. In this stage, the signal is decorrelated by using a linear prediction or a linear transform in order to compact the signal into a few uncorrelated coefficients. Related with the modeling method used to decorrelate a signal, audio compression schemes are divided into two categories, the ***predictive*** coding and the ***transform-based*** coding.

The predictive audio compression makes a prediction of the current sample on the basis of linear combination of previous samples. Only the difference (the residue) between the estimate of a linear predictor and the original signal is transmitted. In the lossy predictive audio compression, the residues resulted from predictive coding can be quantized to reduce the size of output data.

The transform-based audio compression is usually based on orthogonal linear transforms. The most known examples of such transforms are the Discrete Fourier Transform (DFT), the Discrete Cosine Transform (DCT) and the Karhunen-Loeve Transform (KLT). Currently, the most used transform is the Modified Discrete Cosine Transform (MDCT). Another group of transforms is the wavelet transform (WT), which splits the signal into different frequency components and processes each component with a resolution matched to its scale. The transform-based audio schemes usually compacts the signal into a few uncorrelated floating-point coefficients. For an efficient encoding, the coefficients have to be quantized by using either a scalar or a vector quantization method, which causes irreversible errors in the output signal. The lossless transform-based audio compression encodes along with the coefficients the coding error, which is difference between the original signal and its approximation caused by quantization.

Experiments performed by Kim [Ki04] have showed that lossless predictive methods outperform lossless transform-based methods (for example, LTAC [LP$^+$99]). Nevertheless, transform-based methods achieve better effectiveness in lossy audio compression. The most popular lossless audio compressors are nowadays: Lossless Predictive Audio Compression (LPAC [Li02, Li04]), Monkeys Audio Compressor (MAC [As04]), Free Lossless Audio Coder (FLAC [Co05]), Lossless Audio (La [Be04]), and OptimFROG [Gh04]. They all are predictive.

Apart from specialized audio compression methods, there is simple and quite effective preprocessing technique called a *delta coding* or a *differential coding*. This technique utilizes a property that successive samples usually differ insignificantly. If we replace absolute values with differences between successive samples, then the compression effectiveness of universal lossless compressors will be improved. Much better results are achieved with lower quality (8-bit/sample) than with high quality (16-bit/sample) signals, which are usually split into two 8-bit/sample signals. Moreover, as higher the sample rate, as better the compression effectiveness. For example, 16-bit stereo (2 channels) audio can be considered as four 8-bit signals, and the signal ‘1054226634784589’ will be transferred to ‘1054121212121111’ by subtracting the byte four units before the current byte. This technique is widely used in modern compressor like RAR [Ro04], GRZipII [Gr04], UHARC [He04], ACE [Le03], DC [Bi00].

4.3 Image data preprocessing

Images can be divided into three main groups: bi-level, palette-based, and full-color. Bi-level images consist of only two colors (usually black and white). There are several specialized compression methods for bi-level images, for example, JBIG, JBIG2 [HK$^+$98], JB2 [BH$^+$98a], and fax protocols CITT Group 3 [IT80], CCITT Group 4 [IT88], to name the most important ones only. All these methods are predictive. Additionally, modern algorithms (like JBIG2 and JB2) segment an image into different regions and use pattern-matching techniques to efficiently compress the textual regions. The second group of images are palette-based images. A palette-based image consists of two parts: an index of the colors used in the image (the palette) and image information composed of a series of palette indexes. The most known algorithms predestined to the compression of palette-based images are dictionary-based GIF (uses a LZW variation) and PNG [Cr95] (uses LZ77 variation). Nevertheless, for this kind of images, the best compression effectiveness is achieved with specialized predictive methods like RAPP [Ra98] and PWC [Au00]. The last and the biggest group of images are full-color images. They can be considered as two-dimensional table of 24-bit pixels. Each pixel consists of three 8-bit color components (red, green, and blue). Nevertheless, most of image compression algorithms treat the full-color image as a set of independent grayscale images, which consist of a single 8-bit color component. Audio data and grayscale images have similar characteristic, and all the methods presented in the previous section (the predictive coding, the transform-based coding, and the differential coding) can be successfully applied to grayscale and full-color images.

Specialized methods for the full-color image compression can be divided into the predictive coding, the transform-based coding, and the fractal coding. The most known methods for full-color image compression, like JPEG [Wa91] and JPEG-2000 [CS$^+$00], are lossy and achieve compression factors up to 32 with satisfying image quality. The lossless methods achieve compression factors of 2 to 3, what is analogous to the lossless audio compression. Also as in the case of the lossless audio compression, lossless predictive methods outperform the lossless transform-based methods like SPIHT [SP96] and lossless JPEG 2000 (which is based on the Discrete Wavelet Transform (DWT)). Specialized lossless predictive methods for the full-color image compression, for example, FELICS [HV93], CALIC [WM97], JPEG-LS [WS$^+$00], SICLIC [BF99], consider an image as two-dimensional table of pixels and make a prediction of the current pixel on the basis of the pixels which have already been processed. For example, the JPEG-LS algorithm uses only four pixels as a context for the prediction—the pixel to the left (W), above and to the left (NW), above (N), and above and to the right (NE). Of course, only the difference (the residue) between a predicted value and a value of the original pixel is transmitted. The last group of specialized methods for the full-color image compression is the fractal coding. A fractal image compression [WJ99] is a computationally intensive method of the compression, which finds self-similarities within images. In the fractal compression, an image is partitioned into a number of regions, each of which is approximated by some appropriate part of the same image.

In this dissertation we are interested especially in reversible transforms that improve compression effectiveness of universal compression algorithms. The differential coding, a preprocessing technique presented in the previous section, can be used also with grayscale and full-color images. Moreover, for full-color images this technique can be also combined with the fixed-length record aligned data preprocessing, presented in this chapter.

The histogram packing [FP02] is a preprocessing technique for images that do not use the complete set of available intensities (of colors or tones of gray) that improve the compression effectiveness of state-of-the-art lossless image compression methods (JPEG-LS, lossless JPEG-2000, or CALIC). This technique transforms the image to the form, which has lower total variation and it is easier to compress by lossless methods. The reverse transform requires additional information, more precisely—the packing table, as the histogram packing is image-dependent. The authors have reported about 20% improvement in combination with JPEG-LS, lossless JPEG-2000, and CALIC. They did not check how this transform influence on compression of images using universal lossless compression algorithms, but we believe that it will improve the compression effectiveness.

4.4 Executable file preprocessing

Executable files can be compressed with universal data algorithms, but better effectiveness can be achieved using specialized methods. These methods can be divided into two classes: a *code compaction* and a *code compression*. The first class of methods produces files that are executed or interpreted directly, without any decompression. The second class produces files smaller than those obtained using the code compaction, but files must be decompressed to their original form before they can be executed.

The code compression methods (for example, wire code [EE+97], Clazz [HC98], Jazz [BH+98b], Fraser's IR compressor [Fr98], Pugh's compressor [Pu99], SSD [Lu00], PPMexe [DK02]) usually separate file into different streams, which are compressed independently. These methods also take advantage of a file structure. For example, PPMexe explores syntax and semantics of executable files and combines PPM with two preprocessing steps: instruction rescheduling to improve prediction rates and partitioning of a program binary into streams with a high auto-correlation.

Most of the work on the code compaction treats an executable program as a simple linear sequence of instructions (for example, BRISC [EE+97]) and eliminates repeated code fragments. Recent work on the code compaction works with a control flow graph of the program (Squeeze [DE+00]) and reuses whole identical or nearly identical procedures (Squeeze++ [SB+02]) found in programs written in object-oriented languages such as C++. Code compaction methods can be considered as transparent, but irreversible transforms that do not influence on the semantics of the program.

Recently, with growing popularity of Java, transparent irreversible transforms for Java class files have appeared. Java class files are highly redundant, therefore there are many code compactors (for example, Jax [LT+98], DashO [Pr04], JShrink [Ea02], ProGuard [La04], and BIM Obfuscator [Fa03]) that strip out debugging information and perform shrinking and obfuscation by renaming classes, methods and fields to have short, meaningless names. Some of these algorithms can remove unused Java classes, methods, and fields, sort entries in the constant pool according to type, and sort UTF constants according to their content.

Besides the code compression and the code compaction methods there is also a well-known reversible transform for 32-bit Intel 80x86 executable files [FP97]. It works by replacing relative addresses after a CALL instruction (hexadecimal E8) with absolute addresses. The algorithm is very simple. Each CALL byte sequence (E8 followed by 32 bit offset) in the form of "E8 r_0 r_1 r_2 r_3" is replaced by translated CALL byte sequence "E8 a_0 a_1 a_2 a_3" using the following relative-to-absolute conversion:

```
relativeOffset =  $r_0 + r_1*2^8 + r_2*2^{16} + r_3*2^{24}$;
absoluteOffset = relativeOffset + currentPosition;
$a_0$ = bits 0–7 of absoluteOffset;
$a_1$ = bits 8–15 of absoluteOffset;
$a_2$ = bits 16–23 of absoluteOffset;
$a_3$ = bits 24–31 of absoluteOffset;
```

The algorithm uses a *currentPosition* variable, which is the current offset within uncompressed data. The transform is performed "in place", as the relative-to-absolute conversion directly maps from a 32-bit value to a 32-bit value. This transform is used, for example, in LZX [FP97], DC [Bi00], RKC [Ta04], and Durilca [Sh04].

4.5 DNA sequence preprocessing

The deoxyribonucleic acid (DNA) contains genetic information about properties of living organisms. DNA sequences consist only of four nucleotides, which are usually represented as a four letters: ‘a’, ‘c’, ‘g’, and ‘t’. These sequences can be considered as texts over a four-symbol alphabet. For such a small alphabet is trivial to compress the input data to 2 bpc, but is very hard to compress better than 2 bpc. Universal lossless data compressors, such as gzip and bzip2, usually do not cope with DNA sequences and achieve compression effectiveness worse than two bits per character. Only predictive compressors (based on the PPM algorithm or the PAQ algorithm) are able to achieve less than two bits per character.

There are two main approaches to the compression of DNA sequences: dictionary-based and grammar-based. Specialized dictionary-based methods for the DNA compression utilize an idea of the LZ77 algorithm, and they encode repeats as an offset to previous occurrence and a match length. Grammar-based schemes are based on the idea of succinct representing input data by a context-free grammar that generates only these data, and they are fit especially for DNA compression.

Grumbach and Tahi [GT93] have proposed the first specialized dictionary-based compression algorithm for DNA sequences, called Biocompress. It detects exact repeats and palindromes, and then encodes them as an offset to the previous occurrence and a match length. Biocompress-2 is based on its predecessor, with the addition of the order-2 arithmetic encoding used if no significant repeat is found.

Rivals et al. [RD+96] have presented another compression algorithm for DNA sequences—Cfact, which searches the longest exact repeats in an entire DNA sequence. Cfact is based on Biocompress except that it is a two-pass algorithm. It builds a suffix tree in the first pass. In the second pass, the repeats (found using the suffix tree) are encoded with a guaranteed gain. Otherwise, the data are trivially encoded on two bits per symbol.

GenCompress [CK+99] was the first DNA compressor that encode approximate repeats. The approximate repeats are repeats that contain errors, but can be transformed to exact repeats using a combination of copy, replace, insert, or delete operation on a few symbols. To check usefulness of the approximate repeats in the DNA compression, Chen et al. [CK+99] defined a measure of “relatedness” between two DNA sequences. GenCompress achieves significantly higher compression ratios than both Biocompress and Cfact. Unfortunately, searching for approximate repeats takes much longer time than searching for exact repeats and requires a large amount of memory.

After GenCompress, more specialized dictionary-based compression algorithms have appeared. There are CTW+LZ [MS+00], DNACompress [CL+02], NML-Comp [TK+03], and FastDNA [MR04b], to name the most important ones only. All the algorithms encode approximate repeats except the last one, which depart from this strategy by searching and encoding only exact repeats. FastDNA achieves a worse compression ratio than most of DNA compressors, but is very fast and low memory consuming, which makes it possible to compress very long DNA sequences, well beyond the range of any DNA compressor. DNACompress achieves one of the best compression ratios with a good compression speed and seems the current leader in DNA compression field.

Grammar-based schemes are based on the idea of succinct representing input data by a context-free grammar that generates only these data. The most known grammar-based compression algorithm is Sequitur [Ne96], which forms a grammar from a sequence in a single pass. Each repeat creates a rule in the grammar and is replaced by a non-terminal symbol producing a more concise representation of the sequence. Sequitur was improved by Kieffer and Yang in their algorithm, sometimes referred as Sequential [YK00].

Several other grammar-based algorithms have been proposed. They differ the way of creating the grammar. Greedy Off-line Textual Substitution [AL98] in each step selects repeat that reduces a size of the grammar as much as possible. Re-pair [LM99] always selects a pair of symbols that appear most often without overlap. Longest match [KY00] recursively selects the longest repeat, which occur two or more times. Multilevel Pattern Matching [KY+00] partitions input data into equal parts and creates a nonterminal in the grammar for each distinct part. At each level, the parts represented as nonterminals are joined to create distinct next-level nonterminals. Finally, they create a grammar that generates input data.

We have presented specialized methods for DNA compression, but in this dissertation we are especially interested in reversible transforms that improve the compression effectiveness of universal compression algorithms. DNA sequences use four symbol alphabet, and it is obvious that each symbol can be encoded with a fixed-length code on 2 bits. We can, for example, assign binary code ‘00’ for ‘a’, ‘01’ for ‘c’, ‘10’ for ‘g’, and ‘11’ for ‘t’. This transform stores four symbols in one byte, so an output file is four times shorter that an input file. Manzini and Rastero called this transform “4-in-1 preprocessing” [MR04b]. They have showed that this transform improves the compression effectiveness and the compression speed of universal lossless data compressors. Finally, gzip and bzip2 achieve compression effectiveness better than two bits per symbol on DNA sequences. We can add that this transform improves also the compression effectiveness of some PPM-based algorithms.

The following C++ function implements the above-mentioned transform. It takes as an input a pointer to an input file (**`FILE* file`**), the length of an input file (**`int fileLen`**), and a pointer to an output file (**`FILE* fileout`**).

```
void quarter_byte_encode(FILE* file, int fileLen, FILE* fileout)
{
      int c,d;

      d=fileLen%4;
      fputc(d,fileout);

      do
      {
            d=0;
            for (int i=0; i<4; i++)
            {
                  c=fgetc(file);
                  switch (c)
                  {
                        case EOF:
                        case 'a': d=0+(d<<2); break;
                        case 'c': d=1+(d<<2); break;
                        case 'g': d=2+(d<<2); break;
                        case 't': d=3+(d<<2); break;
                  }
            }
            fputc(d,fileout);
      }
      while (c!=EOF);
}
```

4.6 XML data preprocessing

The eXtensible Markup Language (XML) data comprises hierarchically nested collections of elements, where each element is represented by a *start tag* (for example, ‘<element>’) and an *end tag* (for the previous example, ‘</element>’). Start tags can have associated attributes with values (for example, ‘<element attribute1=”value1” attribute2=”value2”>’). Elements can also contain plain texts, comments, and special instructions for XML processors.

The XML data can be compressed with any compressor capable of compressing textual data, but that kind of compressor does not take advantage of the XML data structure. Thus, specialized XML compressors have appeared. The first was XMill [LS00], which splits data into three components: elements and attributes, character data, and a document structure. Each of these groups is separately compressed using the universal lossless compression algorithm. Cheney has showed [Ch01] that this transform improves the compression effectiveness only with LZ77-based compressors. Another XML compressor, XMLPPM [Ch01], refines the XMill idea by using different PPM models with four different XML components: element and attribute names, an element structure, attributes, and character data. Nevertheless, XMill and XMLPPM are not suitable for direct access to XML data. XGrind [TH02], Xpress [MP$^+$03], and LZCS [AN$^+$04] were created to overcome this problem. They are XML compressors that permit querying to be directly carried out on compressed XML data.

Recently, the next XML compressor—SCMPPM [AF$^+$04]—has appeared. It bases on the Structural Contexts Model (SCM [AN$^+$03]), which takes advantage of the context information usually hidden in the structure of a text. SCMPPM combines the PPM technique with an SCM idea, which uses a separate semi-adaptive model (it was originally tested using the word-based Huffman) to compress a text that lies inside each different XML tag and introduces a heuristic that merges similar models. Adiego et al. have reported [AF$^+$04] that SCMPPM compresses better than XMLPPM, the undisputed leader in the XML compression. It is worth to mention that SCM and SCMPPM are predestined for semistructured documents, not only for the XML data.

The Simple API for XML (SAX) is an interface, which produces a series of events that describe XML data, using a concept typical for Java. The SAX parser generates the events like the beginning or the end of an element, an appearance of character data. For example, the SAX parser in parsing the fragment '<element attribute1="valueA" attribute2="valueB">some_text<\element>' would report events:

```
startElement("element",("attribute1","valueA"),("attribute2","valueB"))
characters("some_text")
endElement("element")
```

Cheney in his paper about XMLPPM [Ch01] has presented a simple transform for XML data, which utilizes the SAX parser and improves the compression effectiveness of universal lossless compressors. The SAX parser was used to create the more concise non-textual structure, called Encoded SAX (ESAX), from which a decoder can reconstitute an XML document equivalent to the original. An ESAX encoder uses a simple word-based compression, where element tags and attribute names are transmitted only once in the original form. They are maintained in a dictionary and encoded as codewords at next occurrences. As an example, let us assume that the word 'element' from the previous example has assigned the codeword '10', the word 'attribute1' has assigned the codeword '11', and the word 'attribute2' has assigned the codeword '12'. The fragment '<element attribute1="valueA" attribute2="valueB">some_text<\element>' would be encoded as '10 11 valueA 0 12 valueB 0 2 1 some_text 0 2', where '0' means "end of character data or an attribute value", '1' means "start of character data", and '2' means "end of start/end element tag".

As one can see on the foregoing example, start tags, end tags, and attribute names of an element can be encoded using single bytes. The single bytes can be also used to indicate events such as "begin/end of character data", "end of start/end element tag", "begin/end of comment", and so on. There is, however, one big problem; the size of a dictionary grows during SAX parsing, and a codeword alphabet is limited to less than 256 codewords. If codewords are exhausted, then we can purge the dictionary using, for example, one of several methods of purging the dictionary applied with the LZW algorithm.

The ESAX idea is similar to the Wireless Binary XML (WBXML [WA01]), which was developed to reduce the network load and to simplify work of mobile device. It is done by parsing XML data in a WAP gateway and sending a binary file to the device. The ESAX idea is also similar to Millau [GS00], which was designed for efficient encoding and streaming of XML structures. Millau extends WBXML with separation of the structure and the content. It encodes the structure using the WBXML encoding and the content using universal lossless compression techniques.

4.7 Textual preprocessing

The most advanced transform methods are created for texts in natural languages. In this chapter we concern on the English language as it is the most popular language in the computer science and most of texts in natural languages are written in English. Of course, ideas presented here are universal and can be used with, for example, any language that uses Latin alphabet. The exceptions are the *q*-gram replacement and the word-based preprocessing, which usually use a fixed dictionary (for a given language) that must be created in advance.

Textual preprocessing ideas have been described by, among others, Kruse and Mukherjee [KM98], Teahan [Te98], Grabowski [Gr99], Franceschini et al. [FK$^+$00], Sun et al. [SM$^+$03], and Abel and Teahan [AT05].

To better understand textual preprocessing methods, a description of a structure of texts in a natural language in needed. Texts in natural languages can be divided into sentences finished by a period, a question mark, or an exclamation mark. Each sentence consists of words that are separated from the neighboring words by space and/or punctuation marks. It is not true for languages that do not use separators, for example, Chinese and Korean. Finally, each word consists of letters over some alphabet.

Most of schemes presented in this section are based on the idea of improving the modeling stage in predictive and BWT-based compression techniques. These schemes do not work (except the End-of-Line coding and the *q*-gram replacement) with dictionary-based compression techniques as they increase a size of data while dictionary-based compression techniques do not take advantage of an improved prediction.

4.7.1 Recognizing textual files

Using a text preprocessing algorithm on non-textual data leads to worse compression effectiveness, so there is a need to determine if an input file is suitable for the textual preprocessing. The main problem is that texts do not have a specific structure like, for example, XML data, which consists of element tags and usually have a header. In their work [AT05], Abel and Teahan have presented simple text recognition scheme. They assumed that a file categorized as textual has to fulfill two requirements:

a) The percentage frequency share of letters (‘A’,..., ‘Z’, ‘a’,..., ‘z’), digits (‘0’,..., ‘9’) and the space symbol compared to all symbols should be greater than 66%,
b) The percentage frequency share of the space symbol compared to letters (‘A’,..., ‘Z’, ‘a’,..., ‘z’) and digits (‘0’,...,’9’) should be greater than 10%.

The first condition checks if letters, digits and spaces dominate in an input data as they do in most of textual files. The second condition takes into account a property that the space symbol is a delimiter of words and separates neighboring words.

This approach works very well, but it assumes that textual files are based on Latin letters, so languages not based on Latin letters, like Russian and Chinese, have not been taken into account. Moreover, is worth to mention that, for example, source codes in programming languages will be categorized as textual files.

4.7.2 Capital conversion

The capital conversion (CC) is a well-known preprocessing technique [FM96, Gr99], which increases context dependencies and similarities between words. It is based on the observation that words like, e.g., ‘compression’ and ‘Compression’, are intrinsically very similar, but universal compression methods do not take fully advantage of this dependencies. The CC idea is to replace a capital letter at the beginning of a word with its corresponding lowercase equivalent and denote the change with a flag. Additionally, it is worth to use another flag to mark a conversion of an all-letter capitalized word to its lower case form. The replacement is omitted for words other than initial letter capitalized words or all-letter capitalized words.

The reason why this idea works is simple and will be showed on the first flag variant. The position of a word starting with a capital letter, if additionally the word appears elsewhere in lowercase, is in most cases the beginning of a sentence. The first letter may vary but after the conversion we have a flag followed with a lowercase letter. The flag can be predicted rather easily as it usually is the first non-whitespace symbol after a period (or an exclamation mark, etc.). The following character, that is the first—now lowercase—letter of a word, remains, of course, hard to predict, but the actual idea of the conversion is to gather more context occurrences for the letters that appear later in the converted word. Sticking to the example above, we can say that having, e.g., ten occurrences of the context ‘compr’ is generally more beneficial than eight occurrences of ‘compr’ and two occurrences of ‘Compr’, as contexts with very similar predictions should be merged.

The described CC technique gives even better results if a space symbol is added right after the flag [Gr99], that is, between the flag and the encoded word. It helps to find a longer context with predictive compression algorithms as well as it helps to BWT-based algorithms.

Abel and Teahan [AT05] have reported a slight gain obtained from yet another idea: refraining from the conversion of such words that occur in the text only with the first letter capitalized. The benefit of this idea for the compression is rather obvious but it requires an additional pass over the text.

4.7.3 Space stuffing

The space stuffing is a very simple idea presented by Grabowski [Gr99]. It utilizes a property that most of textual files consist of lines, finished with End-Of-Line symbols (“Linefeed” (LF), “Carriage Return” (CR), or CR+LF), which are line separators, placed in exchange of space symbols. The End-Of-Line (EOL) symbols are usually followed by a word (at the beginning of the next line). In textual files, most words are preceded by the space symbol and EOL symbols spoil the prediction of words, especially if the same word is sometimes preceded by the space symbol and sometimes by an EOL symbol. The space stuffing is designed to overcome this problem. It is based on an idea that a space symbol should be placed after EOL symbol, that is, at the beginning of each line.

The space stuffing expands data, but improves the prediction by providing longer contexts for words, which were preceded by EOL symbols. After the space stuffing transform, EOL symbols are always followed by space symbols, which are very easy to predict. Moreover, the contexts for words are longer as more words are preceded by the space symbol. The same idea was used in the capital conversion method, where the space symbol was added after a flag. This idea can be improved by refraining from adding spaces after EOLs if the next symbol is neither a letter nor a space.

The End-of-Line coding is a bit more complicated technique than space stuffing, which uses the same property (a division of text into lines) of textual files. It is explained in the next subsection.

4.7.4 End-of-Line coding

The End-of-Line (EOL) coding is a preprocessing technique (invented by Malcolm Taylor) based on the observation that End-of-Line symbols are "artificial" in the text and thus spoiling the prediction of the successive symbols. The EOL coding idea is to replace EOL symbols with spaces and to encode information enabling the reverse operation in a separate stream (an EOL stream).

Textual files usually contain only one kind of End-of-Line symbols ("Linefeed" (LF), "Carriage Return" (CR), or CR+LF). If we firstly encode the kind of End-of-Line symbols used in a given file, then we can treat all variations of EOL symbols in the same manner, so two symbols CR and LF can be treated as one EOL symbol.

There are many variations of EOL coding, which have been incorporated in existing compressors. They differ a way in which the EOL stream is encoded, what is involved with a size of encoded data. The simplest idea is to encode a length of line or, in other words, a distance between successive EOL symbols. The PPMN compressor [Sm02b] uses a bit more complicated idea. It calculates the distance between an average line length and a length of the current line. Another idea is to encode the length of a line as the number of spaces in the line, or to encode the numbers of spaces between the current EOL position and the average line length.

We believe that one of the best variations of the EOL coding is used in DC archiver [Bi00] by Edgar Binder. In this scheme, EOL symbols are converted to spaces, and for each space in the text DC writes a binary flag to distinguish an original EOL from a space. The binary flags are encoded with the arithmetic coder, on the basis of an order-1 artificial context, similar to the SEE technique used in PPMZ. Each order-1 artificial context is created as concatenation of the preceding symbol, the following symbol, and the distance between the current position and the last EOL symbol. All components are linearly quantized to a few bits. This form of encoding joins statistics of occurrences of EOL symbols for similar PPM contexts and improves the probability estimation of EOL symbols. This probability is used during encoding of binary flags.

The EOL coding technique used in DC archiver can be further improved. The improvement follows Reference [AT05] that only those EOL symbols that are surrounded by lower-case letters should be converted to spaces. In our case, it means that some spaces and EOL symbols are not encoded, with some benefit for the overall compression effectiveness.

4.7.5 Punctuation marks modeling

The punctuation marks modeling idea was presented by Grabowski [Gr99] and is an extension of the space stuffing, presented earlier in this section. It is based on the observation that there is a widely respected convention not to precede punctuation marks with the space symbol in natural language texts. Thus, punctuation marks usually appear just after a word, without a separating space. It implies that if a given word is sometimes followed, for example, by a comma, and sometimes by a space, then predicting of the first character after this word is not so easy. Inserting a single space in front of a punctuation mark facilitates the prediction. Of course, the punctuation mark itself has still to be encoded, but its context is now more precise, and in the overall the loss from expanding the text is more than compensated. The places with the space symbol after a word and before a punctuation mark in the original text must be treated in a special way to distinguish them from places where the space symbol was inserted. For example, we can assume that if there is already a space before a punctuation mark, then these two symbols will be converted to the punctuation mark only, that is, the space symbol will be removed.

The punctuation marks modeling idea has already been used in a few compressors, for example, PPMN [Sm02b] and Durilca [Sh04]. Durilca inserts a space symbol after the words followed by symbols: ‘'’, ‘"’, ‘:’, ‘;’, ‘,’, ‘.’, ‘?’, ‘!’, ‘]’, ‘)’, ‘}’, ‘>’.

As we mentioned earlier, punctuation marks modeling gives gain only if there are at least a few occurrences of the word, some followed by space symbols and some followed and punctuation marks. In another case, this technique can deteriorate the compression effectiveness.

4.7.6 Alphabet reordering

Predictive and dictionary compression techniques are largely based on the pattern matching, which is entirely independent of the encoding used for the source alphabet. An order of the symbols, however, makes a difference in the BWT-based compression, where it influences on the order of sorting in the Burrows–Wheeler Transform. In turn, the order of sorting determines which contexts are close to each other in the output of BWT stage, what influences on the overall compression effectiveness [CT98].

It turns out that the alphabetical order is not the best choice for BWT-based compressors; therefore Chapin and Tate [CT98] proposed an alphabet reordering technique. The alphabet reordering is a simple permutation (an one-to-one mapping) of the source alphabet. It can be implemented as a transform, which is independent from a compressor. The reverse transform is performed after the decompression and requires knowledge of a permutation used to the transform. If we use a character-based (ASCII) alphabet, then the permutation can be recorded using at the most 256 bytes. Furthermore, the alphabet reordering using a fixed permutation adds almost nothing to the compression and the decompression time.

Chapin and Tate have presented several heuristics and the computed orderings, but the heuristics achieve better results on textual files. One of the best results was achieved using a heuristic that groups vowels, similar consonants, and punctuation marks. For the upper case letters, the order was ‘AEIOUBCDGFHRLSMNPQJKTWVXYZ’ and was analogous for the lower case letters.

Abel and Teahan [AT05] have tried many hand-permutated orderings and obtained a heuristic, which groups the vowels “AOUI” in the middle of the consonants and the 'E' at the end of the consonants. Moreover, other characters like digits and punctuation symbols were also reordered. For the upper case letters, the order was ‘SNLMGQZBPCFMRHAOUIYXVDKTJE’ and was analogous for the lower case letters. Abel and Teahan have reported the gain of over 1% using the new alphabet order, what is more than double as much as that reported by Chapin and Tate.

4.7.7 Q-gram replacement

The 8-bit ASCII alphabet contains less than 128 textual characters. Particularly, symbols exceeding value of 127 are not used in English texts. If we assign two characters for flags needed for capital conversion technique, there are still 126 available characters. The digram replacement [BW$^+$89] is a simple technique that utilizes this property. It maintains a dictionary of frequently used pairs of characters (digrams), which are replaced with tokens (unused characters). Digram replacement can be static, if the dictionary of digrams is fixed and what makes this attempt language dependent. The semi-static digram replacement makes an additional pass over input data and determines the frequently used digrams. This scheme may be easily improved if we take account of situations such as the ‘he’ being used infrequently because the ‘h’ is usually encoded as part of the preceding ‘th’. It is worth to mention that the semi-static digram replacement requires a transmission of the dictionary to make reverse transform possible. The digram replacement is very simple, fast, and quite effective. Teahan [Te98] noted that extending the alphabet only with a single unique symbol denoting the ‘th’ digram, improves PPM compression by up to 1%.

The reason why digram replacement improves the effectiveness of universal data compression algorithm is rather obvious as digram replacement shortens the length of input data. The digram replacement can be improved by generalizing it to substituting frequent sequences of q consecutive characters (q-grams) with single symbols, that is, to the q-gram replacement [Te98]. The sequence length q usually does not exceed 4. The q-gram replacement represents an improvement on the digram coding, but the problem with the static q-gram replacement is that the choice of phrases for the dictionary is critical and depends on the nature of input text. On the other hand, the semi-static q-gram replacement is not so simple as the semi-static digram replacement. The semi-static q-gram replacement schemes usually begin with the dictionary containing all the characters in the input alphabet and then add common digrams, trigrams, and so on, until the available tokens are finished [BW+89].

The compressors like DC [Bi00] or Durilca [Sh04] use the static q-gram replacement, that is, with fixed dictionary. For example, DC uses the following 9 quadgrams: ‘that’, ‘said’, ‘with’, ‘have’, ‘this’, ‘from’, ‘whic’, ‘were’, ‘tion’. It uses the following 26 trigrams: ‘all’, ‘and’, ‘any’, ‘are’, ‘but’, ‘dow’, ‘for’, ‘had’, ‘hav’, ‘her’, ‘him’, ‘his’, ‘man’, ‘mor’, ‘not’, ‘now’, ‘one’, ‘out’, ‘she’, ‘the’, ‘was’, ‘wer’, ‘whi’, ‘whe’, ‘wit’, ‘you’. And finally, it uses the following 88 digrams: ‘ac’, ‘ad’, ‘ai’, ‘al’, ‘am’, ‘an’, ‘ar’, ‘as’, ‘at’, ‘ea’, ‘ec’, ‘ed’, ‘ee’, ‘el’, ‘en’, ‘er’, ‘es’, ‘et’, ‘id’, ‘ie’, ‘ig’, ‘il’, ‘in’, ‘io’, ‘is’, ‘it’, ‘of’, ‘ol’, ‘on’, ‘oo’, ‘or’, ‘os’, ‘ou’, ‘ow’, ‘ul’, ‘un’, ‘ur’, ‘us’, ‘ba’, ‘be’, ‘ca’, ‘ce’, ‘co’, ‘ch’, ‘de’, ‘di’, ‘ge’, ‘gh’, ‘ha’, ‘he’, ‘hi’, ‘ho’, ‘ra’, ‘re’, ‘ri’, ‘ro’, ‘rs’, ‘la’, ‘le’, ‘li’, ‘lo’, ‘ld’, ‘ll’, ‘ly’, ‘se’, ‘si’, ‘so’, ‘sh’, ‘ss’, ‘st’, ‘ma’, ‘me’, ‘mi’, ‘ne’, ‘nc’, ‘nd’, ‘ng’, ‘nt’, ‘pa’, ‘pe’, ‘ta’, ‘te’, ‘ti’, ‘to’, ‘th’, ‘tr’, ‘wa’, ‘ve’. The same q-grams were found experimentally by Grabowski [Gr99].

4.8 Word-based textual preprocessing

The natural languages, for example English, contain some inherent redundancy, because not every combination of letters represents a proper word. The compression algorithms do not fully exploit this redundancy. The word-based preprocessing [KM98] is a preprocessing technique for texts in natural language. This technique is based on the notion of replacing whole words with shorter codes. It gives the most significant improvement in the compression effectiveness among textual preprocessing methods.

The dictionary of words is usually fixed (for a given language) and given in advance. It is created by counting occurrences of words in a training set of textual files. The static word-based preprocessing is faster than the semi-static, as it does not require an additional pass over the text. Moreover, the dictionary of words does not have to be transmitted with output data. It means that even replacing words that occur only once is profitable. On the other hand, the static word-based preprocessing requires a fixed dictionary, which is shared by a compressor and a decompressor. Moreover, the static word-based preprocessing is limited to languages, for which fixed dictionaries are available. In overall, however, fixed dictionaries are commonly used as they give a better compression effectiveness and most of textual files are English.

4.8.1 Semi-static word replacement

In a recent work, Abel and Teahan [AT05] have presented their semi-static word replacement scheme, in which the dictionary is built adaptively and transmitted together with the output data. They suggest that this method is language-independent, but the actual algorithm they present is limited to the Latin alphabet.

The basic idea of the semi-static word replacement is to replace frequently used words by tokens (unused characters), which are indexes into a word dictionary. At the beginning, the semi-static word replacement determines available tokens. In the scheme of Abel and Teahan, four tokens are reserved: two for the most frequent trigrams and two for the most frequent digrams. The rest of available tokens are used for the word replacement. In the second step, which can be performed simultaneously with the determination of available tokens, the frequencies of all words with length of 2 or more in the input file are calculated. In the third step, all words W are omitted for the replacement, if the word S, which is a prefix of the word W, occurs inside the text with a frequency of at least 25% of W. In this case, S is used instead of W for the replacement. Moreover, for each word a weight V is evaluated, using the equation $V = (F-1)\cdot(L-1)$, where F is the frequency of the word W and L is the length of W. In the next step, words are sorted descending by their weights and available tokens are assigned to words with the highest weights, if they have a weight of at least 16. The words with assigned tokens form the dictionary.

In the final step, the words are replaced by tokens. If a word occurs for the first time, then it is replaced by a token, followed by the word in a plain text. Thanks to that, the decoder is able to rebuild the dictionary from the transmitted data. The next occurrences are replaced only by the token.

The method presented by Abel and Teahan can be easily improved. Following the Reference [BW+89], we can use a two-level system, which increases the size of an alphabet of available tokens. For example, a half of tokens can be used as the first byte of new two-byte tokens, so there will be $257\cdot H$ tokens in overall—H one-byte tokens and $256\cdot H$ two-byte tokens—where H is a half of originally available tokens. The first byte distinctly distinguishes between one-byte (original) and two-byte (from extended alphabet) tokens.

4.8.2 Star-encoding

Star-encoding [KM98] is a specialized preprocessing algorithm, which exploits the redundancy in English texts by using a fixed dictionary of English words. It transforms the input text by matching words in the input text with words in the dictionary and replacing such words with a pointer into the dictionary. The pointer is represented by the codeword, which has the same length as the original word, but is easier to compress using universal compression algorithms. The one exception with the codeword length are words transformed to lowercase using the capital conversion method, which include an additional flag and their codewords are longer.

In the Star-encoding method, the dictionary consists of words, which are sequences of at least one symbol over the alphabet [a–z]. The dictionary is obtained from a large training set of English language texts. The dictionary is divided into subdictionaries according to word lengths. The words in subdictionaries are sorted by their frequency in the English texts used for training. The dictionary does not have to contain all words, which appear in the input text. If a word from the input text is not located in the dictionary, then it is copied verbatim. There is also no need to use capital letters in the dictionary, as the Star-encoding scheme makes use of the capital conversion technique; there are two one-byte tokens (reserved characters) in the output alphabet to indicate that either a given word starts with a capital letter while the following letters are all lowercase (token t_{cl}), or a given word consists of capitals only (token t_{co}). Those tokens are inserted after the word. Next used token is t_{or}, which is used to build codewords. Finally, there is yet another token—t_{esc}—used for encoding occurrences of tokens t_{cl}, t_{co}, t_{or} and t_{esc}, that is, to handle collisions.

Star-encoding replaces words in the input text by codewords that mostly consist of repetitions of the token t_{or} (for which the authors chosen the '*' character). The codewords are generated using only the token t_{or} and characters [A–Z, a–z] and the following mapping. In each subdictionary, the first 27 words have assigned codewords: '***…*', 'A**…*', 'B**…*', …, 'Z**…*'. The next 26 words are encoded as 'a**…*', 'b**…*', …, 'z**…*'. The words from the 54th have assigned codewords: '*A*…*', '*B*…*', and so on. Of course, the length of a codeword depends on the length of an input word. Moreover, it is obvious that codewords will be unique and unambiguously decodable if a codeword will contain at least one '*' character.

In Star-encoding the '*' character is the dominant character, which often occupies more than a half of space in the preprocessed file. That is the main reason, why this transform improves the effectiveness of universal compression algorithms, but the improvement is insignificant.

4.8.3 LIPT

Franceschini et al. [FK+00] have extended the Star-encoding method by using different schemes for mapping the codewords, which are pointers into the dictionary. They have introduced the Length-Preserving Transform (LPT), the Reverse Length-Preserving Transform (RLPT) and the Shortened-Context Length-Preserving Transform (SCLPT).

Awan et al. [AZ+01] have presented a further improvement of the Star-encoding scheme, called the Length Index Preserving Transform (LIPT). Comparing to Star-encoding, only mapping of codewords has been changed. The codewords usually have a different length than original words. The token t_{or} (for which the authors chosen the '*' character) is used only at the beginning of each codeword. It distinguishes original words and codewords. The length of a word (equal to the number of subdictionary) is encoded right after the token t_{or}. After the length an index into the subdictionary is encoded. The index is encoded as a number representation of base 52, which cycles through [a–z, A–Z]. The lengths and the indexes use the same output alphabet [a–z, A–Z], which has assigned values [1–26, 27–52].

Resuming, each codeword consists of three parts: a single symbol '*', the length of the original word ([a–z, A–Z]), and the index into the subdictionary. The first index, which corresponds with the most frequent word, is represented as the empty string. For each subdictionary, the next 26 words have assigned indexes: 'a', 'b', …, 'z'. The following 26 indexes are encoded as 'A', 'B', …, 'Z'. The words from the 54th to the 2757th have assigned indexes of the length 2: 'aa', 'ab', …, 'ZY', 'ZZ'. The words from the 2758th have 3-byte indexes: 'aaa', 'aab', ..., and so on. For example, a word of the length 4 with the index 2 (counting from 0) have assigned the codeword '*db'. Of course, the codewords are unique and unambiguously decodable.

The main reason why this transform is better than the Star-encoding scheme is that in LIPT more frequent words have assigned shorter codewords and most of codewords are much shorter than original words. Moreover, the length of the original word placed after the single symbol ‘*’ results in an additional context, which is exploited by predictive and BWT-based compression algorithms. Awan et al. have reported that bzip2 with LIPT gives over 5% improvement in average over bzip2 without LIPT, and PPMD with LIPT gives over 4% improvement in average over PPMD without LIPT.

4.8.4 StarNT

The Star New Transform (StarNT [SM$^+$03]) is the most recent algorithm from the family of Star-transforms, which is based on LIPT. The main change, comparing to LIPT, is removing of the length of the original word from the codeword. The meaning of the token t_{or} (for which the authors chosen the ‘*’ character) has also been changed. In LIPT, the token t_{or} denotes the beginning of a codeword. In StarNT, it means that the following word has not been found in the dictionary. If the dictionary is suitable for an input file, then most of words do exist in the dictionary and they are replaced with codewords. In this case, StarNT improves the compression effectiveness over LIPT as it decreases the amount of t_{or} tokens.

The length of an original word was removed from the codeword, thus the dictionary is not more divided into subdictionaries according to word lengths. The order of words in the dictionary is important as it influences on a mapping of the words to codewords and, overall, on the compression effectiveness. StarNT stores the most frequently used words (312 words) at the beginning of the dictionary. The remaining words are stored in the dictionary according to their ascending lengths. Words of the same length are sorted according to their frequency of occurrence in a training set of textual files.

Resuming, each codeword in LIPT consists of three parts: a single symbol ‘*’, the length of the original word and an index into the subdictionary. We get rid of two parts and the codeword in StarNT consists only of an index into the dictionary. The mapping of indexes is the same as in LIPT, except the first index is not represented by an empty string, for an obvious reason. The length of the index is variable and spans from one to three characters. As only the range [a–z, A–Z] for indexes is used, there are, in total, $52 + 52 \cdot 52 + 52 \cdot 52 \cdot 52 = 143{,}364$ possible indexes. This is also the maximum amount of words in the dictionary, but the authors use the dictionary with only about 56,000 entries.

The last change does not influence on the compression effectiveness, but on the compression speed. StarNT uses the ternary search tree structure to searching the words in the dictionary, while LIPT uses a binary search tree. Experimental results have showed that the average compression time has been significantly improved comparing with LIPT.

Sun et al. have reported [SM$^+$03] that bzip2 with StarNT gives over 11% improvement in average over bzip2 without StarNT, and PPMD with StarNT gives over 10% improvement in average over PPMD without StarNT, what is more than twice as improvement obtained with LIPT.

5 Improved word-based textual preprocessing

The most famous statistical distribution in linguistics is Zipf law [Zi49]. George Kingsley Zipf have observed that the frequency of use of the i^{th}-most-frequently-used word in any natural language is inversely proportional to i^{ν}, with ν close to unity. For instance, the 100-th most frequent word is expected to occur roughly 10 times more frequently than the 1000-th most frequent word.

The natural languages, for example English, contain some inherent redundancy, because not every combination of letters represents a proper word. The compression algorithms do not fully exploit this redundancy. The word-based preprocessing [KM98] is a preprocessing technique for texts in natural languages, which is based on the notion of replacing whole words with shorter codes (according to Zipf's law), what improves the compression effectiveness.

The idea of replacing words with shorter codewords from a given static dictionary has at least two deficiencies. First, the dictionary must be quite large (at least tens of thousands words). Second, no "higher level" (e.g., related to grammar) correlations are implicitly taken into account. In spite of those drawbacks, such an approach to the text compression seems to be an attractive one, and has not, in our opinion, been given as much attention as it deserves. The benefits of dictionary based text compression schemes are the ease of producing the dictionary (assuming enough training texts in a given language), clarity of ideas, high processing speed, cooperation with a wide range of existing compressors, and—last not least—competitive compression ratios.

The static word-based preprocessing is faster than the semi-static, as it does not require an additional pass over the text. Moreover, the dictionary of words does not have to be transmitted with output data. It means that even replacing words that occur only once in the text is profitable. On the other hand, the static word-based preprocessing requires a fixed dictionary (or dictionaries), which is shared by compressor and decompressor. Moreover, the static word-based preprocessing is limited to languages, for which fixed dictionaries are available. In overall, however, the static word-based preprocessing is commonly used as it is simpler to implement and it gives a better compression effectiveness.

In this chapter we present our two new preprocessing algorithms: WRT and TWRT. Word Replacing Transform (WRT [SG+05]) is an English text preprocessing algorithm, based on ideas of algorithms from the family of Star-transforms [FK+00, SM+03]. WRT concerns on English language as it is the most popular language in a computer science and most of texts in natural languages are written in English. WRT ideas, however, are universal and can be used with, for example, any language that uses Latin alphabet. These ideas, however, will not work with languages that do not use separators, for example, Chinese and Korean. Moreover, WRT combines several well-known preprocessing techniques: the static word replacement, the capital the conversion, q-gram replacement, and the EOL coding.

Two-level Word Replacing Transform (TWRT [Sk05b]) is an expansion of WRT [SG+05]. Comparing to its predecessor, TWRT uses several dictionaries. It divides files on various kinds and chooses for each file combination of two best suitable dictionaries, what improves the compression effectiveness in a latter stage. 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.

At the end of this chapter, we present comparison of TWRT to our direct competitors—StarNT and PPM with built-in models. Moreover, we show results of our experiments with TWRT on main classes of lossless data compression algorithms (LZ, BWT-based, PPM, and PAQ).

5.1 WRT

Word Replacing Transform (WRT [SG+05]) is an English text preprocessing algorithm, based on ideas of algorithms from the family of Star-transforms [FK+00, SM+03]. 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 q-gram replacement, and the EOL coding. Moreover, it introduces a few new ideas that improve the effectiveness of universal compression algorithms, which are applied after our algorithm.

In this dissertation we divided, basing on a practical point of view, lossless data compression algorithms into four classes: LZ, BWT-based, PPM, and PAQ. Compressors from the LZ family differ significantly to BWT, PPM, and PAQ. Therefore, we have decided to create two versions of our algorithm, one for BWT-based and predictive compression schemes (WRT [SD04a]) and one for LZ-based compressors (WRT-LZ77 [SD04b]). In this section we often compare our preprocessor to the best currently word-based textual preprocessor—StarNT.

5.1.1 Dictionary

In the WRT scheme, the dictionary consists of words, which are sequences of at least one symbol over the alphabet [a–z]. The dictionary is obtained from a training set of more than 3 GB English texts taken from the Project Gutenberg library[1]. The words in the dictionary are sorted by their frequency in the English texts used for training. The dictionary does not have to contain all words, which appear in the input text. If a word from the input text is not located in the dictionary, then it is copied verbatim. There is also no need to use capital letters in the dictionary, as our scheme makes use of the capital conversion technique: there are two one-byte tokens (reserved characters) in the output alphabet to indicate that either a given word starts with a capital letter while the following letters are all lowercase (token t_{cl}) or a given word consists of capitals only (token t_{co}). There is yet another token t_{esc}, used for encoding occurrences of tokens t_{cl}, t_{co}, and t_{esc}, that is, to handle collisions.

5.1.2 Mode for non-textual data

A practical textual preprocessing algorithm should be protected from a significant loss on data, which do not fit to its model (in particular, non-textual files). StarNT [SM+03], for example, does not meet such a requirement as it plainly encodes special (reserved) symbols that have appeared in the input data (e.g., from the alphabet of codewords) with two bytes, the first of which is token t_{esc}. As a result, some compression loss can be seen on the non-textual files, for example, on the files *geo*, *obj1*, and *obj2* from the Calgary corpus. Our goal was not simply to preserve the compression effectiveness on non-textual files, but also, if possible, improve it, if the non-textual file contains some parts of text (words) inside.

[1] http://www.gutenberg.org

The WRT algorithm behaves identically to StarNT, but if the fraction of non-textual characters in already processed data is higher than 20%, the token t_{esc} is sent and it is assumed that the w successive characters (where w is constant) are also non-textual. In other words, t_{esc} is used to mark the beginning of a non-textual data chunk, rather than encoding a single non-textual character as StarNT does. In the non-textual regime, the read characters are copied verbatim to the output. The encoder, however, switches back to the textual regime if w preceding characters are all textual.

5.1.3 BWT/PPM/PAQ optimized model

In this subsection we present a version of WRT preprocessor optimized for BWT-based and predictive compression techniques. These techniques differ significantly to the LZ family, as they, for example, take into account contextual information. All these schemes are described in Chapter 2.

Capital conversion

The capital conversion idea is to convert the word with the first letter capitalized to its lowercase equivalent and denote the change with a flag. Another flag can be used to mark a conversion of a full uppercase word to its lowercase form. This technique was fully described in the previous chapter.

In StarNT, the flag is appended after the codeword. WRT inserts the flag in a front of the codeword and adds a space symbol after the flag, that is, between the flag and the codeword. This order improves the compression effectiveness of universal compression algorithms as it helps to find a longer context with predictive compression schemes as well as helps to BWT-based schemes.

Dictionary mapping

Ordinary textual files, at least English ones, consist solely of ASCII symbols not exceeding value of 127. The symbols with higher values are unused, and thus could be spent for the codewords. Thus, WRT reserves symbols with values from 128 through 255 to an alphabet of codewords. If a symbol exceeding value of 127 exists in the input file, then WRT outputs token t_{esc} and this symbol. This is a big difference comparing to the family of Star-transforms, where only 52 symbols ([a–z, A–Z]) are used. Moreover, in our algorithm the alphabets for codewords and for original words are separate. Consequently, token t_{or} is not used in WRT.

The order of words in the dictionary as well as a mapping the words to unique codewords are important for the compression effectiveness. As we wrote earlier, WRT reserves 128 symbols for an alphabet of codewords. The length of the codeword varies from one to four symbols. Our dictionary is sorted according to the frequency of words in the English texts used for training as more frequent words should be represented with shorter codes than less frequent words. The dictionary was reduced to 80,000 most frequent entries, and each word in the dictionary has assigned a corresponding codeword.

Star-transforms use the same alphabet ([a–z, A–Z]) for the successive symbols that codeword consists of. In WRT, the alphabet of codewords is divided into four separate parts ($101 + 9 + 9 + 9 = 128$ symbols), and successive symbols of a codeword use a separate alphabet. As the length of the codeword varies from one to four symbols, thus there are $101 + 101 \cdot 9 + 101 \cdot 9 \cdot 9 + 101 \cdot 9 \cdot 9 \cdot 9 = 82{,}820$ distinct codewords available, what is more than enough for our dictionary. The codeword bytes are emitted in the reverse order, this is, the range for the last codeword symbol has always 101 values. The separate alphabet for successive symbols of a codeword and a reverse order of the symbols create a special context. This context improves the compression effectiveness as it is exploited by BWT-based and predictive compression schemes.

Word order in the dictionary

In our dictionary, words are sorted with the relation to their frequency in a training corpus of English texts. We found it of benefit to first sort the dictionary according to the frequency, and then sort it into small groups according to the lexicographical order of suffixes, that is, sorting from right to left. Corresponding to the chosen parameters presented in the previous subsection, the first 10 (1 + 9) small groups have 101 items each, nine subsequent groups have 909 (101·9) items, and finally all the following groups have 8181 (101·9·9) items. Such a solution improves the contextual prediction in the latter phase.

Matching shorter words

Having separate alphabets for original words and codewords, it is easy to encode only a prefix of a word, if the prefix matches some word in the dictionary but the whole word does not. WRT finds the longest prefix of the original word, which matches some word in the dictionary. If the length of the found prefix is greater than four symbols, then WRT replaces the original word with the codeword of the found prefix. The remaining symbols are copied verbatim.

StarNT cannot encode prefixes of the original word as the output codeword and the suffix of the word would be glued and thus ambiguous for the decoder. For this modification, StarNT would need an additional token, for example t_{sep}, in order to separate the codeword and the suffix of the original word. Introducing such a token may not be compensated by the gain achieved from the substitution.

Q-gram replacement

The *q*-gram replacement is a simple technique that maintains a dictionary of frequently used sequences of *q* consecutive characters (*q*-grams), which are replaced with tokens (reserved characters). This technique was fully described in the previous chapter.

English text files consist solely of ASCII symbols not exceeding value of 127. Thus the symbols with higher values are usually used for the *q*-gram replacement. In WRT, however, these symbols are already used as the codeword alphabet. Nevertheless, a nice side-effect of our capital conversion variant is that we get rid from all the capital letters, and therefore those 26 characters ([A–Z]) can be used for encoding *q*-grams. WRT uses the static *q*-gram replacement with the following 2 trigrams: ‘ent’ and ‘ill’. It also uses the following 24 digrams: ‘my’, ‘go’, ‘ah’, ‘is’, ‘if’, ‘am’, ‘of’ , ‘he’, ‘be’, ‘by’, ‘in’, ‘on’, ‘to’, ‘as’, ‘at’, ‘or’, ‘do’, ‘up’, ‘an’, ‘we’, ‘me’, ‘no’, ‘it’, ‘so’.

EOL-coding

The End-of-Line (EOL) coding bases on the observation that End-of-Line symbols in the text spoil the prediction of the successive symbols as they are “artificial”. The EOL coding idea is to replace EOL symbols with spaces and to encode in a separate stream (an EOL stream) information enabling the reverse operation. This technique was fully described in the previous chapter.

WRT uses a variation of EOL coding presented by Edgar Binder in DC archiver [Bi00]. EOL symbols are converted to spaces, and for each space in the text, WRT writes a binary flag to distinguish an original EOL from a space. The flags are encoded with an arithmetic encoder, on the basis of an artificial context. The information taken into account to create the context involves the preceding symbol, the following symbol, and the distance between the current position and the last EOL symbol. This technique is improved according to the assumption that only EOL symbols that are surrounded by lower-case letters should be converted to spaces. The EOL stream is appended at the end of the output file.

Surrounding words with spaces

Most of words in natural language texts are surrounded by space symbols. But there are exceptions like a word followed by a comma, by a period, or a quoted word. The punctuation marks modeling, presented in the previous chapter, concerns on symbol following the word and inserts space symbol after the word. WRT converts all words to be surrounded (that is, preceded and followed) by the space symbol. This technique can be considered as a generalized version of the punctuation marks modeling.

WRT introduces two new tokens—t_{sb} and t_{sa}—which are used to modeling a surrounding of word. Let us assume that a word is surrounded by symbols different than the space and, of course, letters. WRT appends the token t_{sb} and the space symbol before the word. After the word, WRT appends the space symbol and the token t_{sa}, so the word ‘WORD’ is surrounded with spaces and looks like ‘t_{sb} WORD t_{sa}’. The decoding is very simple. If the decoder reads t_{sb}, then it omits the space symbol after this token, and if the decoder reads t_{sa}, then it omits the space symbol before this token.

The method of surrounding words with spaces, combining with predictive compression schemes, joins similar contexts and helps in a better prediction of the first symbol of the word as well as the next symbol just after the word. This technique, like the punctuation marks modeling, gives a gain only if there are at least a few occurrences of the word: some surrounded by space symbols and some preceded or followed by other symbols. In another case, this technique can deteriorate the compression effectiveness. It not possible to decide without an additional pass over an input file if surrounding words with spaces should be used. But the additional pass is not desired and we decided not to make the additional pass over an input file. Nevertheless, surrounding words with spaces works with most of large textual files, and WRT simply sets the minimum file size threshold to 1 MB for switching on this idea.

5.1.4 LZ optimized model

LZ-based compression schemes are currently widely used as they enable a high compression speed and a very high decompression speed. They differ significantly to predictive and BWT-based compression schemes. Basically, they parse the input data into a stream of matching sequences. The LZ matches are encoded by specifying their offsets and lengths. The strength of LZ is succinct encoding of long matching sequences, but in practice most matches are rather short, and the LZ encoding scheme cannot compete with BWT-based and predictive compression techniques, which—for example—take into account contextual information.

Those considerations lead to the (experimentally validated) conclusion that almost any idea to shorten the output of the preprocessor is also beneficial for the LZ-based compression. This was not the case with BWT-based and predictive compression techniques.

In this subsection we present WRT-LZ77—a version of WRT preprocessor optimized for LZ-based compression schemes. The WRT-LZ77 preprocessor shortens textual files to about 35–40% of its original length, that is, it is itself a textual compressor comparable with gzip. This gain is achieved by:

- Reducing the number of literals to encode;
- Decreasing the distances (in bytes) to the previous occurrence of the matching sequence;
- Decreasing the length (in bytes) of the matching sequence;
- Virtually increasing the sliding window, that is the past buffer from which matching sequences are found.

Capital conversion

WRT inserts a capital conversion flag in a front of the codeword and adds the space symbol after the flag, that is, between the flag and the codeword. Adding the space symbol between the capital conversion flag and the codeword does not bring profit in combination with the LZ-based compressor, as LZ-based algorithms do not take advantage of the created context.

In WRT-LZ77 we have also added a specific LZ capital conversion, which inverts—before the ordinary capital conversion—the case of the first letter after a period, a quotation or an exclamation mark. It results in fewer capital conversion flags. This variant, however, is limited to words with the first letter capitalized and it can deteriorate the compression effectiveness with full uppercase words, which open sentences. Of course, words starting with a capital and full uppercase words that appear somewhere in the middle of a sentence are converted to the lowercase and signaled with a proper flag.

Dictionary mapping

WRT reserves symbols with values from 128 through 255 to an alphabet of codewords. The length of the codeword varies from one to four symbols. The alphabet of codewords is divided into four separate parts (101 + 9 + 9 + 9 = 128 symbols). As the length of the codeword varies from one to four symbols, thus there are $101 + 101\cdot9 + 101\cdot9\cdot9 + 101\cdot9\cdot9\cdot9 =$ 82,820 distinct codewords available.

The most important change in WRT-LZ77 is the alphabet of codewords, which is extended to as much as 189 symbols (‘A’–‘Z’, 0–8, 14–31, 128–255 and others). The codewords have at most 3 bytes. The subalphabets for successive codeword bytes are no longer fully disjoint. In fact, the range for the first byte does not overlap with the range for other bytes, but the ranges for the second and third bytes are same. The property to distinguish between the first (or the last) byte of a codeword and the other bytes is required by a spaceless model, which is described in the next subsection. In WRT-LZ77 the alphabet of codewords is divided into only two separate parts (167 + 22 = 189 symbols), thus there are $167 + 167\cdot22 + 167\cdot22\cdot22 =$ 84,669 distinct codewords available.

Spaceless words

A very important addition in WRT-LZ77 is a spaceless model [JJ92, MN^{+}97], which eliminates most of spaces between words. It assumes that there is the space symbol before each word found in the dictionary, and there is no need to encode this character. Most of words fulfill this assumption, and the remaining words are treated in a special way. A special token t_{spc} is encoded before the codeword, if an original word is not preceded by the space symbol.

Still, without spaces between words, there must be a way to detect codeword boundaries. Otherwise, the decoding with spaceless words would be impossible. The tagged

Huffman and similar schemes for the compression and search on words distinguish between the first (or the last) byte of a codeword and the other bytes. Traditionally, this has been performed with sacrificing one bit per byte, but recently more efficient schemes [RT$^+$02, BF$^+$03] have been presented. Our scheme, in specifying disjoint ranges for the first and the latter codeword bytes, is analogous to Reference [BF$^+$03]. We have run an algorithm from Reference [BF$^+$03] for obtaining the optimal (*s*, *c*)-Dense Code for a given distribution, and it suggested the alphabet partition 135 + 54 symbols for our training corpus. We, however, use the partition 167 + 22 symbols, chosen experimentally. The difference to the experimentally found partition means probably that the distribution of words in the training and the test data do not fit perfectly.

Word order in the dictionary

WRT sorts the dictionary according to the frequency of words, and then it sorts the dictionary into small groups according to the lexicographical order of suffixes. The LZ compression scheme does not take into account contextual information, so an additional sorting of the dictionary by a lexicographical order of suffixes does not help LZ-based compressors. Therefore WRT-LZ77 purely sorts the dictionary according to the frequency of words in a training data.

Matching shorter words

WRT can encode only a prefix of a word, if the prefix matches some word in a dictionary but the whole word does not. If the length of the found prefix is greater than four symbols, then WRT replaces the original word with the codeword of the found prefix. Remaining symbols are copied verbatim. WRT-LZ77 replaces the original word with the codeword of the found prefix if the length of the found prefix is equal to or greater than just one symbol.

Q-gram replacement

The *q*-gram replacement is a simple technique that maintains a dictionary of frequently used sequences of *q* consecutive characters (*q*-grams), which are replaced with tokens (reserved characters). WRT uses 26 characters ([A–Z]) for encoding *q*-grams. In WRT-LZ77, however, these characters are already used for an alphabet of codewords. Finally, *q*-gram replacement is removed from WRT-LZ77, but *q*-grams, which are popular English words, are included into a dictionary.

EOL-coding

WRT-LZ77 uses the same variation of the EOL coding as WRT. Nevertheless, assumption that only EOL symbols that are surrounded by lower-case letters should be converted to spaces does not help LZ-based compressors. Therefore WRT-LZ77 converts all EOL symbols to spaces.

Surrounding words with spaces

The LZ-based compression schemes does not take into account contextual information, so an idea of surrounding words with spaces does not work with LZ-based compressors. Moreover, this technique lengthens the output of the preprocessor, so it is disabled in WRT-LZ77.

5.2 TWRT

Two-level Word Replacing Transform (TWRT [Sk05b]) is an expansion of WRT [SG$^+$05], the English text preprocessor. Comparing to its predecessor, TWRT divides files on various kinds, and chooses for each file two best suitable dictionaries (one from the first group of dictionaries and one from the second group), what improves the compression effectiveness in a latter stage. Moreover, TWRT automatically recognizes multilingual text files, but operates with almost the same speed as its predecessor—WRT. TWRT is designed to work with all languages (for example, English, German, Polish, Russian, and French) that use the space symbol as a separator.

In TWRT the biggest gain in the compression effectiveness is achieved with the static word replacement, but TWRT combines several other well-known preprocessing techniques: the capital conversion, the q-gram replacement, the EOL coding, the fixed-length record aligned data preprocessing and the DNA sequence preprocessing. The last two schemes were not implemented in WRT.

In this dissertation, we have divided, basing on a practical point of view, lossless data compression algorithms into four classes: LZ, BWT-based, PPM, and PAQ. WRT is available in two versions, one optimized for BWT-based and predictive compression schemes (WRT) and one optimized for LZ-based compressors (WRT-LZ77). TWRT is available as one computer program, but is has separate optimizations for the further LZ, BWT-based, PPM, or PAQ compression. For this purpose, TWRT adaptively partitions an alphabet of codewords and automatically selects the codewords.

5.2.1 Description

A source file in any programming language, e.g., C, C++, Java, can be divided into programming language commands and comments. WRT replaces only comments in the English language with shorter codewords. If we add the programming language commands to the dictionary, then the compression effectiveness on source files should be improved, but the compression effectiveness on the remaining files should be deteriorated. Based on this observation, we have developed the first level dictionaries (small dictionaries).

TWRT can use up to two dictionaries, which are dynamically selected before the actual preprocessing starts. The first level dictionaries (small dictionaries) are specific for some kind of data (e.g., programming languages, references). The second level dictionaries (large dictionaries) are specific for the natural language (e.g., English, Russian, French). If no appropriate first level dictionary is found, then the first level dictionary is not used. Selection of the second level dictionary is analogous. When TWRT has selected only the one (the first or the second level) dictionary, the preprocessor works like WRT. If TWRT has selected both, the first and the second level dictionary, then the dictionaries are automatically joined (the second level dictionary is appended after the first level dictionary). If the same word exists in the first and the second level dictionary, then the second appearance of word is ignored to minimize length of codewords. Only the names of dictionaries are written in the output file, so the decoder can use the same combination of the dictionaries.

The main problem for TWRT is a selection of the dictionaries before the preprocessing. The simplest idea is to use a multi-pass preprocessing to preprocess the input file step by step with all dictionaries, and finally to choose the smallest output file. Nevertheless, this idea is very slow. We propose to read only the first f (for example, 250) words from each of g dictionaries (small and large) and create one, joined dictionary (only in a computer's memory). The joined dictionary consists of g parts (from g dictionaries). If there is the same word in different dictionaries, then all occurrences of this word are skipped. When the joined dictionary is created, we have to preprocess the input file using only this dictionary. While preprocessing we should count occurrences of words for each of *g* parts of the joined dictionary. In the end, we have to find the large dictionary with the biggest count of words' occurrences and the small dictionary with the biggest count of words' occurrences. These dictionaries, with a big probability, will give the best results in the latter compression. We call this technique a *dictionaries detection*. Moreover, we propose, what is experimentally checked, that the small dictionary should not be used if the count of words for the small dictionary is lower than 20% of the count of words for the large dictionary.

WRT is designed to be a one-pass preprocessor, but the dictionaries detection mechanism in TWRT needs an additional preprocessing pass. To improve the preprocessing speed, we can preprocess only some initial part of the input file (for example, 50 kB), but this can cause that TWRT will sometimes select (especially on heterogeneous files) wrong dictionaries. A better idea is to read data from random part of the input file. TWRT uses yet another idea: it divides file into several (for example, 5) equal blocks and it reads only a few kB (for example, 10 kB) from each block. This technique is called a *faster dictionaries detection* as it speed-ups the preprocessing speed in a high degree.

Some languages (e.g., French, German, Polish) use the Latin alphabet with added national characters. These characters use ASCII values over 127, so they should not be used as TWRT codewords alphabet. Moreover, sometimes for a given language there are several encoding standards of national characters (e.g., cp850 and ISO 8859-1 for the German language). Thus to find the suitable dictionary it is necessary to use words in all possible encodings. Therefore, for each word from the joined dictionary, if the word includes one or more national characters, then TWRT adds this word to the joined dictionary in all possible encodings. Furthermore, the same idea is used with, for example, the Russian language that does not use the Latin alphabet at all. Finally, TWRT selects the large dictionary with the most probable encoding. This encoding is required by the actual preprocessing, when TWRT reads entire large dictionary.

The faster dictionaries detection stage gives additional opportunities. TWRT can decide for each file if the mode for non-textual data, the method of surrounding words with spaces, and the EOL coding should be used in the actual preprocessing stage. It also gives possibility to find record-based files and DNA sequences and to use, respectively, the fixed-length record aligned data preprocessing or the DNA sequence preprocessing.

5.2.2 The first and the second-level dictionaries

Dictionary file name	Short name	File size in bytes	Number of words	Description
wrt-short-c++.dic	C++	700	85	Most frequent C++ words
wrt-short-comp_sc.dic	CompSc	164,381	18,596	Computer science dictionary
wrt-short-lisp.dic	Lisp	629	74	Most frequent LISP & Pascal words
wrt-short-math.dic	Math	196,368	21,301	Words from Encyclopedia of Math
wrt-short-politics.dic	Politics	134,127	16,845	Based on CIA World Factbooks
wrt-short-ref.dic	Ref	68,940	8,546	References
wrt-de.dic	DE	463,918	45,599	German dictionary
wrt-eng.dic	ENG	1,023,195	107,680	English dictionary
wrt-fr.dic	FR	648,314	67,667	French dictionary
wrt-pl.dic	PL	1,479,847	163,843	Polish dictionary
wrt-ru.dic	RU	4,954,251	473,524	Russian dictionary

Table 5.1: Detailed information about the first and the second-level dictionaries

For each kind of data we have collected a set of files of this kind to create the appropriate dictionary. Each dictionary is created by counting occurrences of words in the corresponding set of files. For example, English dictionary is created using over 3 GB English text files from the Project Gutenberg library. Each dictionary is sorted with a frequency of words in a descending order. TWRT uses six small dictionaries (C++, CompSc, Lisp, Math, Politics, Ref) and five large dictionaries (DE, ENG, FR, PL, RU). A full description of dictionaries is available in Table 4.1. Additional dictionaries can be easily added, without modifying the preprocessor.

5.2.3 LZ optimized model

TWRT-LZ is based on WRT optimized for LZ (WRT-LZ77). The alphabet of codewords has as much as 189 symbols (‘A’–‘Z’, 0–8, 14–31, 128–255 and others). It is divided into only two separate parts p_1, p_2 (for example, 167 and 22 symbols). The parameter p_1 is automatically selected, and it has the maximal value that allow mapping of all dictionary words. The parameter p_2 is equal to $189–p_1$. The codewords have at most 3 bytes and codeword bytes are emitted in the reverse order. The range for the first byte does not overlap with the range for other bytes, but the ranges for the second and third bytes are same.

TWRT-LZ uses a specific LZ capital conversion, which inverts—before ordinary capital conversion—the case of the first letter after a period, a quotation or an exclamation mark. A capital conversion flag is inserted in the front of the codeword, without the space symbol after the flag.

A spaceless model [JJ92, MN$^+$97], which eliminates most spaces between words is used in TWRT-LZ. It assumes that there is the space symbol before each word found in a dictionary, and there is no need to encode this symbol. Most of words fulfill this assumption, and the remaining words are treated in a special way.

TWRT-LZ can encode only a prefix of a word, if the prefix matches some word in a dictionary but the whole word does not. If the length of the found prefix is equal to or greater than one symbol, then WRT replaces the original word with the codeword of the found prefix. The remaining symbols are copied verbatim.

Moreover, the *q*-gram replacement is not used in TWRT-LZ, but *q*-grams, which are popular English words, are included into a dictionary. TWRT-LZ uses the same variation of the EOL coding as WRT-LZ77; it converts all EOL symbols to spaces, not only EOL symbols that are surrounded by lower-case letters. The method of surrounding words with spaces is disabled in TWRT-LZ.

TWRT-LZ automatically detects record-based files and DNA sequences and uses, respectively, the fixed-length record aligned data preprocessing or the DNA sequence preprocessing. For textual files TWRT-LZ automatically decides if the mode for non-textual data and the EOL coding should be used in the actual preprocessing stage.

5.2.4 BWT optimized model

TWRT-BWT is very close to TWRT-LZ. There are, however, three exceptions. First, TWRT-BWT does not use a specific LZ capital conversion, which inverts—before ordinary capital conversion—the case of the first letter after a period, a quotation or an exclamation mark. Second, TWRT-BWT uses an EOL coding variation, where only EOL symbols that are surrounded by lower-case letters are converted to spaces. Third, TWRT-BWT has an alphabet of codewords with 189 symbols (‘A’–‘Z’, 0–8, 14–31, 128–255 and others), but the alphabet is divided into four separate parts ($p_1 + p_2 + p_2 + p_2$ symbols). The parameter p_2 is automatically selected, and it has the minimal value that allow mapping of all dictionary words. The parameter p_1 is equal to $189–3·p_2$. The codewords have at most 4 bytes and codeword bytes are emitted in the reverse order. The range for each codeword byte does not overlap with the range for other bytes of the codeword.

5.2.5 PPM optimized model

TWRT-PPM is based on WRT optimized for BWT/PPM/PAQ. The alphabet of codewords has 128 symbols (values from 128 through 255), and it is divided into four separate parts ($p_1 + p_2 + p_2 + p_2$ symbols). The parameter p_2 is automatically selected, and it has the minimal value that allow mapping of all dictionary words. The parameter p_1 is equal to $128–3·p_2$. The length of the codeword varies from one to four bytes, and codeword bytes are emitted in the reverse order. The range for each codeword byte does not overlap with the range for other bytes of the codeword.

During usage of the capital conversion technique, TWRT-PPM inserts the flag in the front of the codeword and adds the space symbol after the flag, that is, between the flag and the codeword.

TWRT-PPM can encode only a prefix of a word, if the prefix matches some word in a dictionary but the whole word does not. If the length of the found prefix is greater than four symbols, then WRT replaces the original word with the codeword of the found prefix. The remaining symbols are copied verbatim.

The capital letters ([A–Z]) are not used as we get rid from all the capital letters using the capital conversion technique. In TWRT-PPM we also do not require them for the alphabet of codewords. Thus TWRT-PPM uses these 26 characters for encoding q-grams.

TWRT-PPM uses the same variation of EOL coding as WRT optimized for BWT/PPM/PAQ. EOL symbols are converted to spaces, and for each space in the text, TWRT-PPM writes a binary flag, to distinguish an original EOL from a space. The flags are encoded with an arithmetic encoder, on the basis of an artificial context. Only EOL symbols that are surrounded by lower-case letters are converted to spaces. The EOL stream is appended at the end of the processed file.

TWRT-PPM uses the technique of surrounding words with spaces, which converts all words to be surrounded (that is, preceded and followed) by the space symbol. WRT simply sets the minimum file size threshold for switching on this idea to 1 MB. TWRT-PPM automatically decides (in the dictionaries detection stage) if the surrounding words with spaces should be used.

TWRT-PPM automatically detects record-based files and DNA sequences and uses, respectively, the fixed-length record aligned data preprocessing or the DNA sequence preprocessing. For textual files TWRT-PPM decides if the mode for non-textual data, the method of surrounding words with spaces, and the EOL coding should be used in the actual preprocessing stage.

5.2.6 PAQ optimized model

TWRT-PAQ is very close to TWRT-PPM. The alphabet of codewords has 128 symbols, and it is divided into four separate parts (64 + 32 + 16 + 16 symbols). The partition is fixed, and it is optimized for PAQ’s bit-level predictors.

There are three more changes comparing to TWRT-PPM. First, a capital conversion flag is inserted in the front of the codeword, but without the space symbol after the flag. Second, the q-gram replacement is not used at all. Third, the fixed-length record aligned data preprocessing and the DNA sequence preprocessing are not used, as PAQ includes the context model for data with fixed-length records and models, which help in compression of DNA sequences.

For textual files TWRT-PAQ automatically decides if the mode for non-textual data, the technique of surrounding words with spaces, and the EOL coding should be used in the actual preprocessing stage.

5.3 Results of experiments

In this section, we present comparison of TWRT to our direct competitors—StarNT, RKC (PPM with word-based preprocessing), and PPM with built-in models (using Durilca). We have planned to include word-based BWT in our experiments, but it is not publicly available [Mo05]. Moreover, we show results of our experiments with TWRT [Sk05a] on main classes of lossless data compression algorithms. We have selected most popular and currently top compressors from main classes of lossless data compression algorithms, that is, LZ (gzip [Ga93], 7-Zip [Pa05], UHARC [He04]), BWT (bzip2 [Se02], GRZipII [Gr04], UHBC [He03]), PPM (PPMVC [Sk03], RKC [Ta04], Durilca [Sh04]), PAQ (PAQAR [MR04a]). Our transform routines in TWRT can interfere with possible preprocessing techniques used by compressors, but the influence is small. The detailed information about switches used during the experiments for examined compression programs can be found in Appendix F.

Durilca is essentially PPMII with built-in models, selected for given data via a data detection routine. If the model fits to data (e.g., to English text), then PPM starts the compression with lots of useful knowledge. Only English and Russian built-in models are available. An expanded description of Durilca can be found in Chapter 2, subsection "PPM with built-in models".

RKC is a general-purpose compressor with built-in English, Russian and Polish dictionaries. It is based on PPMZ with added variation of the StarNT transform. Unfortunately for our experiments, the preprocessing stage and the compression stage cannot be separated in RKC, so we cannot use RKC just as a pure preprocessor.

PAQAR 4.0 is an improvement of PAQ6 (described in Chapter 2), which is a binary-based PPM. Nevertheless, PAQ uses several models (bit-level predictors) comparing to the single model used in the PPM algorithm.

TWRT is available as one computer program, but is has separate optimizations for the further LZ, BWT-based, PPM, or PAQ compression. The optimizations for the compression algorithm implemented in a compressor used after the preprocessing stage were selected in TWRT, what gives the best compression effectiveness.

All tests were carried out on AMD Sempron 2200+ (1500 MHz) machine equipped with 512 MB RAM, under Windows 98 SE operating system.

5.3.1 Experiments with the Calgary corpus

We have carried out our first experiments on a well-known set of files—the Calgary corpus [BC$^+$90], fully described in Appendix A.

Because all the files in the Calgary corpus are English or non-textual, TWRT uses only the English large dictionary. To be more specific, the following dictionaries was automatically selected for files: ***bib*** (Ref, ENG), ***book1*** (ENG), ***book2*** (CompSc, ENG), ***geo*** (none), ***news*** (ENG), ***obj1*** (CompSc, ENG), ***obj2*** (none), ***paper1*** (CompSc, ENG), ***paper2*** (ENG), ***pic*** (none), ***progc*** (C++, ENG), ***progl*** (Lisp, ENG), ***progp*** (Lisp, ENG), ***trans*** (Ref, ENG).

As can be seen in Figure 5.1, TWRT improves the compression effectiveness of all tested compressors. The improvement is significant and span from 5% (RKC) to almost 15% (gzip). Full results are available at Table 5.2 and Table 5.3.

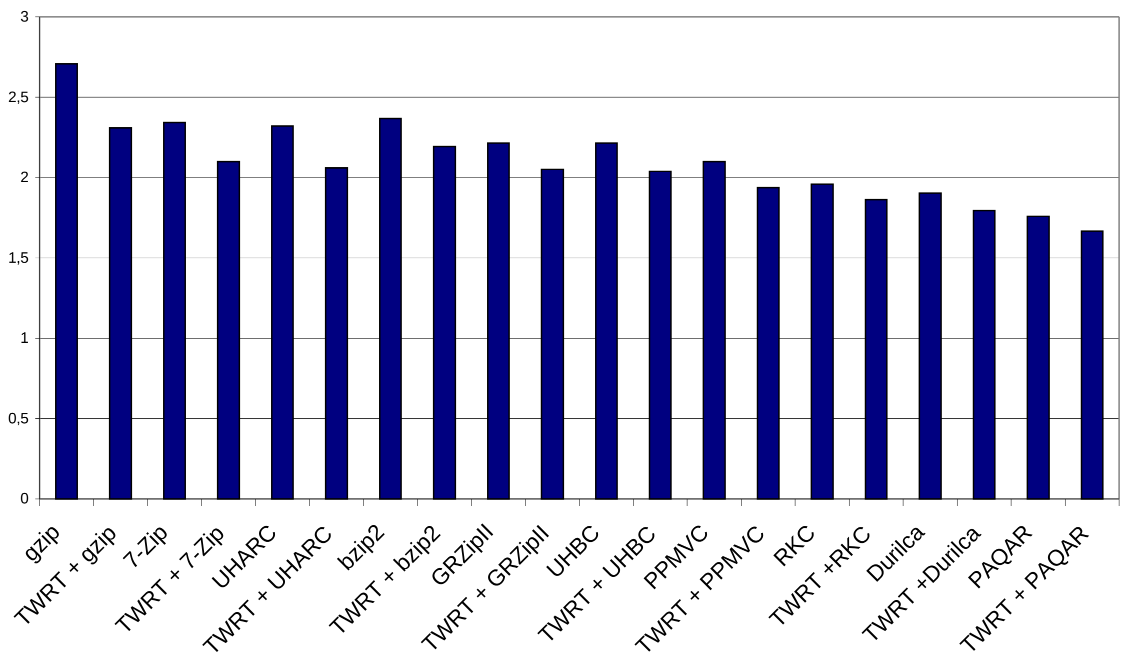

Figure 5.1: Average compression effectiveness (in bpc, lower is better) with TWRT and LZ, BWT, PPM, PAQ compressors on the Calgary corpus.

File	gzip	TWRT + gzip	7-Zip	TWRT + 7-Zip	UHARC	TWRT + UHARC	Bzip2	TWRT + bzip2	GRZipII	TWRT + GRZipII	UHBC	TWRT + UHBC
bib	2.521	2.175	2.210	1.880	2.191	1.830	1.975	1.721	1.873	1.606	1.866	1.593
book1	3.261	2.312	2.719	2.167	2.760	2.079	2.420	2.131	2.241	2.008	2.235	1.981
book2	2.707	2.122	2.227	1.908	2.298	1.850	2.062	1.872	1.924	1.774	1.922	1.759
geo	5.351	4.128	4.227	3.935	3.966	3.792	4.447	4.178	4.211	4.007	4.067	3.913
news	3.073	2.615	2.535	2.262	2.521	2.197	2.516	2.330	2.335	2.169	2.368	2.183
obj1	3.840	3.848	3.541	3.561	3.562	3.580	4.013	4.037	3.738	3.763	3.653	3.683
obj2	2.646	2.647	2.013	2.017	2.146	2.149	2.478	2.476	2.316	2.316	2.377	2.377
paper1	2.796	2.167	2.612	2.080	2.582	2.034	2.492	2.104	2.365	1.973	2.367	1.958
paper2	2.896	2.096	2.657	2.054	2.641	2.008	2.437	2.069	2.318	1.960	2.317	1.935
pic	0.880	0.692	0.686	0.612	0.494	0.500	0.776	0.529	0.744	0.488	0.703	0.454
progc	2.681	2.429	2.551	2.297	2.542	2.277	2.533	2.384	2.402	2.229	2.408	2.203
progl	1.817	1.674	1.687	1.546	1.681	1.515	1.740	1.637	1.620	1.501	1.647	1.503
progp	1.822	1.763	1.693	1.638	1.691	1.594	1.735	1.718	1.585	1.540	1.651	1.587
trans	1.621	1.662	1.445	1.447	1.434	1.431	1.528	1.521	1.352	1.371	1.425	1.405
average	2.708	2.309	2.343	2.100	2.322	2.060	2.368	2.193	2.216	2.050	2.215	2.038
ctime	0.60	2.15	5.00	4.39	7.31	6.43	2.20	2.97	2.64	3.86	5.05	5.93
dtime	0.16	1.26	0.55	1.64	1.54	2.85	0.93	1.92	2.53	3.45	4.18	5.32
CTA	3.028	3.586	3.335	3.767	3.166	3.635	3.572	3.802	3.663	3.830	3.401	3.570

Table 5.2: Results of experiments with TWRT and LZ, BWT-based compressors on the Calgary corpus. Results are given in bits per character (bpc). The compression and decompression times (ctime and dtime) are given in seconds.

File	PPMVC	TWRT + PPMVC	RKC	TWRT + RKC	StarNT + RKC	RKC with dictionary	Durilca	TWRT + Durilca	StarNT + Durilca	Durilca with built-in models	PAQAR	TWRT + PAQAR
bib	1.726	1.535	1.651	1.506	1.607	1.647	1.645	1.486	1.614	1.515	1.528	**1.363**
book1	2.188	1.947	2.132	1.951	2.087	2.048	2.119	1.901	2.062	**1.840**	2.031	1.863
book2	1.830	1.683	1.764	1.653	1.736	1.723	1.722	1.587	1.684	1.618	1.604	**1.525**
geo	4.336	3.944	3.676	3.885	3.676*	3.676	3.747	3.799	3.747*	3.747	**3.467**	3.469
news	2.186	2.033	2.112	1.987	2.065	2.084	2.058	1.906	2.010	1.917	1.925	**1.811**
obj1	3.537	3.511	3.234	3.209	3.234*	3.234	3.083	3.075	3.083*	3.083	2.841	**2.817**
obj2	2.165	2.166	1.803	1.803	1.803*	1.803	1.758	1.758	1.758*	1.758	**1.477**	1.478
paper1	2.192	1.891	2.144	1.876	2.022	2.087	2.108	1.809	1.969	1.836	1.973	**1.695**
paper2	2.176	1.867	2.132	1.870	2.002	2.062	2.105	1.813	1.958	1.749	1.995	**1.746**
pic	0.743	0.475	0.681	0.461	0.681*	0.681	0.503	0.448	0.503*	0.503	**0.372**	**0.372***
progc	2.203	2.069	2.152	2.039	2.073	2.121	2.079	1.945	1.998	1.934	1.945	**1.824**
progl	1.436	1.359	1.388	1.316	1.338	1.379	1.321	1.240	1.265	1.300	1.223	**1.159**
progp	1.442	1.406	1.395	1.360	1.369	1.462	1.298	1.259	1.272	1.294	1.200	**1.179**
trans	1.219	1.215	1.149	1.154	1.149*	1.157	1.105	1.099	1.105*	**1.014**	1.038	1.029
average	2.098	1.936	1.958	1.862	1.917	1.940	1.904	1.795	1.859	1.793	1.758	**1.666**
ctime	3.73	4.24	41.52	50.64	48.45	44.05	19.99	25.92	24.56	22.13	913.02	724.52
dtime	3.19	3.78	46.47	55.14	54.17	48.83	19.61	24.98	24.05	21.26	913.01	723.42
CTA	3.680	3.935	1.468	1.320	1.327	1.424	2.371	2.151	2.154	2.350	0.103	0.129

Table 5.3: Results of experiments with TWRT and PPM, PAQ compressors on the Calgary corpus. Results are given in bits per character (bpc). The compression and decompression times (ctime and dtime) are given in seconds. Files marked with an asterisk were not preprocessed.

Table 5.3 also shows that TWRT with RKC and Durilca achieves about 3% better compression effectiveness than StarNT in a connection with RKC and Durilca. Moreover, TWRT with RKC achieves over 4% better compression effectiveness than RKC with the built-in word-based preprocessing. TWRT with Durilca achieves almost the same results as Durilca with built-in models. Nevertheless, differences between single files spread up to 8%.

As we expected, TWRT with PAQAR gives the best compression effectiveness. It has the best compression on ten from fourteen files, and it slightly looses on two files to single PAQAR by adding a few bytes headers. On the remaining two files it is very close to Durilca with built-in models. The main problem with PAQAR is that it is very slow (3 kB/s) and high memory consuming.

On experiments with StarNT, the files *geo*, *obj1*, *obj2*, *pic*, *trans* were not preprocessed as StarNT does not operate with non-textual files. Also on experiments with WRT and PAQAR the file *pic* was not preprocessed as PAQAR contains a special model for this file that depends on the size of this file.

A coefficient of transmission acceleration (defined in Chapter 2) for the transmission speed 128 kbit/s for the Calgary corpus and selected compressors is presented in Figure 5.2. TWRT improves the transmission acceleration on all tested compressors except RKC and Durilca. We believe that lower transmission acceleration for RKC and Durilca is caused by the fact that the Calgary corpus files are small. Before preprocessing TWRT performs some actions (especially, reading of the dictionary), which take some time. That is why, TWRT is relatively slower on small files than on larger files, where preparation time is proportionally lower. This thesis is confirmed in the next experiments, which were performed on larger files. The highest coefficient of transmission acceleration is obtained with PPMVC, GRZipII, bzip2, and 7-Zip.

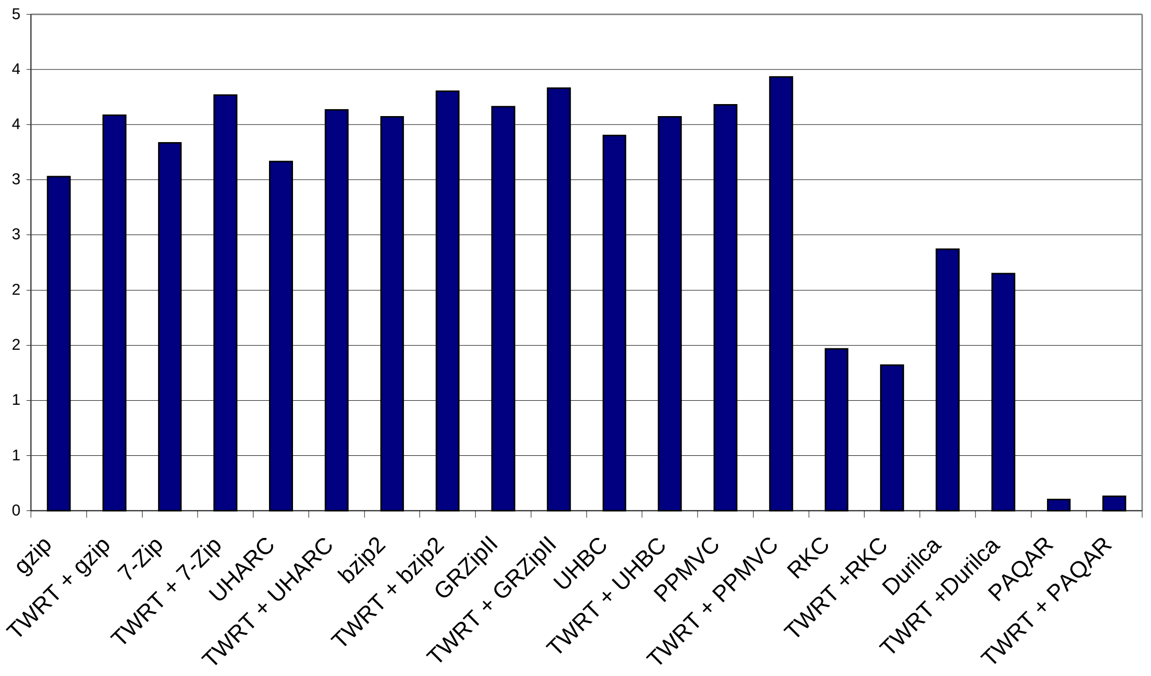

Figure 5.2: Coefficient of transmission acceleration (higher is better) with TWRT and LZ, BWT, PPM, PAQ compressors on the Calgary corpus.

5.3.2 Experiments with the Canterbury corpus and the large Canterbury corpus

The next experiments were carried out on a combination of the Canterbury corpus and the large Canterbury corpus, which are fully described in Appendix B.

Again, all the files in the Canterbury and the large Canterbury corpus are English or non-textual, and TWRT uses only the English large dictionary. To be more specific, the following dictionaries was automatically selected for files: *alice29.txt* (ENG), *asyoulik.txt* (ENG), *bible.txt* (ENG), *cp.html* (Math, ENG), *E.coli* (none), *fields.c* (Math, ENG), *grammar.lsp* (ENG), *kennedy.xls* (none), *lcet10.txt* (Ref, ENG), *plrabn12.txt* (ENG), *ptt5* (none), *sum* (C++, ENG), *world192.txt* (Politics, ENG), *xargs.1* (Math, ENG).

As can be seen in Figure 5.3, TWRT again improves the compression effectiveness of all tested compressors. The improvement is significant and span from almost 7% (Durilca) to almost 18% (gzip). Full results are available at Table 5.4 and Table 5.5. RKC results are not included as RKC crashed during the experiments.

Table 5.5 also shows that TWRT with Durilca achieves more than 2% better compression effectiveness than StarNT in a connection with Durilca. TWRT with Durilca is about 2% worse than Durilca with built-in models.

As we expected, TWRT with PAQAR again gives the best compression effectiveness. It has the best compression on ten from fourteen files, and it slightly looses on one file to single PAQAR. Durilca with built-in models is better than TWRT with PAQAR on the remaining three files.

On experiments with StarNT, the files *E.coli*, *kennedy.xls*, *ptt*, *sum* were not preprocessed as StarNT does not operate with non-textual files.

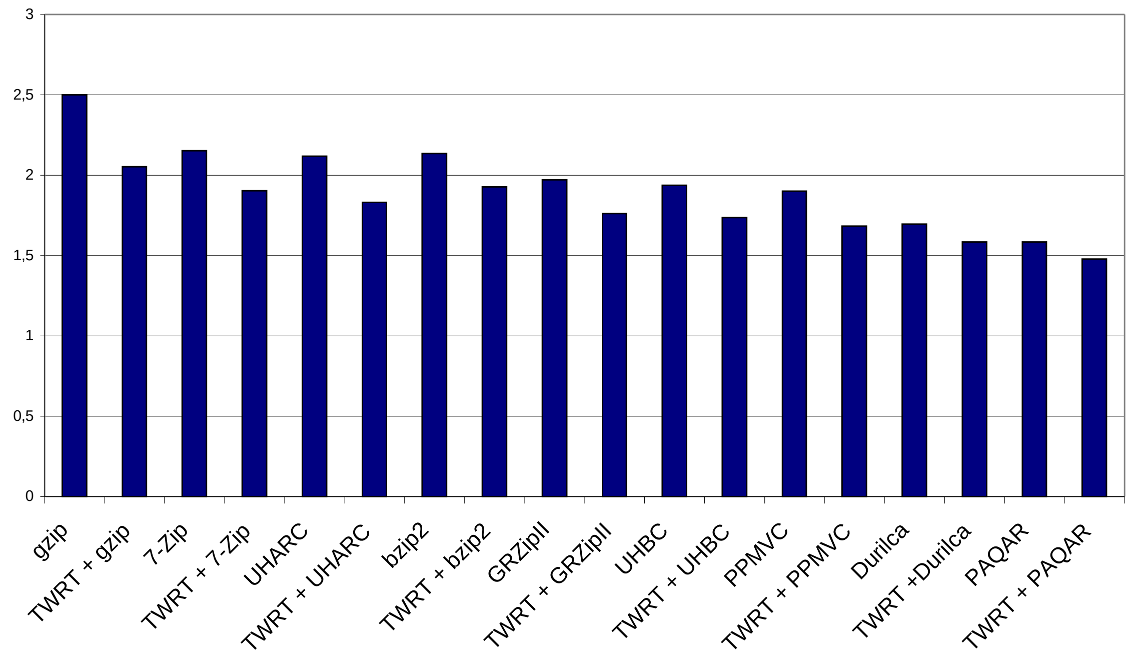

Figure 5.3: Average compression effectiveness (in bpc, lower is better) with TWRT and LZ, BWT, PPM, PAQ compressors on the Canterbury and the large Canterbury corpus.

File	gzip	TWRT + gzip	7-Zip	TWRT + 7-Zip	UHARC	TWRT + UHARC	bzip2	TWRT +bzip2	GRZipII	TWRT + GRZipII	UHBC	TWRT + UHBC
alice29.txt	2.863	2.060	2.555	1.999	2.550	1.927	2.272	1.916	2.143	1.825	2.142	1.809
asyoulik.txt	3.128	2.375	2.851	2.309	2.785	2.199	2.529	2.232	2.402	2.117	2.398	2.090
bible.txt	2.354	1.747	1.761	1.519	1.894	1.490	1.671	1.523	1.451	1.416	1.452	1.408
cp.html	2.599	2.528	2.518	2.388	2.465	2.302	2.479	2.322	2.335	2.176	2.342	2.123
E.coli	2.313	1.981	2.053	1.981	1.942	1.985	2.158	1.992	1.966	1.982	1.978	1.958
fields.c	2.254	2.157	2.215	2.107	2.221	2.059	2.180	2.191	2.072	1.976	2.069	1.952
grammar.lsp	2.670	2.522	2.915	2.666	2.765	2.500	2.758	2.737	2.556	2.380	2.481	2.260
kennedy.xls	1.606	0.379	0.392	0.260	0.334	0.244	1.012	0.193	0.922	0.192	0.647	0.163
lcet10.txt	2.716	2.058	2.243	1.852	2.283	1.752	2.019	1.777	1.885	1.688	1.882	1.666
plrabn12.txt	3.241	2.438	2.747	2.287	2.775	2.143	2.417	2.212	2.264	2.073	2.258	2.050
ptt5	0.880	0.692	0.686	0.612	0.494	0.500	0.776	0.529	0.744	0.488	0.703	0.454
sum	2.704	2.727	2.017	2.054	2.190	2.221	2.701	2.757	2.390	2.438	2.445	2.540
world192.txt	2.344	1.799	1.615	1.406	1.631	1.396	1.584	1.318	1.325	1.174	1.291	1.163
xargs.1	3.320	2.964	3.539	3.134	3.342	2.954	3.335	3.127	3.138	2.759	3.043	2.665
average	2.499	2.031	2.151	1.898	2.119	1.834	2.135	1.916	1.971	1.763	1.938	1.736
ctime	6.70	7.58	38.88	26.20	43.06	36.69	9.01	9.28	7.75	8.13	17.30	16.20
dtime	0.44	1.92	1.21	2.63	3.85	5.93	3.18	3.90	5.55	6.27	13.95	13.95
CTA	3.118	3.773	3.165	3.689	3.126	3.581	3.556	3.922	3.817	4.215	3.585	3.962

Table 5.4: Results of experiments with TWRT and LZ, BWT-based compressors on the Canterbury and the large Canterbury corpus. Results are given in bits per character (bpc). The compression and decompression times (ctime and dtime) are given in seconds.

File	PPMVC	TWRT + PPMVC	Durilca	TWRT + Durilca	StarNT + Durilca	Durilca with built-in models	PAQAR	TWRT + PAQAR
alice29.txt	2.035	1.753	1.974	1.708	1.862	**1.565**	1.863	1.652
asyoulik.txt	2.308	2.052	2.232	1.984	2.107	1.900	2.121	**1.907**
bible.txt	1.408	1.334	1.318	1.254	1.289	1.231	1.255	**1.223**
cp.html	2.135	2.028	2.077	1.940	2.026	2.011	1.949	**1.780**
E.coli	2.021	1.974	1.941	1.965	1.941*	1.941	**1.923**	**1.923**
fields.c	1.853	1.789	1.751	1.685	1.672	1.611	1.601	**1.554**
grammar.lsp	2.298	2.128	2.195	2.042	1.993	2.131	2.040	**1.830**
kennedy.xls	1.344	0.226	0.109	0.177	0.109*	0.109	**0.091**	**0.091**
lcet10.txt	1.793	1.595	1.711	1.531	1.655	**1.463**	1.595	1.494
plrabn12.txt	2.204	2.017	2.141	1.978	2.084	1.955	2.059	**1.930**
ptt5	0.743	0.475	0.503	0.448	0.503*	0.503	0.372	**0.372***
sum	2.360	2.365	1.840	1.853	1.840*	1.840	**1.638**	1.646
world192.txt	1.226	1.110	1.138	1.044	1.125	1.106	1.044	**0.963**
xargs.1	2.871	2.665	2.812	2.597	2.532	**2.324**	2.633	2.347
average	1.900	1.679	1.696	1.586	1.624	1.549	1.585	**1.479**
ctime	19.11	18.56	68.60	65.08	67.66	69.81	4799.12	3067.26
dtime	16.37	15.10	68.10	61.79	64.04	68.27	4799.00	3066.71
CTA	3.583	4.010	2.686	2.881	2.797	2.812	0.087	0.135

Table 5.5: Results of experiments with TWRT and PPM, PAQ compressors on the Canterbury and the large Canterbury corpus. Results are given in bits per character (bpc). The compression and decompression times (ctime and dtime) are given in seconds. Files marked with an asterisk were not preprocessed.

A coefficient of the transmission acceleration for the transmission speed 128 kbit/s for the Canterbury and the large Canterbury corpus and selected compressors is presented in Figure 5.4. TWRT improves the transmission acceleration for all tested compressors. This agrees with the thesis from the previous experiments that TWRT is proportionally slower on small files than on larger files. The highest coefficient of the transmission acceleration is obtained with GRZipII, PPMVC, UHBC, and bzip2.

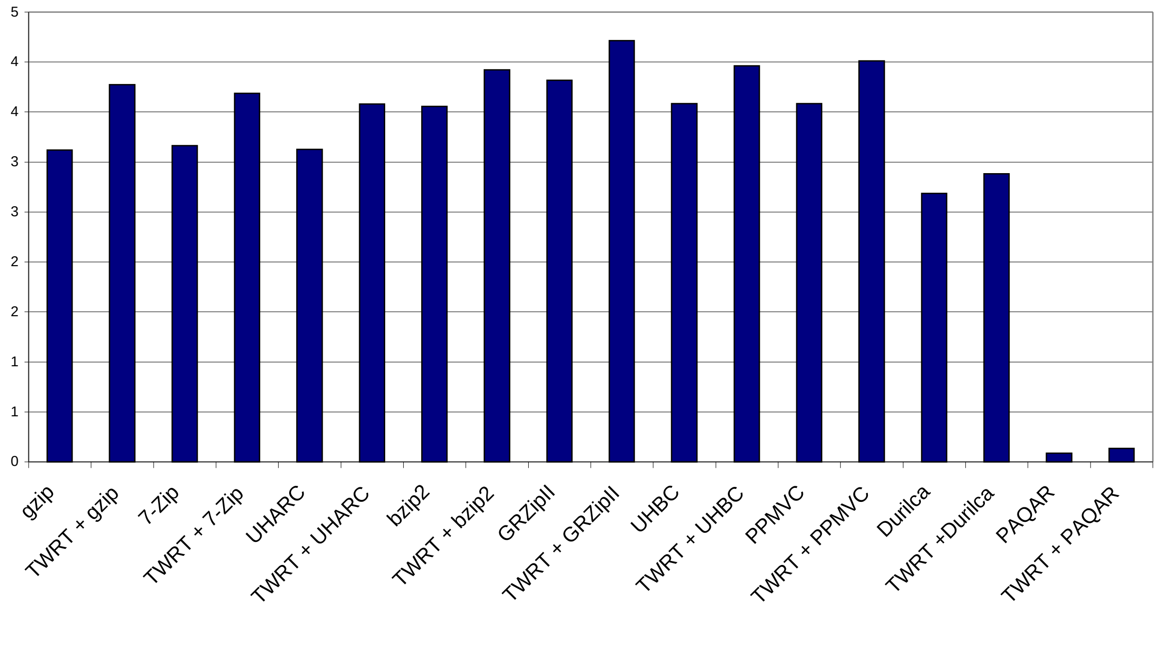

Figure 5.4: Coefficient of transmission acceleration (higher is better) with TWRT and LZ, BWT, PPM, PAQ compressors on the Canterbury and the large Canterbury corpus.

5.3.3 Experiments with the multilingual corpus

We have carried out our last experiments on multilingual text files. We have selected multilingual text files from Project Gutenberg [Ha04], as there is no well-known corpus with multilingual text files. The files are fully described in Appendix C.

Because all selected multilingual text files are books, TWRT does not use small dictionaries at all. To be more specific, the following dictionaries was automatically selected for files: *10055-8.txt* (DE), *12267-8.txt* (DE), *1musk10.txt* (ENG), *plrabn12.txt* (ENG), *11176-8.txt* (FR), *11645-8.txt* (FR), *rnpz810.txt* (PL), *sklep10.txt* (PL), *master.txt* (RU), *misteria.txt* (RU). The files other than English are tested with different encodings.

As can be seen in Figure 5.5, TWRT improves the compression effectiveness of all tested compressors. The improvement is significant and span from 7% (PAQ) to over 23% (gzip). Full results are available at Table 5.6 and Table 5.7.

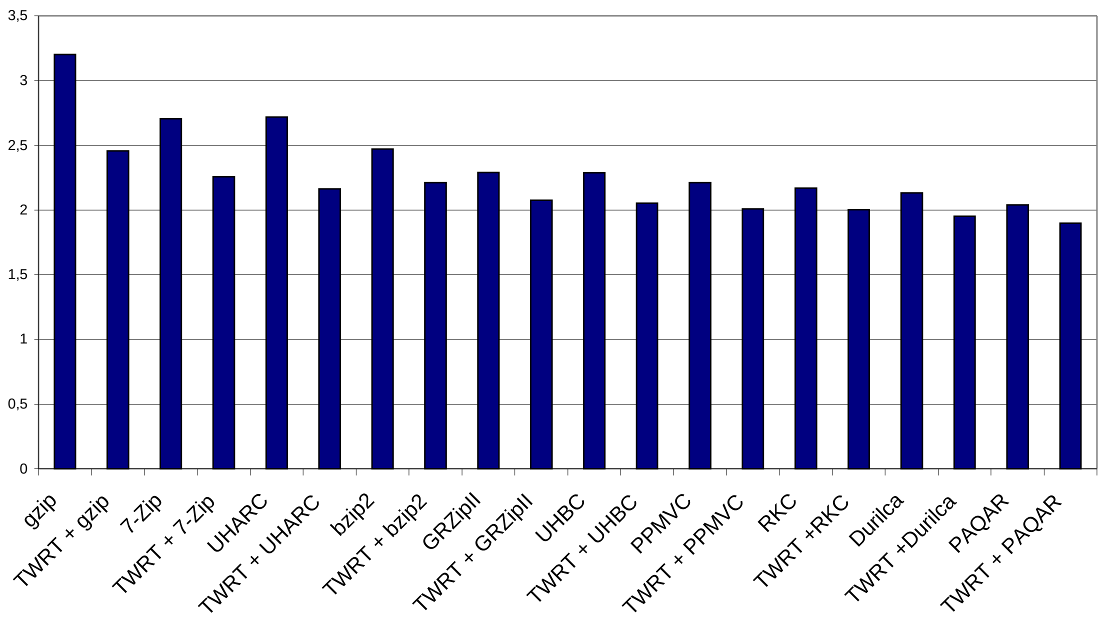

Figure 5.5: Average compression effectiveness (in bpc, lower is better) with TWRT and LZ, BWT, PPM, PAQ compressors on the multilingual corpus.

Table 5.7 also shows that TWRT with RKC achieves about 7% better compression effectiveness than StarNT in a connection with RKC, what is obvious as StarNT uses only English dictionary. Moreover, TWRT with RKC achieves about 4% better compression effectiveness than RKC with built-in the word-based preprocessing, which includes English, Russian and Polish dictionaries. TWRT with Durilca achieves about 6% better compression effectiveness than Durilca with English and Russian built-in models. Nevertheless, Durilca has better compression effectiveness on English and Russian files, but it seems that Russian built-in model works only with the codepage cp866.

As we expected, TWRT with PAQAR gives the best compression effectiveness. It has the best compression on nineteen from twenty-two files, and it only looses on three files to Durilca with built-in models.

File	encoding	gzip	TWRT + gzip	7-Zip	TWRT + 7-Zip	UHARC	TWRT + UHARC	bzip2	TWRT + bzip2	GRZipII	TWRT + GRZipII	UHBC	TWRT + UHBC
10055-8.txt	cp850	3.046	2.328	2.499	2.106	2.566	2.030	2.312	2.092	2.076	1.962	2.074	1.940
10055-8.txt	iso	3.046	2.328	2.499	2.105	2.566	2.029	2.312	2.094	2.076	1.963	2.074	1.940
12267-8.txt	cp850	3.010	2.266	2.535	2.090	2.567	2.011	2.302	2.109	2.128	1.968	2.125	1.954
12267-8.txt	iso	3.010	2.266	2.536	2.090	2.567	2.011	2.302	2.108	2.128	1.969	2.125	1.954
plrabn12.txt	ASCII	3.241	2.438	2.747	2.287	2.775	2.143	2.417	2.212	2.264	2.073	2.258	2.050
1musk10.txt	ASCII	2.920	2.016	2.300	1.847	2.390	1.792	2.080	1.770	1.844	1.671	1.842	1.655
11176-8.txt	cp850	2.847	2.008	2.258	1.806	2.333	1.759	2.015	1.758	1.851	1.668	1.853	1.652
11176-8.txt	iso	2.847	2.008	2.259	1.807	2.333	1.760	2.017	1.758	1.850	1.669	1.852	1.654
11645-8.txt	cp850	3.112	2.244	2.622	2.093	2.641	2.016	2.385	2.057	2.208	1.948	2.208	1.923
11645-8.txt	iso	3.112	2.244	2.622	2.095	2.641	2.018	2.386	2.057	2.208	1.948	2.208	1.932
rnpz810.txt	cp1250	3.384	2.549	2.862	2.354	2.854	2.252	2.600	2.349	2.426	2.205	2.422	2.183
rnpz810.txt	cp852	3.384	2.549	2.863	2.353	2.855	2.252	2.599	2.348	2.424	2.204	2.421	2.181
rnpz810.txt	iso	3.384	2.549	2.862	2.354	2.854	2.252	2.603	2.351	2.426	2.205	2.422	2.183
sklep10.txt	cp1250	3.596	2.710	3.242	2.553	3.155	2.441	3.022	2.598	2.867	2.451	2.861	2.408
sklep10.txt	cp852	3.596	2.710	3.244	2.550	3.155	2.441	3.017	2.596	2.864	2.450	2.859	2.406
sklep10.txt	iso	3.596	2.710	3.242	2.553	3.154	2.441	3.020	2.598	2.867	2.451	2.862	2.407
master.txt	cp1251	3.407	2.737	2.839	2.479	2.855	2.367	2.602	2.332	2.418	2.179	2.415	2.160
master.txt	cp866	3.409	2.737	2.841	2.494	2.870	2.381	2.602	2.337	2.417	2.184	2.414	2.166
master.txt	KOI8	3.405	2.737	2.839	2.479	2.855	2.367	2.603	2.334	2.419	2.182	2.416	2.164
misteria.txt	cp1251	3.029	2.635	2.590	2.381	2.613	2.276	2.388	2.251	2.200	2.094	2.196	2.078
misteria.txt	cp866	3.033	2.636	2.593	2.395	2.628	2.291	2.389	2.258	2.199	2.102	2.196	2.086
misteria.txt	KOI8	3.027	2.635	2.590	2.381	2.613	2.275	2.389	2.255	2.200	2.099	2.196	2.083
average		3.202	2.456	2.704	2.257	2.720	2.164	2.471	2.210	2.289	2.075	2.286	2.053
ctime		3.24	6.47	30.04	13.45	34.49	21.52	9.00	9.06	7.86	9.89	17.14	18.56
dtime		0.60	3.30	1.76	4.23	4.29	8.47	3.68	5.82	6.37	8.79	14.72	16.76
CTA		2.470	3.136	2.655	3.294	2.583	3.258	3.084	3.398	3.296	3.545	3.082	3.339

Table 5.6: Results of experiments with TWRT and LZ, BWT-based compressors on the multilingual corpus. Results are given in bits per character (bpc). The compression and decompression times (ctime and dtime) are given in seconds.

A coefficient of the transmission acceleration for transmission speed 128 kbit/s for the multilingual corpus and selected compressors is presented in Figure 5.6. Again, TWRT improves the transmission acceleration for all tested compressors. Moreover, TWRT + Durilca has a higher coefficient of the transmission acceleration than Durilca with built-in models. The highest coefficient of the transmission acceleration without TWRT is obtained respectively with GRZipII, PPMVC, and bzip2. The highest coefficient of the transmission acceleration in a connection with TWRT is obtained with the same compressors. These results can explain increasing popularity of BWT-based (GRZipII, bzip2) and PPM-based (PPMVC) archivers. The PPMVC algorithm, also presented in this dissertation, has shown very good trade-off between compression ratio and compression time.

File	encoding	PPMVC	TWRT + PPMVC	RKC	TWRT + RKC	StarNT + RKC	RKC with dictionary	Durilca	TWRT + Durilca	Durilca with built-in models	PAQAR	TWRT + PAQAR
10055-8.txt	cp850	2.024	1.874	1.981	1.874	1.981	1.868	1.941	1.814	1.893	1.846	1.757
10055-8.txt	iso	2.024	1.875	1.979	1.874	1.979	1.867	1.941	1.814	1.891	1.846	1.759
12267-8.txt	cp850	2.056	1.873	2.012	1.876	2.012	1.973	1.983	1.819	1.945	1.895	1.769
12267-8.txt	iso	2.056	1.873	2.010	1.876	2.010	1.972	1.983	1.819	1.943	1.895	1.771
plrabn12.txt		2.204	2.017	2.149	2.026	2.098	2.123	2.141	1.978	1.955	2.059	1.930
1musk10.txt		1.801	1.606	1.749	1.599	1.714	1.649	1.735	1.561	1.482	1.648	1.529
11176-8.txt	cp850	1.775	1.579	1.734	1.580	1.734	1.600	1.712	1.541	1.618	1.628	1.518
11176-8.txt	iso	1.775	1.580	1.733	1.581	1.733	1.599	1.712	1.543	1.611	1.627	1.522
11645-8.txt	cp850	2.117	1.866	2.075	1.870	2.075	1.951	2.050	1.821	2.013	1.967	1.782
11645-8.txt	iso	2.117	1.868	2.072	1.873	2.072	1.950	2.050	1.823	2.003	1.967	1.788
rnpz810.txt	cp1250	2.335	2.139	2.299	2.141	2.299	2.192	2.269	2.091	2.313	2.196	2.053
rnpz810.txt	cp852	2.335	2.138	2.301	2.133	2.301	2.193	2.269	2.089	2.317	2.196	2.052
rnpz810.txt	iso	2.335	2.138	2.300	2.138	2.300	2.192	2.269	2.089	2.309	2.196	2.053
sklep10.txt	cp1250	2.737	2.396	2.698	2.386	2.698	2.564	2.657	2.326	2.683	2.556	2.251
sklep10.txt	cp852	2.736	2.394	2.700	2.383	2.700	2.565	2.657	2.324	2.692	2.555	2.248
sklep10.txt	iso	2.736	2.394	2.698	2.383	2.698	2.564	2.657	2.324	2.676	2.555	2.250
master.txt	cp1251	2.351	2.141	2.312	2.130	2.312	2.291	2.260	2.080	2.260	2.161	2.018
master.txt	cp866	2.351	2.141	2.325	2.128	2.325	2.296	2.262	2.081	1.884	2.174	2.023
master.txt	KOI8	2.351	2.141	2.312	2.131	2.312	2.291	2.260	2.081	2.260	2.156	2.018
misteria.txt	cp1251	2.131	2.040	2.084	2.024	2.084	2.068	2.029	1.976	2.029	1.904	1.888
misteria.txt	cp866	2.131	2.040	2.095	2.023	2.095	2.074	2.030	1.976	1.754	1.921	1.894
misteria.txt	KOI8	2.131	2.040	2.083	2.024	2.083	2.068	2.029	1.976	2.029	1.898	1.887
average		2.210	2.007	2.168	2.002	2.164	2.087	2.132	1.952	2.071	2.038	1.898
ctime		14.33	14.79	186.75	177.74	187.13	178.29	81.78	86.57	89.96	4297.8	2760.5
dtime		14.16	14.66	210.42	192.90	208.72	201.91	80.46	83.87	87.72	4297.6	2759.6
CTA		3.217	3.489	1.328	1.428	1.331	1.384	2.158	2.218	2.107	0.094	0.144

Table 5.7: Results of experiments with TWRT and PPM, PAQ compressors on the multilingual corpus. Results are given in bits per character (bpc). The compression and decompression times (ctime and dtime) are given in seconds.

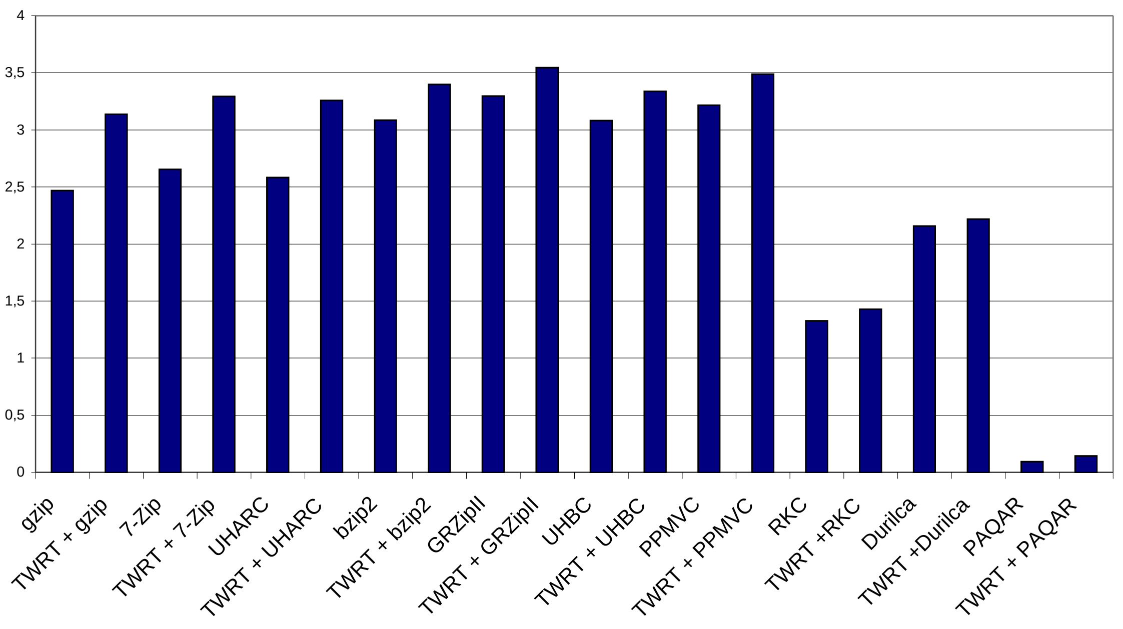

Figure 5.6: Coefficient of the transmission acceleration (higher is better) with TWRT and LZ, BWT, PPM, PAQ compressors on the multilingual corpus.

6 Conclusions

The subject of this dissertation are universal lossless data compression algorithms. It is a very important field of research as data compression allows to reduce the amount of space needed to store data or to reduce the amount of time needed to transmit data. It bring measurable profits expressed in a better usage of the existing technical infrastructure.

The background of a lossless data compression field was presented in Chapter 2. There we talked about main classes of practical lossless data compression algorithms (LZ, BWT-based, PPM, and PAQ). We have also introduced PPM with built-in models and word-based algorithms, based on universal compression methods, which are direct competitors for our word-based textual preprocessing. In Chapter 3 state-of-art predictive compression methods were improved with our two predictive compression algorithms—PPMEA and PPMVC. The experiments showed that PPMEA and PPMVC improve the average compression effectiveness for about 1% and 3% respectively in comparison to PPMII. In Chapter 4 we have presented most of well-known nowadays reversible data transforms that improve effectiveness of universal data compression algorithms. The biggest emphasis was placed on textual preprocessing and word-based textual preprocessing. The main contribution of this dissertation was presented in Chapter 5, with our two word-based textual preprocessing algorithms—WRT and TWRT. TWRT uses several dictionaries; it chooses for each file combination of two best suitable dictionaries depending of kind of an input file; it also automatically recognizes multilingual text files.

A particularly important issue, which usually has not attached an appropriate importance, is a dependency of the compression effectiveness from the compression and the decompression speed as well as the complexity. In this connection, we have compared lossless data compression algorithms, PPM with built-in models (Durilca), and word-based textual preprocessors (StarNT, TWRT), taking into consideration all these criteria (Chapter 5). For this purpose, we have used a coefficient of the transmission acceleration as a measure of the compression performance. We have also limited memory requirements of predictive compression methods to the contemporary standard, that is 256 MB.

The experiments were performed on well-known corpuses: the Calgary corpus, the Canterbury corpus, the large Canterbury corpus, and also on our multilingual corpus. TWRT significantly improves the compression effectiveness of universal lossless data compression algorithms; improvement spans from 5% (RKC on the Calgary corpus) to over 23% (gzip on the multilingual corpus). Moreover, TWRT achieves from 2% (on the Canterbury and the large Canterbury corpus) up to 7% (on the multilingual corpus) better compression effectiveness comparing to StarNT, a state-of-the-art word-based textual preprocessor.

TWRT influences in a very high degree on a coefficient of the transmission acceleration for transmission speed 128 kbit/s and selected compressors for the Calgary corpus. The influence spans from –9% (Durilca) to 25% (PAQ). The main reason for the deterioration of the transmission acceleration is that the Calgary corpus files are small, and TWRT is relatively slower on small files than on larger files. This thesis was confirmed in the following experiments, where TWRT influence on a coefficient of the transmission acceleration spans from 3% (Durilca on the multilingual corpus) to 36% (PAQ on the Canterbury and the large Canterbury corpus).

Concluding, TWRT significantly improves the compression effectiveness of universal lossless data compression algorithms (from 5% to over 23%). Moreover, TWRT has a very high encoding and decoding speed, which is amortized (especially on larger files) by a better compression effectiveness. It is confirmed using a coefficient of the transmission acceleration.

The computational complexity remains the same as TWRT works in a linear time. Furthermore, TWRT requires only about 10 MB of memory, which is allocated before or after the actual compression. We conclude that TWRT significantly improves compression ratio of lossless data compression algorithms, while the compression and the decompression speed as well as the complexity remain almost the same.

The results in this work can surely be a point of departure for further research. Please note that TWRT can be used in a spell checking. It has multilingual dictionaries and it can help the spell-checker with an automatically detecting language of a textual file. The preprocessor and the spell-checker can share the same dictionaries. Spell checker's dictionaries are probably bigger than adequate TWRT dictionaries so some last words from the dictionaries will not be used in the preprocessing.

TWRT can be used as a converter of encodings in multilingual text files. It outputs postprocessed files with the same encoding as occurs in the input file, but it can be easily changed. A more interesting idea is an adding national characters to multilingual text files with stripped national characters. Currently for this kind of files, TWRT reads a proper dictionary and strips national characters from it. Nevertheless, if we find words in a stripped dictionary, then we can output original words, with national characters. But there is a problem with words that have the same form after stripping, e.g., Polish words 'sad' and 'sąd'. We do not know if 'sad' should be interpreted as 'sąd' or 'sad'. It is an interesting subject for further research.

Another interesting possibility is searching for a word-based pattern (or multiple patterns at once) directly in the TWRT-transformed text, preferably in the TWRT-LZ version (a higher data reduction rate, which also implies a faster search). Putting apart the symbols not belonging to encoded words (that is, punctuation marks, letters in words from outside the dictionary, etc.), the remaining TWRT-LZ transformed data can be seen as a sequence of (*s*, *c*)-Dense Code [BF$^+$03], which is a prefix byte-aligned[2] code enabling fast Boyer-Moore search over it. To keep things simple for any given pattern, the conversion of End-of-Line symbols should be omitted.

Several other ideas require further investigations. First, words in the dictionary may be labeled with part-of-speech and/or category tags, and experiments performed by Smirnov [Sm02a] have showed that such an additional information in the dictionary is beneficial for the compression. The drawback of this approach is that preparing dictionaries for various languages gets very laborious. Second, it might be possible to achieve a slightly higher compression by extending the dictionary with a number of frequent phrases in a given language, consisting of a few words.

[2] The (*s*, *c*)-Dense Codes are quite a broad class of codes and byte-alignment is only a practical, not necessary, condition.

Acknowledgements

Heartfelt thanks to Szymon Grabowski with whom I discussed many ideas and technical problems that I have encountered during my Ph.D. studies. Thank you Szymon for introducing me to Maxim Smirnov, Uwe Herklotz, Jürgen Abel, and Jakub Swacha. Also thanks to Sebastian Deorowicz for introducing me to Szymon Grabowski.

Thanks to Szymon Grabowski and Sebastian Deorowicz for a collaboration on papers we wrote together. I also thank to Szymon Grabowski, Jakub Swacha, and Sebastian Deorowicz for proofreading this dissertation, for suggestions of possible improvements, and for constructive remarks on my work.

Some parts of the PPMVC compressor and WRT/TWRT preprocessors are based on existing source codes. Thanks to Dmitry Shkarin for making public the sources of his excellent PPMd, on which the PPMVC implementation is based. The arithmetic coding routines used in WRT/TWRT preprocessors are modified source codes by Dmitry Shkarin and Eugene Shelwien.

I would also like to thank dr hab. Marek Piotrów, who supervised my research in data compression. Last but not least, I would like to thank my parents—Mirosława and Janusz—and my girlfriend—Magdalena Więcek—for their love, care, support and encouragement over all these years.

Przemysław Skibiński
Wrocław, Poland
May, 2005

Bibliography

[AT05] J. Abel and W. Teahan. Universal Text Preprocessing for Data Compression. *IEEE Transactions on Computers*, 54 (5), pp. 497–507, May 2005.

[Ab04] J. Abel. Record preprocessing for data compression. *Proceedings of the IEEE Data Compression Conference (DCC'04)*, pp. 521, 2004.

[AS+97] J. Aberg, Yu. M. Shtarkov, and B. J. M. Smeets. Towards understanding and improving escape probabilities in PPM. *Proceedings of the IEEE Data Compression Conference (DCC'97)*, pp. 22, 1997.

[Ab63] N. Abramson. Information Theory and Coding. New–York: McGraw–Hill, 1963.

[AF+04] J. Adiego, P. de la Fuente, and G. Navarro. Merging Prediction by Partial Matching with Structural Contexts Model. *Proceedings of the IEEE Data Compression Conference (DCC'04)*, pp. 522, 2004.

[AN+03] J. Adiego, G. Navarro, and P. de la Fuente. SCM: Structural Contexts Model for Improving Compression in Semistructured Text Databases. *Proceedings of the String Processing and Information Retrieval (SPIRE 2003)*, 10th International Symposium, pp. 153–167, 2003.

[AN+04] J. Adiego, G. Navarro, and P. de la Fuente. Lempel–Ziv Compression of Structured Text. *Proceedings of the IEEE Data Compression Conference (DCC'04)*, pp. 112–121, 2004.

[AL98] A. Apostolico and S. Lonardi. Some Theory and Practice of Greedy Off-Line Textual Substitution. *Proceedings of the IEEE Data Compression Conference (DCC'98)*, pp. 119–128, 1998.

[AL00] A. Apostolico and S. Lonardi. Off-line compression by greedy textual substitution. *Proceedings of the IEEE Data Compression Conference (DCC'00)*, pp. 143–152, 2000.

[AM97] Z. Arnavut and S. S. Magliveras. Block sorting and compression. *Proceedings of the IEEE Data Compression Conference (DCC'97)*, pp. 181–190, 1997.

[AB97] R. Arnold and T. C. Bell. A corpus for the evaluation of lossless compression algorithms. *Proceedings of the IEEE Data Compression Conference (DCC'97)*, pp. 201–210, 1997. http://corpus.canterbury.ac.nz/descriptions/#large.

[As04] M. T. Ashland. Monkeys Audio Compressor (MAC) 3.99 (computer program). April 2004. http://www.monkeysaudio.com/.

[Au00] P. J. Ausbeck. The Piecewise-Constant Image Model. *Proceedings of the IEEE*, 88 (11), pp. 1779–1789, November 2000.

[AZ+01] F. Awan, N. Zhang, N. Motgi, R. Iqbal, and A. Mukherjee. LIPT: A Reversible Lossless Text Transform to Improve Compression Performance. *Proceedings of the IEEE Data Compression Conference (DCC'01)*, pp. 481, 2001.

[BK+99] B. Balkenhol, S. Kurtz, and Y. M. Shtarkov. Modifications of the Burrows and Wheeler data compression algorithm. *Proceedings of the IEEE Data Compression Conference (DCC'99)*, pp. 188–197, 1999.

[BF99] R. Barequet and M. Feder. SICLIC: A Simple Inter-Color Lossless Image Coder. *Proceedings of the IEEE Data Compression Conference (DCC'99)*, pp. 501–510, 1999.

[Ba79] D. S. Batory. On Searching Transposed Files. *ACM Transactions on Database Systems (TODS)*, Volume 4 (4), pp. 531–544, December 1979.

[Be87] T. C. Bell. A unifying theory and improvements for existing approaches to text compression. Ph.D. dissertation, Department of Computer Science, University of Canterbury, New Zealand, 1987.

[BM+93] T. C. Bell, A. Moffat, C. G. Nevill-Manning, I. H. Witten, and J. Zobel. Data compression in full-text retrieval systems. *J. Am. Soc. Inf. Sci. 44* (9), pp. 508–531, October 1993.

[BW+89] T. C. Bell, I. H. Witten, and J. Cleary. Modeling for Text Compression. *ACM Computing Surveys*, Vol. 21, No. 4, December 1989.

[BC+90] T. C. Bell, J. Cleary, and I. H. Witten. Text compression. Advanced Reference Series. Prentice Hall, Englewood Cliffs, New Jersey, 1990. http://corpus.canterbury.ac.nz/descriptions/#calgary.

[BM01] J. Bentley and D. McIlroy. Data compression with long repeated strings. *Information Sciences* 135 (1–2) pp. 1–11, 2001.

[BS+86] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29 (4), pp. 320–330, April 1986.

[Be05] W. Bergmans. Lossless data compression software benchmarks / comparisons. 2005. http://www.maximumcompression.com/.

[Be04] M. Bevin. La (Lossless Audio) 0.4b (computer program). February 2004. http://www.lossless-audio.com/download.htm.

[Bi00] E. Binder. The DC archiver v0.98b (computer program). 2000. ftp://ftp.elf.stuba.sk/pub/pc/pack/dc124.zip.

[Bl96] C. Bloom. LZP — a new data compression algorithm. *Proceedings of the IEEE Data Compression Conference (DCC'96)*, pp. 425, 1996.

[Bl98] C. Bloom. Solving the problems of context modeling. 1998. http://www.cbloom.com/papers/ppmz.zip.

[Bl99] C. Bloom. PPMZ2—High Compression Markov Predictive Coder. 1999. http://www.cbloom.com/src/.

[Bo05] J. de Bock. Ultimate Command Line Compressors. 2005. http://uclc.info/.

[BK93] A. Bookstein and S. T. Klein. Is Huffman coding dead?. *Computing 50*, 4, pp. 279–296.

[BB+96] M. Bosi, K. Brandenburg, Sch. Quackenbush, L. Fielder, K. Akagiri, H. Fuchs, M. Dietz, J. Herre, G. Davidson, and Yoshiaki Oikawa. ISO/IEC MPEG-2 Advanced Audio Coding. *Proceedings of the 101st AES Convention*, preprint 4382, 1996.

[BH+98a] L. Bottou, P. Haffner, P. G. Howard, P. Simard, Y. Bengio, and Y. LeCun. High quality document image compression with "DjVu". *Journal of Electronic Imaging*, 7 (3), pp. 410–425, July 1998.

[BH+98b] Q. Bradley, R. N. Horspool, and J. Vitek. JAZZ: an efficient compressed format for Java archive files. Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, pp. 7, November–December 1998.

[BS94] K. Brandenburg and G. Stoll. ISO-MPEG-1 Audio: A Generic Standard for Coding of High Quality Digital Audio. *Journal of the Audio Engineering Society*, 10, pp. 780–792, October 1994.

[BF+03] N. Brisaboa, A. Farina, G. Navarro, and M. Esteller. (S,C)-Dense Coding: An Optimized Compression Code for Natural Language Text Databases. *Lecture Notes in Computer Science* 2857, pp. 122–136, 2003.

[Bu97a] S. Bunton. Semantically motivated improvements for PPM variants. *The Computer Journal* 40 (2/3), pp. 76–93, 1997.

[BW94] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. *Digital Equipment Corporation (SRC Research Report 124)* 1994. ftp://gatekeeper.research.compaq.com/pub/DEC/SRC/research-reports/SRC-124.pdf.

[BC+00] A. L. Buchsbaum, D. F. Caldwell, K. W. Church, G. S. Fowler, and S. Muthukrishnan. Engineering the Compression of Massive Tables – An Experimental Approach. *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 175–184, January 2000.

[BF+02] A. L. Buchsbaum, G. S. Fowler, and R. Giancarlo. Improving Table Compression with Combinatorial Optimization. *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 213–222, 2002.

[Bu97b] G. Buyanovsky. ACB 2.00c (computer program). 1997. ftp://ftp.elf.stuba.sk/pub/pc/pack/acb_200c.zip.

[CT98] B. Chapin and S.R. Tate. Higher Compression from the Burrows–Wheeler Transform by Modified Sorting. *Proceedings of the IEEE Data Compression Conference (DCC'98)*, pp. 532, 1998.

[CK+99] X. Chen, S. Kwong, and M. Li. A compression algorithm for DNA sequences and its applications in genome comparison. *Proceedings of the Fourth Annual International Conference on Computational Molecular Biology (RECOMB 2000),* pp. 107, 2000.

[CL+02] X. Chen, M. Li, B. Ma, and J. Tromp. DNACompress: fast and effective DNA sequence compression. *Bioinformatics*, 18 (12), pp. 1696–1698, 2002.

[Ch01] J. Cheney. Compressing XML with multiplexed hierarchical models. *Proceedings of the IEEE Data Compression Conference (DCC'01)*, pp. 163–172, 2001.

[CS+00] C. Christopoulos, A. Skodras, and T. Ebrahimi. The JPEG2000 still image coding system: An Overview. *IEEE Transactions on Consumer Electronics*, 46 (4), pp. 1103–1127, November 2000.

[CR97] P. R. Clarkson and A. J. Robinson. Language model adaptation using mixtures and an exponentially decaying cache. IEEE ICASSP, Vol. 2, pp. 799–802, 1997.

[CW84] J.G. Cleary and I.H Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, COM-32 (4), pp. 396–402, 1984.

[CT+95] J. G. Cleary, W. J. Teahan, and I. H. Witten. Unbounded length contexts for PPM. *Proceedings of the IEEE Data Compression Conference (DCC'95)*, pp. 52–61, 1995.

[Co05] J. Coalson. FLAC (Free Lossless Audio Coder) 1.1.2 (computer program). February 2005. http://flac.sourceforge.net/.

[Cr95] L. D. Crocker. PNG: The Portable Network Graphic Format. *Dr. Dobb's Journal*, 20 (7), pp. 36–44, July 1995.

[DE+00] S. K. Debray, W. Evans, R. Muth, and B. Sutter. Compiler techniques for code compaction. ACM Transactions on Programming Languages and Systems (TOPLAS), 22 (2), pp. 378–415, March 2000.

[De00] S. Deorowicz. Improvements to Burrows–Wheeler compression algorithm. *Software—Practice and Experience,* 30 (13), pp. 1465–1483, 2000.

[De02] S. Deorowicz. Second step algorithms in the Burrows–Wheeler compression algorithm. *Software—Practice and Experience*, 32 (2), pp. 99–111, 2002.

[De03a] S. Deorowicz. Universal lossless data compression algorithms. Ph.D. dissertation, Silesian University of Technology, Gliwice, Poland, June 2003.

[De03b] S. Deorowicz. Silesia compression corpus. 2003. http://www-zo.iinf.polsl.gliwice.pl/~sdeor/corpus.htm.

[DK02] M. Drinić and D. Kirovski. PPMexe: PPM for Compressing Software. *Proceedings of the IEEE Data Compression Conference (DCC'02),* pp. 192–201, 2002.

[DP+99] J. Dvorský, J. Pokorný, and V. Snásel. Word-Based Compression Methods and Indexing for Text Retrieval Systems. *ADBIS '99: Proceedings of the Third East European Conference on Advances in Databases and Information Systems*, pp. 75–84, 1999.

[Ea02] Eastridge Technology. JShrink 2.0 (computer program). April 2002. http://www.e-t.com/jshrink.html.

[EE+97] J. Ernst, W. Evans, C. W. Fraser, T. A. Proebsting, and S. Lucco. Code compression. *ACM SIGPLAN Notices*, 32 (5), pp. 358–365, June 1997.

[Fa73] N. Faller. An adaptive system for data compression. *Record of the 7th Asilomar Conference on Circuits, Systems, and Computers*, pp. 593–597, 1973.

[Fa03] M. D. Farhat. BIM Obfuscator (computer program). December 2003. http://bimopensourcesolutions.freewebspace.com/.

[Fe96] P. Fenwick. The Burrows–Wheeler Transform for block sorting text compression: principles and improvements. *The Computer Journal*, 39 (9), pp.731–740, 1996.

[FP02] P. J. Ferreira and A. J. Pinho. Histogram packing, total variation, and lossless image compression. Signal Processing XI — Theories and Applications. *Proceedings of EUSIPCO-2002, XI European Signal Processing Conference*, vol. II, pp. 498–501, September 2002.

[FP97] J. Forbes and T. Poutanen. Microsoft LZX Data Compression Format (part of Microsoft Cabinet SDK). 1997.

[FM96] R. Franceschini and A. Mukherjee. Data compression using encrypted text. *Proceedings of the IEEE Data Compression Conference (DCC'96)*, pp. 437, 1996.

[FK+00] R. Franceschini, H. Kruse, N. Zhang, R. Iqbal, and A. Mukherjee. Lossless, Reversible Transforms that Improve Text Compression Ratios. Preprint of the M5 Lab, University of Central Florida, 2000.

[Fr99] C. W. Fraser. Automatic inference of models for statistical code compression. *ACM SIGPLAN Notices*, 34 (5), pp. 242–246, May 1999.

[Ga93] J.–L. Gailly. gzip 1.2.4 (computer program). August 1993. http://www.gzip.org/.

[Gh04] F. Ghido, OptimFROG 4.509 (computer program). May 2004. http://losslessaudiocompression.com/.

[GS00] M. Girardot and N. Sundaresun. Millau: an encoding format for efficient representation and exchange of XML over the Web. WWW9 proceedings, IBM Almaden Research Center, 2000. http://www9.org/w9cdrom/154/154.html.

[Gr99] Sz. Grabowski. Text Preprocessing for Burrows–Wheeler Block–Sorting Compression. *Proceedings of VII Konferencja Sieci i Systemy Informatyczne – Teoria, Projekty, Wdrożenia*, Łódź, Poland, pp. 229–239, 1999.

[Gr04] I. Grebnov. GRZip II version 0.2.4. May 2004. http://magicssoft.ru/?folder=projects&page=GRZipII.

[GT93] S. Grumbach, and F. Tahi. Compression of DNA Sequences. *Proceedings of the IEEE Data Compression Conference (DCC'93)*, pp. 340–350, 1993.

[Gu80] M. Guazzo. A general minimum-redundancy source-coding algorithm. *IEEE Transactions on Information Theory*, IT-26 (1), pp. 15–25, January 1980.

[HL72] W. D. Hagamen, D. J. Linden, H. S. Long, and J. C. Weber. Encoding verbal information as unique numbers. *IBM Systems Journal 11*, pp. 278–315, October 1972.

[Ha04] M. S. Hart. Project Gutenberg, http://www.gutenberg.org/.

[He03] U. Herklotz. UHBC 1.0 (computer program). June 2003. ftp://ftp.elf.stuba.sk/pub/pc/pack/uhbc10.zip.

[He04] U. Herklotz. UHARC 0.6 (computer program). December 2004. ftp://ftp.elf.stuba.sk/pub/pc/pack/uharc06.zip.

[HC98] R. N. Horspool and J. Corless. Tailored compression of Java class files. *Software—Practice & Experience*, 28 (12), pp. 1253–1268, October 1998.

[Ho75] J. A. Hoffer. A clustering approach to the generation of subfiles for the design of a computer database. Ph.D. dissertation, Cornell II., Ithaca, N.Y., January 1975.

[HC84] R. N. Horspool and G. V. Cormack. A general purpose data compression technique with practical applications. *Proceedings of the CIPS Session 84*, pp. 138–141, 1984.

[HC92] R. N. Horspool and G. V. Cormack. Constructing Word–Based Text Compression Algorithms. *Proceedings of the IEEE Data Compression Conference (DCC'92)*, pp. 62–71, 1992.

[Ho93] P. G. Howard. The design and analysis of efficient lossless data compression systems. Technical Report CS–93–28, Brown University, Providence, Rhode Island, 1993. ftp://ftp.cs.brown.edu/pub/techreports/93/cs93-28.ps.Z.

[HK+98] P. G. Howard, F. Kossentini, B. Martins, S. Forchhammer, and W.J. Rucklidge. The emerging JBIG2 standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 8 (7), pp. 838–848, November 1998.

[HV93] P. G. Howard and J. S. Vitter. Fast and Efficient Lossless Image Compression. *Proceedings of the IEEE Data Compression Conference (DCC'93)*, pp. 351–360, 1993.

[Hu52] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40 (9), pp. 1098–1101, September 1952.

[IM01] R. Y. K. Isal and A. Moffat. Word-based block-sorting text compression. *Australian Computer Science Communications*, 23 (1), pp. 92–99, January-February 2001.

[IM+02] R. Y. K. Isal, A. Moffat, A. C. H. Ngai. Enhanced word-based block-sorting text compression. *Australian Computer Science Communications*, 24 (1), pp. 129–137, January-February 2002.

[IT80] International Telecommunication Union. ITU-T (CCITT) Recommendation T.4 — Standardization of Group 3 Facsimile Terminals for Document Transmission. 1980.

[IT88] International Telecommunication Union. ITU-T (CCITT) Recommendation T.6 — Facsimile coding schemes and coding control functions for group 4 facsimile apparatus. October 1988.

[JJ92] J. Jiang and S. Jones. Word-based dynamic algorithms for data compression. *Communications, Speech and Vision, IEE Proceedings I*, 139 (6), pp. 582–586, December 1992.

[KR87] R. M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development* 32 (2), pp. 249–260, 1987.

[KY00] J. C. Kieffer and E. Yang. Grammar-Based Codes: a New Class of Universal Lossless Source Codes. *IEEE Transactions on Information Theory*, 46 (3), pp. 737–754, 2000.

[KY$^+$00] J. C. Kieffer, E. Yang, G. J. Nelson, and P. Cosman. Universal Lossless Compression via Multilevel Pattern Matching. *IEEE Transactions on Information Theory*, 46 (3), pp. 1227–1245, 2000.

[Ki04] J.-H. Kim. Lossless Wideband Audio Compression: Prediction and Transform. Ph.D. dissertation, Technische Universitat Berlin, 2004.

[KM98] H. Kruse and A. Mukherjee. Preprocessing Text to Improve Compression Ratios, *Proceedings of the IEEE Data Compression Conference (DCC'98)*, pp. 556, 1998.

[LT$^+$98] C. Laffra, F. Tip, and P. Sweeny. JAX—the Java Application eXtractor (computer program). 1998. http://www.alphaworks.ibm.com/formula/JAX.

[La04] E. Lafortune. ProGuard 3.2 (computer program). December 2004. http://proguard.sourceforge.net/.

[La84] G. G. Langdon. An introduction to arithmetic coding. *IBM Journal of Research and Development*, 28(2), pp. 135–149, March 1984.

[LZ05] J. Lansky and M. Zemlicka. Text Compression: Syllables. *Proceedings of Annual International Workshop on DAtabases, TExts, Specifications and Objects (Dateso 2005)*, pp. 32–45, 2005.

[LM99] N. J. Larsson and A. Moffat. Offline Dictionary-Based Compression. *Proceedings of the IEEE Data Compression Conference (DCC'99)*, pp. 296–305, 1999.

[Le03] M. Lemke. WinAce 2.5 (computer program). August 2003. http://www.winace.com/.

[LH$^+$94] A. Lettieri, K. Holtz, and E. Holtz. Data compression in the V.42bis modems. *IEEE WESCON'94 Conference Record*, pp. 398-403, September 1994.

[Li02] T. Liebchen. Lossless Audio Coding Using Adaptive Multichannel Prediction. *Proceedings of the 113th AES Convention*, preprint 5680, 2002.

[Li04] T. Liebchen. LPAC (Lossless Predictive Audio Compression) Archiver 1.41 (computer program). September 2004. http://www.nue.tu-berlin.de/wer/liebchen/lpac.html.

[LP+99] T. Liebchen, M. Purat, and P. Noll. Improved Lossless Transform Coding of Audio Signals. Impulse und Antworten - Festschrift für Manfred Krause, Wissenschaft & Technik Verlag, Berlin, pp. 159–170, 1999.

[LS00] H. Liefke and D. Suciu. XMILL: An efficient compressor for XML data. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 153–164, 2000.

[Lu00] S. Lucco. Split-stream dictionary program compression. *ACM SIGPLAN Notices*, 35 (5), pp. 27–34, May 2000.

[Ma02] M. V. Mahoney. The PAQ1 data compression program. 2002. http://www.cs.fit.edu/~mmahoney/compression/paq1.pdf.

[Ma03] M. V. Mahoney. PAQ6 (computer program). 2003. http://www.cs.fit.edu/~mmahoney/compression/.

[MR04a] M. V. Mahoney and A. Ratushnyak. PAQAR 4.0 (computer program). 2004. http://www.cs.fit.edu/ ~mmahoney/compression/.

[MR+06] M. V. Mahoney, A. Ratushnyak, and P. Skibiński. PASQDA 4.4 (computer program). January 2006. http://www.ii.uni. wroc.pl/~inikep/research/pasqda44.zip.

[MR04b] G. Manzini and M. Rastero. A simple and fast DNA compressor. *Software—Practice and Experience*, 34 (14), pp. 1397–1411, 2004.

[MS+00] T. Matsumoto, K. Sadakane, and H. Imai. Biological Sequence Compression Algorithms. *Genome Informatics 2000*, pp. 43–52, 2000.

[MP+03] J.-K. Min, M.-J. Park, and C.-W. Chung. XPRESS: A Queriable Compression for XML Data. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 122–133, 2003**.**

[Mo89] A. Moffat. Word based text compression, *Software–Practice and Experience* 19 (2), pp. 185–198, 1989.

[Mo90] A. Moffat. Implementing the PPM data compression scheme. *Proceedings of the IEEE Transactions on Communications*, 38 (11), pp. 1917–1921, November 1990.

[Mo05] A. Moffat. Private correspondence. May 2005.

[MN+98] A. Moffat, R. M. Neal, and I. H. Witten. Arithmetic coding revisited. *ACM Transactions on Information Systems*, 16 (3), pp. 256–294, 1998.

[MT97] A. Moffat and A. Turpin. On the implementation of minimum-redundancy prefix codes. *IEEE Transactions on Communications*, 45 (10), pp. 1200–1207, 1997.

[MN$^+$97] E. S. de Moura, G. Navarro, and N. Ziviani. Indexing Compressed Text. *Proceedings of the 4th South American Workshop on String Processing (WSP'97)*, pp. 95–111, 1997.

[MN$^+$00] E. S. de Moura, G. Navarro, N. Ziviani, R. Baeza-Yates. Fast and Flexible Word Searching on Compressed Text. *ACM Transactions on Information Systems*, 18 (2), pp. 113–139, April 2000.

[Ne96] C. G. Nevill-Manning. Inferring sequential structure. Ph.D. dissertation, Computer Science Department, University of Waikato, New Zealand, 1996.

[Pa76] R. C. Pasco. Source Coding Algorithms for Fast Data Compression. Ph.D. dissertation, Department of Electrical Engineering, Stanford University, California, USA, November 1976.

[Pa05] I. Pavlov. 7-Zip 4.17 beta (computer program). April 2005. http://7-zip.org/.

[PM92] W. B. Pennebaker and J.L. Mitchell. The JPEG Still Image Data Compression Standard. Van Nostrand Reinhold, 1992.

[Pr04] PreEmptive Solutions. Dash0 Professional 3.1 (computer program). October 2004.

[Pu99] W. Pugh. Compressing Java class files. *ACM SIGPLAN Notices*, 34 (5), pp. 247–258, May 1999.

[Ra98] V. Ratnakar. RAPP: lossless image compression with runs of adaptive pixel patterns. *Conference Record of the 32nd Asilomar Conference on Signals, Systems and Computers*, vol. 2, pp. 1251–1255, November 1998.

[RT$^+$02] J. Rautio, J. Tanninen, and J. Tarhio. String matching with stopper compression. *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching (CPM'02)*, Springer, pp. 42–52, 2002.

[Ri76] J. Rissanen. Generalised Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20, pp. 198–203, 1976.

[RL79] J. Rissanen and G. G. Langdon. Arithmetic coding. *IBM Journal of Research and Development*, 23 (2), pp. 149–162, 1979.

[RL81] J. Rissanen and G. G. Langdon. Universal modeling and coding. *IEEE Transactions on Information Theory*, 27 (1), pp. 12–23, 1981.

[RD$^+$96] E. Rivals, J. Delahaye, M. Dauchet, and O. Delgrange. A Guaranteed Compression Scheme for Repetitive DNA Sequences. *Proceedings of the IEEE Data Compression Conference (DCC'96)*, pp. 453, 1996.

[Ro04] E. Roshal. RAR 3.41 (computer program). 2004. http://www.rarsoft.com.

[Ru79] F. Rubin. Arithmetic stream coding using fixed precision registers. *IEEE Transactions on Information Theory*, IT-25 (6), pp. 672–675, November 1979.

[Ry80] B. Y. Ryabko. Data compression by means of a 'book stack'. *Prob. Inf. Transm.*, 16 (4), 1980. In Russian.

[RH+87] B. Y. Ryabko, R. N. Horspool, G. V. Cormack, S. Sekar, and S. B. Ahuja. Technical Correspondence. *Communications of the ACM*, 30 (9), pp. 792–796, September 1987.

[Sa99] K. Sadakane. Unifying Text Search and Compression — Suffix Sorting, Block Sorting and Suffix Arrays. Ph.D. dissertation, The University of Tokyo, December 1999.

[SP96] A. Said and W.A. Pearlman. A new, fast, and efficient image coded based on set partitioning in hierarchical trees. *IEEE Transactions on Circuits Systems Video Technologies*, 6(3), pp. 243–249, 1996.

[Sa05] Sandhills Publishing Company. Huffman code. 2005. http://www.smartcomputing.com/editorial/dictionary/detail.asp?searchtype=1&DicID=17660&RefType=Encyclopedia

[Se02] J. Seward. bzip2 1.0.2 (computer program). January 2002. http://sources.redhat.com/bzip2/.

[Sh48] C. E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27, pp. 379–423, 623–656, 1948.

[Sh51] C. E. Shannon. Prediction and entropy of printed English. *Bell System Technical Journal*, 30, pp. 50–64, 1951.

[Sh02a] D. Shkarin. PPM: one step to practicality. *Proceedings of the IEEE Data Compression Conference (DCC'02)*, pp. 202–211, 2002.

[Sh02b] D. Shkarin. PPMd and PPMonstr, var. I (computer programs). April 2002. http://compression.graphicon.ru/ds/.

[Sh04] D. Shkarin. Durilca Light and Durilca 0.4b (computer programs). September 2004. http://compression.graphicon.ru/ds/.

[Sk06] P. Skibiński. PPM with extended alphabet. *Information Sciences*, 176 (7), pp. 861–874, April 2006. http://dx.doi.org/10.1016/j.ins.2005.01.014.

[Sk05b] P. Skibiński. Two-level directory based compression. *Proceedings of the IEEE Data Compression Conference (DCC'05)*, pp. 481, 2005. http://www.ii.uni.wroc.pl/~inikep/papers/05-TwoLevelDict.pdf.

[Sk03] P. Skibiński. PPMVC 1.0 (computer program). July 2003. http://www.ii.uni.wroc.pl/~inikep/research/PPMVC10.zip.

[Sk05a] P. Skibiński. TWRT (computer program). 2005. http://www.ii.uni.wroc.pl/~inikep/research/WRT44.zip.

[SD04a] P. Skibiński and S. Deorowicz. WRT (computer program). 2004. http://www.ii.uni.wroc.pl/~inikep/research/WRT30d.zip.

[SD04b] P. Skibiński and S. Deorowicz. WRT-LZ77 (computer program). 2004. http://www.ii.uni.wroc.pl/~inikep/research/WRT-LZ77.zip.

[SG04] P. Skibiński and Sz. Grabowski. Variable-length contexts for PPM. *Proceedings of the IEEE Data Compression Conference (DCC'04)*, pp. 409–418, 2004. http://doi.ieeecomputersociety.org/10.1109/DCC.2004.1281486.

[SG+05] P. Skibiński, Sz. Grabowski, and S. Deorowicz. Revisiting dictionary-based compression. *Software — Practice & Experience*, 35 (15), pp. 1455–1476, December 2005. http://www.ii.uni.wroc.pl/~inikep/papers/05-RevisitingDictCompr.pdf.

[Sm02a] M. Smirnov. Techniques to enhance compression of texts on natural languages for lossless data compression methods. *Proceedings of V session of post-graduate students and young scientists of St. Petersburg*, State University of Aerospace Instrumentation, Saint-Petersburg, Russia, 2002. In Russian.

[Sm02b] M. Smirnov. PPMN v1.00b1+ (computer program). 2002. http://compression.ru/ms/ppmnb1+.rar.

[SP98] T. C. Smith and R. Peeters. Fast convergence with a greedy tag-phrase dictionary. *Proceedings of the IEEE Data Compression Conference (DCC'98)*, pp. 33–42, 1998.

[SS82] J. Storer and T. G. Szymanski. Data compression via textual substitution. *Journal of the ACM* 29, pp. 928–951, 1982.

[SM+03] W. Sun, A. Mukherjee, and N. Zhang. A Dictionary-based Multi-Corpora Text Compression System. *Proceedings of the IEEE Data Compression Conference (DCC'03)*, pp. 448, 2003.

[SB+02] B. Sutter, B. Bus, and K. Bosschere. Sifting out the mud: low level C++ code reuse. *ACM SIGPLAN Notices*, 37 (11), November 2002.

[Sw04] J. Swacha. Predictive-substitutive compression method. Ph.D. dissertation, Technical University of Wrocław, Wrocław, Poland, 2004. In Polish.

[TK+03] I. Tabus, G. Korodi, and J. Rissanen. DNA sequence compression using the normalized maximum likelihood model for discrete regression. *Proceedings of the IEEE Data Compression Conference (DCC'03)*, pp. 253–262, 2003.

[Ta04] M. Taylor. RKC 1.02 (computer program). March 2004. http://www.msoftware.co.nz.

[Te95] W.J. Teahan. Probability estimation for PPM. *Proceedings of the Second New Zealand Computer Science Research Students' Conference*, University of Waikato, Hamilton, New Zealand, April 1995. ftp://ftp.cs.waikato.ac.nz/pub/papers/ppm/pe_for_ppm.ps.gz.

[TC96] W. Teahan and J. Cleary. The entropy of English using PPM-based models. *Proceedings of the IEEE Data Compression Conference (DCC'96)*, pp. 53–62, 1996.

[TC97] W. Teahan and J. Cleary. Models of English text. *Proceedings of the IEEE Data Compression Conference (DCC'97)*, pp. 12–21, 1997.

[Te98] W. Teahan. Modelling English text. Ph.D. dissertation, Department of Computer Science, University of Waikato, New Zealand, 1998.

[TD+94] C. C. Todd, G. A. Davidson, M. F. Davis, L. D. Fielder, B. D. Link, and S. Vernon. "AC-3: Flexible Perceptual Coding For Audio Transmission And Storage", *Proceedings of 96th AES Convention*, preprint 3796, February 1994.

[TH02] P. M. Tolani and J. R. Haritsa. XGRIND: A Query-friendly XML Compressor. *Proceedings of 18th International Conference on Database Engineering*, pp. 225–234, February 2002.

[TS02] A. Turpin and W.F. Smyth. An approach to phrase selection for off-line data compression. *Proceedings of the twenty-fifth Australasian conference on Computer science (ACSC),* Volume 4, pp. 267–273, Melbourne, Victoria, Australia, 2002.

[VV04] B. D. Vo and K.-P. Vo. Using Column Dependency to Compress Tables. *Proceedings of the IEEE Data Compression Conference (DCC'04)*, pp. 92–101, 2004.

[VW98a] P. A. J. Volf and F. M. J. Willems. The switching method – elaborations. *Proceedings of the 19th Symposium on Information Theory in the Benelux,* pp. 13–20, 1998.

[VW98b] P. A. J. Volf and F. M. J. Willems, Switching between two universal source coding algorithms. *Proceedings of the IEEE Data Compression Conference (DCC'98),* pp. 491–500, 1998.

[Wa91] G. K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34 (4), pp. 30–44, April 1991.

[WS+00] M. J. Weinberger, G. Seroussi and G. Sapiro. The LOCO-I lossless image compression algorithm: principles and standardization into JPEG-LS. *IEEE Transactions on Image Processing*, 9 (8), pp. 1309–1324, August 2000.

[We84] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17, pp. 8–19, 1984.

[WM+99] I. H. Witten, A. Moffat, and T. C. Bell. Managing Gigabytes: Compressing and Indexing Documents and Images. Morgan Kaufmann Publishers, San Francisco, second edition, 1999.

[WA01] Wireless Application Protocol Forum, Ltd. Binary XML Content Format Specification Version 1.3. Specification WAP-192-WBXML-20010725, July 2001. http://www.openmobilealliance.org/tech/affiliates/wap/wap-192-wbxml-20010725-a.pdf.

[WJ99] B. Wohlberg and G. de Jager, A review of the fractal image coding literature. *IEEE Transactions on Image Processing*, 8 (12), pp. 1716–1729, December 1999.

[WM97] X. Wu and N.D. Memon. Context-based, adaptive, lossless image coding. *IEEE Transactions on Communications*, 45 (4), pp. 437–444, April 1997.

[YK00] E. Yang and J. C. Kieffer. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform – Part one: Without context models. *IEEE Transactions on Information Theory*, 46 (3), pp. 755–777, 2000.

[Yo03] V. Yoockin. Compressors Comparison Test 6.5. November 2003. http://compression.graphicon.ru/ybs/best.htm.

[Zi49] G. K. Zipf. Human Behaviour and the Principle of Least Effort. Addison-Wesley , Cambridge, MA, 1949.

[ZL77] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory* 23 (3), pp. 337–342, May 1977.

[ZL78] J. Ziv and A. Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory* 24 (5), pp. 530–536, September 1978.

Appendices

A The Calgary corpus

A.1 Description

The Calgary corpus [BC$^+$90] has been, for many years, the most known set of files created for testing lossless data compression algorithms. The files in the corpus were chosen to cover up the typical types of data used in the computer science. In spite of some types of data went out of use, the corpus is still a good benchmark. It consists of eighteen files; five files (*geo*, *obj1*, *obj2*, *pic* and *trans*) are non-textual and the others are textual. Please note that four files *paper3*, *paper4*, *paper5*, *paper6* are usually excluded. Descriptions of files from the Calgary corpus and their sizes are given in Table A.1.

Name	Size	Description
bib	111,261	Bibliographic files (in the Unix “refer” format)
book1	768,771	A book “Far from the Madding Crowd” by Thomas Hardy
book2	610,856	A book “Principles of Computer Speech” by Ian Witten
geo	102,400	Geophysical data of seismic activity
news	377,109	Postings from various newsgroups on USENET
obj1	21,504	VAX executable of program “progp”
obj2	246,814	Macintosh executable of “Knowledge support system”
paper1	53,161	A paper “Arithmetic coding for data compression” by Ian Witten, Radford Neal, and John Cleary
paper2	82,199	A paper “Computer (in)security” by Ian Witten
paper3	46,526	A paper “In search of autonomy” by Ian Witten
paper4	13,286	A paper “Programming by example revisited” by John Cleary
paper5	11,954	A paper “A logical implementation of arithmetic” by John Cleary
paper6	38,105	A paper “Compact hash tables using bidirectional linear probing” by John Cleary
pic	513,216	Picture number 5 from the CCITT Facsimile test files (text + drawings)
progc	39,611	C source code of Unix compress version 4.0
progl	71,646	LISP source code
progp	49,379	Pascal source code of Prediction by Partial Matching evaluation program
trans	93,695	Transcript of a session on a EMACS terminal

Table A.1: An overview of the files in the Calgary corpus.

A.2 History of the best compression results on the Calgary corpus

Table A.2 shows a history of the best compression results on the Calgary corpus. The first well-known compressor was pack, which implements the Huffman coding. It was easily beaten by compress, based on the LZW algorithm. PPMC crossed border of 3.0 bpc of an average compression effectiveness on the Calgary corpus. Continuously improved PPM-based algorithms have had the best compression effectiveness until the year 2002. Currently, the best compression effectiveness is achieved with PAQ-based compressors, which can be considered as binary-based PPM.

File	pack[3]	compress[4] 1985	PPMC[5] 1988	PPMD[6] 1993	PPMD+[7] 1995	PPM-FSMX[8] 1997	CTW+PPM*[9] 1998	cPPMII[10] 2002	PAQ[11] 2003	PAQAR[12] 2004
bib	5.23	3.35	2.11	1.88	1.862	1.786	1.714	1.663	1.617	1.528
book1	4.56	3.46	2.48	2.31	2.303	2.184	2.150	2.119	2.090	2.031
book2	4.83	3.28	2.26	1.97	1.963	1.862	1.820	1.742	1.691	1.604
geo	5.69	6.08	4.78	4.74	4.733	4.458	4.526	3.869	3.536	3.467
news	5.23	3.86	2.65	2.38	2.355	2.285	2.210	2.084	2.034	1.925
obj1	6.08	5.23	3.76	3.74	3.728	3.678	3.607	3.345	3.047	2.841
obj2	6.30	4.17	2.69	2.42	2.378	2.283	2.245	1.898	1.703	1.477
paper1	5.03	3.77	2.48	2.34	2.330	2.250	2.152	2.122	2.052	1.973
paper2	4.65	3.52	2.45	2.32	2.315	2.213	2.136	2.112	2.056	1.995
pic	1.66	0.97	1.09	0.80	0.795	0.781	0.764	0.693	0.456	0.372
progc	5.26	3.87	2.49	2.38	2.363	2.291	2.195	2.106	2.031	1.945
progl	4.81	3.03	1.90	1.70	1.677	1.545	1.482	1.352	1.314	1.223
progp	4.91	3.11	1.84	1.71	1.696	1.531	1.460	1.360	1.312	1.200
trans	5.58	3.27	1.77	1.50	1.467	1.352	1.256	1.151	1.126	1.038
average	4.99	3.64	2.48	2.30	2.283	2.177	2.123	1.972	1.861	1.758

Table A.2: History of the best compression results on the Calgary corpus.
The results are in bits per character.

[3] pack (Huffman) — results taken from Reference [AL00]
[4] compress (LZW) — results taken from Reference [AL00]
[5] PPMC — results taken from Reference [BW89]
[6] PPMD — results taken from the author’s implementation of PPMD based on [Ho93]
[7] PPMD+ — results taken from Reference [Bl96]
[8] PPM-FSMX — results taken from Reference [Bu97a]
[9] CTW+PPM* — results taken from Reference [VW98b]
[10] cPPMII — experimental results taken by the author from: PPMonstr var. I [Sh02b]
[11] PAQ — experimental results taken by the author from: PAQ6 v2 [Ma03]
[12] PAQAR — experimental results taken by the author from: PAQAR 4.0 [MR04]

B The Canterbury corpus and the large Canterbury corpus

Arnold and Bell proposed [AB97] a replacement for the Calgary corpus. The authors reviewed the types of data used contemporarily, performed statistical research, and selected eleven files (from about 800) that are characteristic for their types.

To eliminate a disadvantage of the absence of large files in the Canterbury corpus, Arnold and Bell created another set of files, called the large Canterbury corpus. This corpus consists of three large text files: *bible.txt*, *E.coli*, and *world192.txt*.

Descriptions of files from the Canterbury corpus and the large Canterbury corpus and also their sizes are given in Table A.3.

Name	Size	Description
alice29.txt	152,089	A book “Alice’s Adventures in Wonderland” by Lewis Carroll
asyoulik.txt	125,179	A play “As you like it” by William Shakespeare
bible.txt	4,047,392	The King James version of the Bible
cp.html	24,603	Compression pointers
E.coli	4,638,690	Complete genome of the Esxherichia coli bacterium
fileds.c	11,150	C source code
grammar.lsp	3,721	LISP source code
kennedy.xls	1,029,774	Excel spreadsheet
lcet10.txt	426,754	Proceedings from “Workshop on electronic texts”
plrabn12.txt	481,861	A book “Paradise Lost” by John Milton
ptt5	513,216	Picture number 5 from the CCITT Facsimile test files (text + drawings)
sum	38,240	SPARC executable
world192.txt	2,473,400	The CIA world factbook
xargs.1	4,227	GNU manual page of xargs

Table A.3: An overview of the files in the Canterbury corpus and the large Canterbury corpus.

C The multilingual corpus

There is no well-known corpus with multilingual text files. Therefore, we have selected multilingual text files from Project Gutenberg [Ha04]. The multilingual corpus consist of English, German, French, Polish, and Russian the files. English files use the Latin alphabet encoded as 7-bit ASCII codes. German, French, and Polish files use the Latin alphabet with additional national characters like, for example, 'ąćęłóśżź'. Russian files do not use the Latin alphabet at all, except English comments and proper nouns. Descriptions of files from the multilingual corpus and also their sizes are given in Table A.4.

Name	Size	Language	Description
10055-8.txt[13]	1,014,352	German	"Hamburgische Dramaturgie" by Gotthold Ephraim Lessing
12267-8.txt[14]	474,834	German	"Aus meinem Leben, Erster Teil" by August Bebel
1musk10.txt[15]	1,344,739	English	"The Three Musketeers" by Alexandre Dumas (Pere)
plrabn12.txt[16]	481,861	English	"Paradise Lost" by John Milton
11176-8.txt[17]	831,352	French	"Memoires du sergent Bourgogne" by Adrien-Jean-Baptiste-Francois Bourgogne
11645-8.txt[18]	447,885	French	"La rotisserie de la Reine Pedauque" by Anatole France
rnpz810.txt[19]	571,718	Polish	"Ironia Pozorow" by Maciej hr. Lubienski
sklep10.txt[20]	187,180	Polish	"Sklepy cynamonowe" by Bruno Schulz
master.txt[21]	811,213	Russian	"The Master and Margarita" by Mikhail Bulgakov
misteria.txt[22]	473,434	Russian	"O egipetskih misteriyah" by Yamvlih Halkidskij

Table A.4: An overview of the files in the multilingual corpus.

[13] http://www.gutenberg.net/1/0/0/5/10055/10055-8.txt
[14] http://www.gutenberg.net/1/2/2/6/12267/12267-8.txt
[15] http://www.gutenberg.net/etext98/1musk10.txt
[16] http://www.gutenberg.net/etext92/plrabn12.txt
[17] http://www.gutenberg.net/1/1/1/7/11176/11176-8.txt
[18] http://www.gutenberg.net/1/1/6/4/11645/11645-8.txt
[19] http://www.gutenberg.net/etext04/rnpz810.txt
[20] http://www.gutenberg.net/etext05/sklep10.txt
[21] http://www.lib.ru/BULGAKOW/master.txt
[22] http://www.lib.ru/URIKOVA/misteria.txt

D PPMVC program usage

PPMVC 1.0 computer program is publicly available at Reference [Sk03] and on CD included to this dissertation.

PPMVC compression

Usage: “PPMVC.exe e [options] <FileName[s] | Wildcard[s]>”.
Options:

```
    -oN = set model order to N - [2,32], default: 16
    -mN = use N MB memory - [1,256], default: 8
    -d = delete file[s] after processing, default: disabled
    -y = delete archive if already exists, default: disabled
    -fName = set output file name to Name
    -a = log results to file ppmvc.log
    -rN = set method of model restoration at memory insufficiency:
       -r0 = restart model from scratch (default)
       -r1 = cut off model (slow)
       -r2 = freeze model (dangerous)
```

Options (for VC testing only, should be specified for encoding and decoding):

```
    -g = group size [1,256], default value depends on order
    -h = min left (before) match lenght [1,10240], default value depends on order
    -i = max left (before) match lenght [1,10240], default value depends on order
    -j = min right (after) match lenght [1,10240], default value depends on order
    -l = check [1,64] last pointers, default value depends on order
    -b = ignore min left match len (faster compression)
    -u = turn off variable-length contexts (ordinary PPMII compression)
```

PPMVC decompression

Usage: “PPMVC.exe d <FileName[s] | Wildcard[s]>”.

No additional options are required.

PPMVC1, PPMVC2 and PPMVC3

PPMVC1 is equal to “PPMVC.exe –l1 –b”
PPMVC2 is equal to “PPMVC.exe –l1”
PPMVC3 is equal to “PPMVC.exe” (default)

E TWRT program usage

TWRT (WRT 4.4) computer program is publicly available at Reference [Sk05a] and on CD included to this dissertation.

TWRT compression

Usage: “WRT.exe [options] <input_file> <output_file>”.

Options:
The are general options (which also set default additional options):
–0 = preprocessing optimized for further LZ-based compression
–1 = preprocessing optimized for further BWT-based compression
–2 = preprocessing optimized for further PPM-based compression (default)
–3 = preprocessing optimized for further PAQ-based compression
and additional options:
–b= Turn off the mode for non-textual data (ignore auto detection)
+b = Turn on the mode for non-textual data (ignore auto detection)
–d = Turn off usage of the dictionary (word-based preprocessing)
–e = Turn off the EOL-coding (ignore auto detection)
+e = Turn on the EOL-coding (ignore auto detection)
–n = Turn off the *q*-gram replacement
–p = Turn off the surrounding words with spaces (ignore auto detection)
+p = Turn on the surrounding words with spaces (ignore auto detection)
–q = Turn off the DNA sequence preprocessing (ignore auto detection)
–r = Turn off the fixed-length record aligned data preprocessing option (ignore auto detection)
–t = Turn off the matching shorter words option

Example:
“WRT.exe –0 –e book2 book2.wrt” chooses the preprocessing optimized for further LZ-based compression and turns off the EOL-coding.

TWRT decompression

Usage: “WRT.exe <input_file> <output_file>”.

No additional options are required.

F Switches of examined compression programs used during the experiments

7-Zip 4.17 beta

The default options are used.

bzip2 1.0.2

The default options are used.

DURILCA 0.4b

The PPM order is set to 128 with the option –o128. The memory limit is set to 230 MB, with the option –m230. Built-in models are switched on with the option –t2.

GRZipII 0.2.4

The default options (–m1 –b5m) are used.

gzip 1.2.4

The default options are used.

PAQAR 4.0

The memory limit is set to 230 MB, with the default option –6.

PPMVC 1.0

The PPM order is set to 12 with the option –o12. The memory limit is set to 230 MB, with the option –m230.

RKC 1.02

The PPM order is set to 128 with the option –o128. The memory limit is set to 230 MB, with the option –M230m. The dictionary (word-based preprocessing) is switched off with the option –a–.

UHARC 0.6

The default options (–m2 –b1024) are used.

UHBC 1.0

The default options (–m2 –b5m) are used.

G Contents of CD included to this dissertation

File	Description
Dissertation.pdf	This dissertation in PDF format
PPMVC10.zip	PPMVC 1.0 computer program with source code (July 2003)
WRT30d.zip	WRT computer program with source code (April 2004)
WRT-LZ77.zip	WRT-LZ77 computer program with source code (2004)
WRT44.zip	TWRT computer program with source code (March 2005)

Table A.5: Contents of CD included to this dissertation.