Silesian University of Technology
Faculty of Automatic Control, Electronics
and Computer Science
Institute of Computer Science

Doctor of Philosophy Dissertation

Universal lossless data compression algorithms

Sebastian Deorowicz

Supervisor: Prof. dr hab. inż. Zbigniew J. Czech

Gliwice, 2003

To My Parents

Contents

Chapter 1

Preface

I am now going to begin my story
(said the old man), so please attend.

— ANDREW LANG
The Arabian Nights Entertainments (1898)

Contemporary computers process and store huge amounts of data. Some parts of these data are excessive. *Data compression* is a process that reduces the data size, removing the excessive information. Why is a shorter data sequence often more suitable? The answer is simple: it reduces the costs. A full-length movie of high quality could occupy a vast part of a hard disk. The compressed movie can be stored on a single CD-ROM. Large amounts of data are transmitted by telecommunication satellites. Without compression we would have to launch many more satellites that we do to transmit the same number of television programs. The capacity of Internet links is also limited and several methods reduce the immense amount of transmitted data. Some of them, as mirror or proxy servers, are solutions that minimise a number of transmissions on long distances. The other methods reduce the size of data by compressing them. Multimedia is a field in which data of vast sizes are processed. The sizes of text documents and application files also grow rapidly. Another type of data for which compression is useful are database tables. Nowadays, the amount of information stored in databases grows fast, while their contents often exhibit much redundancy.

Data compression methods can be classified in several ways. One of the most important criteria of classification is whether the compression algorithm removes some parts of data which cannot be recovered during the decompression. The algorithms removing irreversibly some parts of data are called *lossy*, while others are called *lossless*. The lossy algorithms are usually used when a perfect consistency with the original data is not necessary after the decompression. Such a situation occurs for example in compression of video or picture data. If the recipient of the video is a human, then small changes of colors of some pixels introduced during the compression could be imperceptible. The lossy compression methods typically yield much better compression ratios than lossless algorithms, so if we can accept some distortions of data, these methods can be used. There are, however, situations in which the lossy methods must not be used to compress picture data. In many countries, the medical images can be compressed only by the lossless algorithms, because of the law regulations.

One of the main strategies in developing compression methods is to prepare a specialised compression algorithm for the data we are going to transmit or store. One of many examples where this way is useful comes from astronomy. The distance to a spacecraft which explores the universe is huge, what causes big communication problems. A critical situation took place during the Jupiter mission of Galileo spacecraft. After two years of flight, the Galileo's high-gain antenna did not open. There was a way to get the collected data through a supporting antenna, but the data transmission speed through it was slow. The supporting antenna was designed to work with a speed of 16 bits per second at the Jupiter distance. The Galileo team improved this speed to 120 bits per second, but the transmission time was still quite long. Another way to improve the transmission speed was to apply highly efficient compression algorithm. The compression algorithm that works at Galileo spacecraft reduces the data size about 10 times before sending. The data have been still transmitted since 1995. Let us imagine the situation without compression. To receive the same amount of data we would have to wait about 80 years.

The situation described above is of course specific, because we have here good knowledge of what kind of information is transmitted, reducing the size of the data is crucial, and the cost of developing a compression method is of lower importance. In general, however, it is not possible to prepare a specialised compression method for each type of data. The main reasons are: it would result in a vast number of algorithms and the cost of developing a new compression method could surpass the gain obtained by the reduction of the data size. On the other hand, we can assume nothing about the data. If we do so, we have no way of finding the excessive information. Thus a compromise is needed. The standard approach in compression is to define the *classes of sources* producing different types of data. We assume that the data are produced by a source of some class and apply a compression method designed for this particular class. The algorithms working well on the data that can be approximated as an output of some general source class are called *universal*.

Before we turn to the families of universal lossless data compression algorithms, we have to mention the entropy coders. An *entropy coder* is a method that assigns to every symbol from the alphabet a code depending on the probability of symbol occurrence. The symbols that are more probable to occur get shorter codes than the less probable ones. The codes are assigned to the symbols in such a way that the expected length of the compressed sequence is minimal. The most popular entropy coders are *Huffman coder* and an *arithmetic coder*. Both the methods are optimal, so one cannot assign codes for which the expected compressed sequence length would be shorter. The Huffman coder is optimal in the class of methods that assign codes of integer length, while the arithmetic coder is free from this limitation. Therefore it usually leads to shorter expected code length.

A number of universal lossless data compression algorithms were proposed. Nowadays they are widely used. Historically the first ones were introduced by Ziv and Lempel [202, 203] in 1977–78. The authors propose to search the data to compress for identical parts and to replace the repetitions with the information where the identical subsequences appeared before. This task can be accomplished in several ways. Ziv and Lempel proposed two main variants of their method: LZ77 [202], which encodes the information of repetitions directly, and LZ78 [203], which maintains a supporting dictionary of subsequences appeared so far, and stores the indexes from this dictionary in the output sequence. The main advantages of these methods are: high speed and ease of implementation. Their compression ratio is, however, worse than the ratios obtained by other contemporary methods.

A few years later, in 1984, Cleary and Witten introduced a *prediction by partial matching* (PPM) algorithm [48]. It works in a different way from Ziv and Lempel methods. This algorithm calculates the statistics of symbol occurrences in contexts which appeared before. Then it uses them to assign codes to the symbols from the alphabet that can occur at the next position in such a way that the expected length of the sequence is minimised. It means that the symbols which are more likely to occur have shorter codes than the less probably ones. The statistics of symbol occurrences are stored for separate contexts, so after processing one symbol, the codes assigned for symbols usually differ completely because of the context change. To assign codes for symbols an arithmetic coder is used. The main disadvantages of the PPM algorithms are slow running and large memory needed to store the statistics of symbol occurrences. At present, these methods obtain the best compression ratios in the group of universal lossless data compression algorithms. Their low speed of execution limits, however, their usage in practice.

Another statistical compression method, a *dynamic Markov coder* (DMC), was invented by Cormack and Horspool [52] in 1987. Their algorithm assumes that the data to compress are an output of some *Markov source* class, and during the compression, it tries to discover this source by better and better estimating the probability of occurrence of the next symbol. Using this probability, the codes for symbols from the alphabet are assigned by making use of an arithmetic coder. This algorithm also needs a lot of memory to store the statistics of symbol occurrences and runs rather slowly. After its formulation, the DMC algorithm seemed as an interesting alternative for the PPM methods, because it led to the comparable compression ratios with similar speed of running. In last years, there were significant improvements in the field of the PPM methods, while the research on the DMC algorithms stagnated. Nowadays, the best DMC algorithms obtain significantly worse compression ratios than the PPM ones, and are slower.

In 1995, an interesting compression method, a *context tree weighting* (CTW) algorithm, was proposed by Willems *et al.* [189]. The authors introduced a concept of a *context tree source* class. In their compression algorithm, it is assumed that the data are produced by some source of that class, and relating on this assumption the probability of symbol occurrence is estimated. Similarly to the PPM and the DMC methods, an arithmetic coder is used to assign codes to the symbols during the compression. This method is newer than the before-mentioned and a relatively small number of works in this field have appeared so far. The main advantage of this algorithm is its high compression ratio, only slightly worse than those obtained by the PPM algorithms. The main disadvantage is a low speed of execution.

Another compression method proposed recently is a *block-sorting compression algorithm*, called usually a *Burrows–Wheeler compression algorithm* (BWCA) [39]. The authors invented this method in 1994. The main concept of their algorithm is to build a matrix, whose rows store all the one-character cyclic shifts of the compressed sequence, to sort the rows lexicographically, and to use the last column of this matrix for further processing. This process is known as the *Burrows–Wheeler transform* (BWT). The output of the transform is then handled by a *move-to-front* transform [26], and, in the last stage, compressed by an entropy coder, which can be a Huffman coder or an arithmetic coder. As the result of the BWT, a sequence is obtained, in which all symbols appearing in similar contexts are grouped. The important features of the BWT-based methods are their high speed of running and reasonable compression ratios, which are much better than those for the LZ methods and only slightly worse than for the best existing PPM algorithms. The algorithms based on the transform by Burrows and Wheeler seem to be an interesting alternative to fast, Ziv–Lempel methods, which give comparatively poor compression ratios, and the PPM algorithms which obtain the better compression ratios, but work slowly.

In this dissertation the attention is focused on the BWT-based compression algorithms. We investigate the properties of this transform and propose an improved compression method based on it.

Dissertation thesis: An improved algorithm based on the Burrows–Wheeler transform we propose, achieves the best compression ratio among the BWT-based algorithms, while its speed of operation is comparable to the fastest algorithms of this family.

In the dissertation, we investigate all stages of the BWT-based algorithms, introducing new solutions at every phase. First, we analyse the structure of the Burrows–Wheeler transform output, proving some results, and showing that the output of the examined transform can be approximated by the output of the piecewise stationary memoryless source. We investigate also methods for the BWT computation, demonstrating that a significant improvement to the Itoh–Tanaka's method [90] is possible. The improved BWT computation method is fast and is used in further research. Using the investigation results of the BWT output we introduce a *weighted frequency count* (WFC) transform. We examine several proposed *weight functions*, being the important part of the transform, on the piecewise stationary memoryless source output, performing also numerical analysis. The WFC transform is then proposed as the second stage of the improved BWT-based algorithm (the replacement of the originally used MTF transform). In the last stage of the compression method, the output sequence of previous phases is compressed by the arithmetic coder. The most important problem at this stage is to estimate the probabilities of symbol occurrences which are then used by the entropy coder. An original method, a *weighted probability*, is proposed for this task. Some of the results of this dissertation were published by the author in References [54, 55].

The properties of the proposed compression method are examined on the real-world data. There are three well known data sets, used by researchers in the field of compression: the *Calgary corpus* [20], the *Canterbury corpus* [12], and the *large Canterbury corpus* [98]. The first one, proposed in 1989, is rather old, but it is still used by many researchers. The later corpora are more recent, but they are not so popular and, as we discuss in the dissertation, they are not good candidates as the standard data sets. In the dissertation, we discuss also a need of examining the compression methods on files of sizes significantly larger than the existing ones in the three corpora. As the result, we propose a *Silesia corpus* to perform tests of the compression methods on files of sizes and contents which are used nowadays.

The dissertation begins, in Chapter 2, with the formulation of the data compression problem. Then we define some terms needed to discuss compression precisely. In this chapter, a modern paradigm of data compression, *modelling and coding*, is described. Then, the entropy coding methods, such as Huffman and arithmetic coding are presented. The families of universal lossless compression algorithms that are used nowadays: Ziv–Lempel algorithms, prediction by partial matching algorithms, dynamic Markov coder algorithms, and context tree weighting algorithms are also described in detail. Chapter 2 ends with a short review of some specialised compression methods. Such methods are useful when we know something more about the data to compress. One of these methods was discussed by Ciura and the author of this dissertation in Reference [47]. In Chapter 3, we focus our attention on the family of compression algorithms based on the Burrows–Wheeler transform. In the first part of this chapter, we describe the original BWCA in detail. Then, we present the results of the investigations on this algorithm, which were made by other researches.

Chapter 4, containing the main contribution of the dissertation, starts from the description of the proposed solutions. Then we discuss the methods of comparing data compression algorithms. To this end, we describe three existing corpora for examining universal lossless data compression algorithms. The arguments for and against usage of the Calgary, the Canterbury, and the large Canterbury corpora are presented. We also argue for a need of existence of a corpus containing larger and more representable files to the ones used contemporarily. Therefore, we introduce a Silesia corpus. Then, we discuss *multi criteria optimisation*. This is done because the compression process cannot be treated as a simple optimisation problem in which only one criterion is optimised, say for example the compression ratio. Such criteria as compression and decompression speeds are also important. In the end of this chapter, we describe the experiments and comment the obtained results.

Chapter 5 contains a discussion of obtained results and justification of the dissertation thesis. The dissertation ends with appendices, in which some details are presented. Appendix A contains a more precise specification of files in the Silesia corpus. Appendix B contains some technical information of the implementation. We briefly discuss here the contents of the source code files, employed techniques, and the usage of the compression program implementing the proposed compression algorithm. The detailed information of the compression programs used for the comparison can be found in Appendix C. In Appendix D, some more detailed graphs obtained during the investigation of the probability estimation method in sequences produced by the piecewise stationary memoryless sources are presented. Appendix E contains the auxiliary results of the performed experiments.

Chapter 2

Introduction to data compression

'It needs compression,'
I suggested, cautiously.

— RUDYARD KIPLING
The Phantom 'Rickshaw
and Other Ghost Stories (1899)

2.1 Preliminaries

Computers process miscellaneous data. Some data, as colours, tunes, smells, pictures, voices, are analogue. Contemporary computers do not work with infinite-precise analogue values, so we have to convert such data to a digital form. During the digitalisation process, the infinite number of values is reduced to a finite number of quantised values. Therefore some information is always lost, but the larger the target set of values, the less information is lost. Often the precision of digitalisation is good enough to allow us neglecting the difference between digital version of data and their analogue original.

There are also discrete data, for example written texts or databases contain data composed of finite number of possible values. We do not need to digitise such types of data but only to represent them somehow by encoding the original values. In this case, no information is lost.

Regardless of the way we gather data to computers, they usually are sequences of elements. The elements come from a finite ordered set, called an *alphabet*. The elements of the alphabet, representing all possible values, are called *symbols* or *characters*. One of the properties of a given alphabet is its number of symbols, and we call this number the *size of the alphabet*. The *size of a sequence* is the number of symbols it is composed of.

The size of the alphabet can differ for various types of data. For a Boolean sequence the alphabet consists of only two symbols: *false* and *true*, representable on 1 bit only. For typical English texts the alphabet contains less than 128 symbols and each symbol is represented on 7 bits using the ASCII code. The most popular code in contemporary computers is an 8-bit code; some texts are stored using the 16-bit Unicode designed to represent all the alphabetic symbols used worldwide. Sound data typically are sequences of symbols, which represent temporary values of the tone. The size of the alphabet to encode this data is usually 2^8, 2^{16}, or 2^{24}. Picture data typically contain symbols from the alphabet representing the colours of image pixels. The colour of a pixel can be represented using various coding schemes. We mention here only one of them, the RGB code that contains the brightness of the three components red, green, and blue. The brightness of each component can be represented, for example, using 2^8 different values, so the size of the alphabet is 2^{24} in this case.

A sequence of symbols can be stored in a file or transmitted over a network. The sizes of modern databases, application files, or multimedia files can be extremely large. Reduction of the sequence size can save computing resources or reduce the transmission time. Sometimes we even would not be able to store the sequence without compression. Therefore the investigation of possibilities of compressing the sequences is very important.

2.2 What is data compression?

A sequence over some alphabet usually exhibits some regularities, what is necessary to think of compression. For typical English texts we can spot that the most frequent letters are e, t, a, and the least frequent letters are q, z. We can also find such words as the, of, to frequently. Often also longer fragments of the text repeat, possibly even the whole sentences. We can use these properties in some way, and the following sections elaborate this topic.

A different strategy to compress the sequence of picture data is needed. With a photo of night sky we can still expect that the most frequent colour of pixels is black, or dark grey. But with a generic photo we usually have no information what colour is the most frequent. In general, we have no *a priori* knowledge of the picture, but we can find regularities in it. For example, colours of successive pixels usually are similar, some parts of the picture are repeated.

Video data are typically composed of subsequences containing the data of the successive frames. We can simply treat the frames as pictures and compress them separately, but more can be achieved with analysing the consecutive frames. What can happen in a video during a small fraction of a second? Usually not much. We can assume that successive video frames are often similar.

We have noticed above that regularities and similarities often occur in the sequences we want to compress. Data compression bases on such observations and attempts to utilise them to reduce the sequence size. For different types of data there are different types of regularities and similarities, and before we start to compress a sequence, we should know of what type it is. One more thing we should mark here is that the compressed sequence is useful only for storing or transmitting, but not for a direct usage. Before we can work on our data we need to expand them to the original form. Therefore the compression methods must be reversible. The decompression is closely related to the compression, but the latter is more interesting because we have to find the regularities in the sequence. Since during the decompression nothing new is introduced, it will not be discussed here in detail.

A detailed description of the compression methods described in this chapter can be found also in one of several good books on data compression by Bell *et al.* [22], Moffat and Turpin [116], Nelson and Gailly [118], Sayood [143], Skarbek [157], or Witten *et al.* [198].

2.3 Lossy and lossless compression

2.3.1 Lossy compression

The assumed recipient of the compressed data influences the choice of a compression method. When we compress audio data some tones are not audible to a human because our senses are imperfect. When a human will be the only recipient, we can freely remove such unnecessary data. Note that after the decompression we do not obtain the original audio data, but the data that sound identically. Sometimes we can also accept some small distortions if it entails a significant improvement to the compression ratio. It usually happens when we have a dilemma: we can have a little distorted audio, or we can have no audio at all because of data storage restrictions. When we want to compress picture or video data, we have the same choice—we can sacrifice the perfect conformity to the original data gaining a tighter compression. Such compression methods are called *lossy*, and the strictly bi-directional ones are called *lossless*.

The lossy compression methods can achieve much better compression ratio than lossless ones. It is the most important reason for using them. The gap between compression results for video and audio data is so big that lossless methods are almost never employed for them. Lossy compression methods are also employed to pictures. The gap for such data is also big but there are situations when we cannot use lossy methods. Sometimes we cannot use lossy methods to the images because of the law regulations. This occurs for medical images as in many countries they must not be compressed loosely.

Roughly, we can say that lossy compression methods may be used to data that were digitised before compression. During the digitalisation process small distortions are introduced and during the compression we can only increase them.

2.3.2 Lossless compression

When we need certainty that we achieve the same what we compressed after decompression, lossless compression methods are the only choice. They are of course necessary for binary data or texts (imagine an algorithm that could change same letters or words). It is also sometimes better to use lossless compression for images with a small number of different colours or for scanned text.

The rough answer to the question when to use lossless data compression methods is: We use them for digital data, or when we cannot apply lossy methods for some reasons.

This dissertation deals with lossless data compression, and we will not concern lossy compression methods further. From time to time we can mention them but it will be strictly denoted. If not stated otherwise, further discussion concerns lossless data compression only.

2.4 Definitions

For precise discussion, we introduce now some terms. Some of them were used before but here we provide their precise definitions.

Let us assume that $x = x_1x_2\ldots x_n$ is a *sequence*. The sequence *length* or *size* denoted by n is a number of elements in x, and x_i denotes the ith element of x. We also define the *reverse sequence*, x^{-1}, as $x_nx_{n-1}\ldots x_1$. Given a sequence x let us assume that $x = uvw$ for some, possibly empty, subsequences u, v, w. Then u is called a *prefix* of x, v a *component* of x, and w a *suffix* of x. Each element of the sequence, x_i, belongs to a finite ordered set $\mathcal{A} = \{a_0, a_1, \ldots, a_{k-1}\}$ that is called an *alphabet*. The number of elements in $\mathcal{A}$ is the *size of the alphabet* and is denoted by k. The elements of the alphabet are called *symbols* or *characters*. We introduce also a special term for a non-empty component of x that consists of identical symbols that we call a *run*. To simplify the notation we denote the component $x_ix_{i+1}\ldots x_j$ by $x_{i..j}$. Except the first one, all characters x_i in a sequence x are preceded by a nonempty prefix $x_{(i-d)..(i-1)}$. We name this prefix a *context of order d* of the x_i. If the order of the context is unspecified we arrive at the longest possible context $x_{1..(i-1)}$ that we call simply a *context* of x_i. It will be useful for clear presentation to use also terms such as *past*, *current*, or *future*, related to time when we talk about the positions in a sequence relative to the current one.

2.5 Modelling and coding

2.5.1 Modern paradigm of data compression

The modern paradigm of compression splits it into two stages: modelling and coding. First, we recognise the sequence, look for regularities and similarities. This is done in the *modelling* stage. The modelling method is specialised for the type of data we compress. It is obvious that in video data we will be searching for different similarities than in text data. The modelling methods are often different for lossless and lossy methods. Choosing the proper modelling method is important because the more regularities we find the more we can reduce the sequence length. In particular, we cannot reduce the sequence length at all if we do not know what is redundant in it.

The second stage, *coding*, is based on the knowledge obtained in the modelling stage, and removes the redundant data. The coding methods are not so diverse because the modelling process is the stage where the adaptation to the data is made. Therefore we only have to encode the sequence efficiently removing known redundancy.

Some older compression methods, such as Ziv–Lempel algorithms (see Section 2.7.2), cannot be precisely classified as representatives of modelling-coding paradigm. They are also still present in contemporary practical solutions, but their importance seems to be decreasing. We consider them to have a better view of the background but we pay our attention to the modern algorithms.

2.5.2 Modelling

The modelling stage builds a model representing the sequence to compress, the *input sequence*, and predicts the future symbols in the sequence. Here we estimate a probability distribution of occurrences of symbols.

The simplest way of modelling is to use a precalculated table of probabilities of symbol occurrences. The better is our knowledge of symbol occurrences in the current sequence, the better we can predict the future characters. We can use precalculated tables if we know exactly what we compress. If we know that the input sequence is an English text, we can use typical frequencies of character occurrences. If, however, we do not know the language of the text, and we use, e.g., the precalculated table for the English language to a Polish text, we can achieve much worse results, because the difference between the input frequencies and the precalculated ones is big. The more so, the Polish language uses letters such as ó, ś that do not exist in English. Therefore the frequencies table contains a frequency equal to zero for such symbols. It is a big problem and the compressor may not work when such extraordinary symbols appear. Furthermore, the probability of symbol occurrences differs for texts of various authors. Probably the most astonishing example of this discrepancy is a two hundred pages French novel *La Disparition* by Georges Perec [128] and its English translation *A Void* by Gilbert Adair [129], both not using the letter e at all!

Therefore a better way is not to assume too much about the input sequence and build the model from the encoded part of the sequence. During the decompression process the decoder can build its own model in the same way. Such an approach to the compression is called *adaptive* because the model is built only from past symbols and adapts to the contents of the sequence. We could not use the future symbols because they are unknown to the decompressor. Other, *static*, methods build the model from the whole sequence before the compression and then use it. The decompressor has no knowledge of the input sequence, and the model has to be stored in the output sequence, too. The static approach went almost out of use because is can be proved that the adaptive way is equivalent, so we will not be considering it further.

2.5.3 Entropy coding

Entropy

The second part of the compression is typically the entropy coding. The methods of entropy coding are based on a probability distribution of occurrences of the alphabet symbols, which is prepared by the modelling stage, and then compress these characters. When we know the probability of occurrences of every symbol from the alphabet, but we do not know the current character, the best what we can do is to assign to each character a code of length

$$\log \frac{1}{p_i} = -\log p_i \tag{2.1}$$

where p_i is the probability of occurrence of symbol a_i [151]. (All logarithms in the dissertation are to the base 2.) If we do so, the expected code length of the current symbol is

$$E(.) = -\sum_{i=0}^{k-1} p_i \cdot \log p_i. \tag{2.2}$$

The difference between these codes and the ones used for representing the symbols in the input sequence is that here the codes have different length, while such codes as ASCII, Unicode, or RGB store all symbols using the identical number of bits.

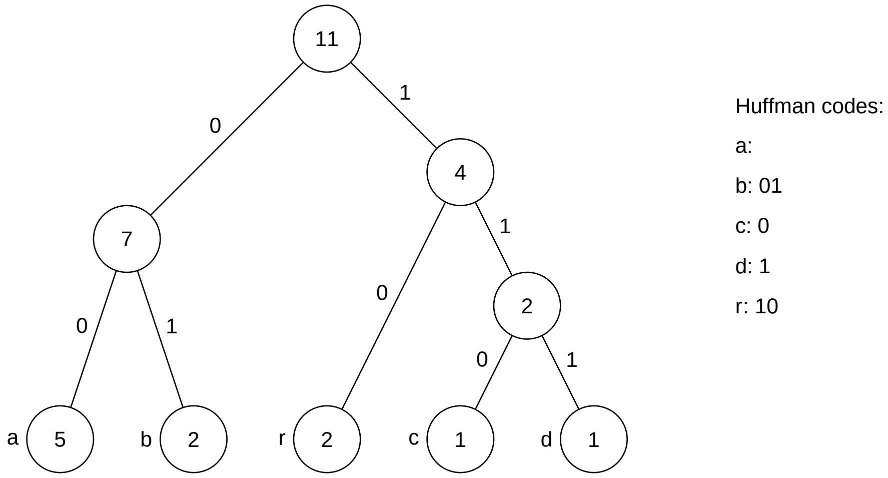

Figure 2.1: Example of the Huffman tree for the sequence abracadabra

The expected code length of the current symbol is not greater than the code length of this symbol. Replacing all characters from the input sequence with codes of smaller expected length causes a reduction of the total sequence length and gives us compression. Note that if the modelling stage produces inadequate estimated probabilities, the sequence can expand during the compression. Here we can also spot why a data compression algorithm will not be working on the Polish text with the English frequency table. Let us assume that we have to compress the Polish letter ś for which the expected probability is 0. Using the rule of the best code length (Equation 2.1) the coder would generate an infinite-length code, what is impossible.

Huffman coding

Expression 2.1 means that we usually should assign codes of noninteger length to most symbols from the input sequence. It is possible, but let us first take a look on a method giving the best expected code length among the methods which use codes of integer length only. This coding procedure was introduced by Huffman [87] in 1952.

Let us assume that we have a table of frequencies of symbol occurrences in the encoded part of the sequence (this is what the simple modelling method can do). Now we start building a tree by creating the leaves, one leaf for each symbol from the alphabet. Then we create a common parent for the two nodes without parents and with the smallest frequency. We assign to the new node a frequency being a sum of the frequencies of its sons. This process is repeated until there is only one node without a parent. We call this node a *root* of the tree. Then we create the code for a given symbol, starting from the root and moving towards the leaf corresponding to the symbol. We start with an empty code, and whenever we go to a left son, we append 0 to it, whenever we go to a right son, we append 1. When we arrive to the leaf, the code for the symbol is ready. This procedure is repeated for all symbols. Figure 2.1 shows an example of the Huffman tree and the codes for symbols after processing a sequence abracadabra. There is more than one possible Huffman tree for our data, because if there are two nodes with the same frequency we can choose any of them.

The Huffman coding is simple, even though rebuilding the tree after processing each character is quite complicated. It was shown by Gallager [73] that its maximum inefficiency, i.e., the maximum difference between the expected code length and the optimum (Equation 2.2) is bounded by

$$p_m + \log \frac{2 \log e}{e} \approx p_m + 0.086, \tag{2.3}$$

where p_m is the probability of occurrence of the most frequent symbol. Typically the loss is smaller and, owing to its simplicity and effectiveness, this algorithm is often used when compression speed is important.

The Huffman coding was intensively investigated during the years. Some of the interesting works were provided by Faller [62], Knuth [93], Cormack and Horspool [51], and Vitter [176, 177]. These works contain description of methods of storing and maintaining the Huffman tree.

Arithmetic coding

The reason of the inefficiency of the Huffman coding results from using the codes of integer length only. If we get rid of this constraint, we can be close to the optimum. The arithmetic coding method offers such a solution. Similarly to the Huffman coding, we need a table of frequencies of all symbols from the alphabet. At the beginning of coding, we start with a left-closed interval $[0, 1)$. For each sequence symbol the current interval is divided into subintervals of length proportional to the frequencies of character occurrences. Then we choose the subinterval of the current symbol. This procedure is repeated for all characters from the input sequence. At the end we output the binary representation of any number from the final interval.

Suppose that the table of frequencies is the same as in the Huffman coding example. Figure 2.2 shows the arithmetic encoding process of the five symbols abrac. We start from the interval $[0, 1)$ and then choose the subintervals relating to the encoded symbol. The subintervals become smaller and smaller during the coding. Arithmetic coding works on infinite-precision numbers. With such an assumption it can be proved that the Elias algorithm (unpublished but described, e.g., by Jelinek [91]) which was the first attempt to the arithmetic coding is no more than 2 bits away from the optimum for the whole sequence. Of course from practical reasons we cannot meet this assumption, and we work on finite-

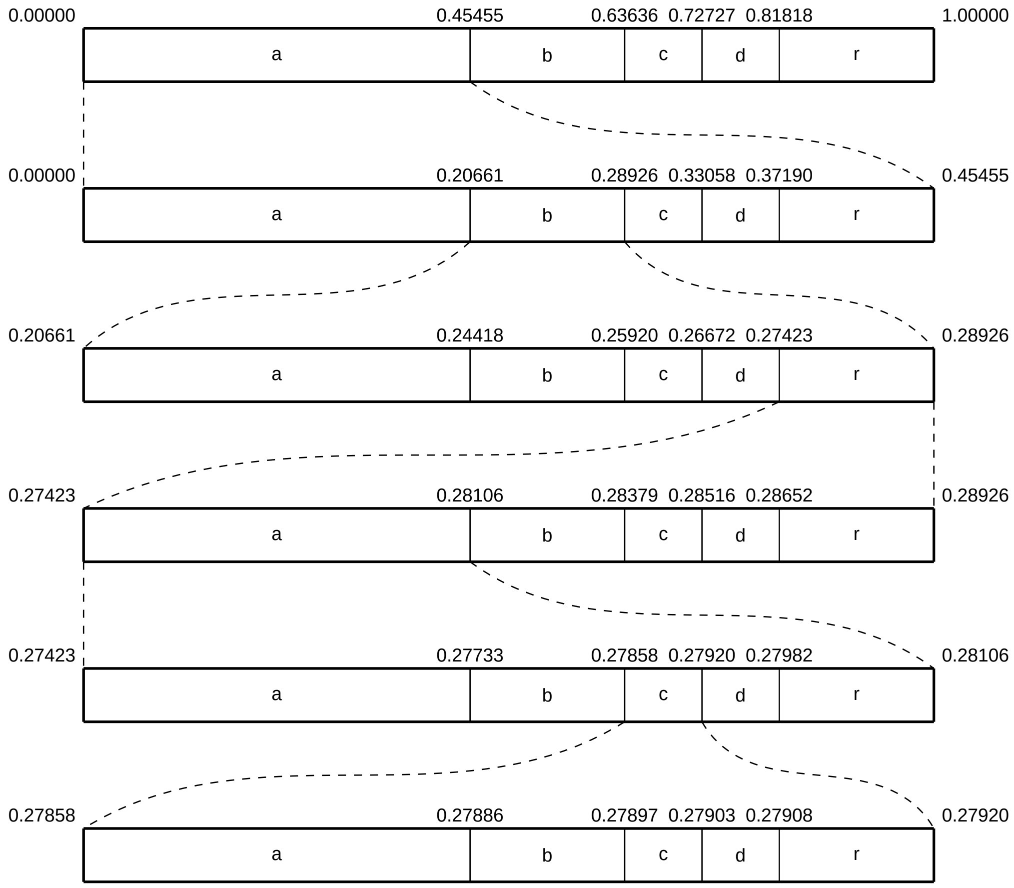

Figure 2.2: Example of the arithmetic coding process

precision numbers accepting some efficiency loss. The loss is, however, small and usually can be neglected.

The first idea of the arithmetic coding was invented in 1960s. Its more precise description was provided, however, by Rissanen [134], Pasco [125], Rissanen and Langdon [136], Langdon [97], Rubin [138], and Guazzo [78] about ten years later. Some of the interesting works on this topic was also presented by Witten *et al.* [199] and Howard and Vitter [84, 85, 86]. A modern approach to the arithmetic coding is described in the work of Moffat *et al.* [115]. The recent authors showed that in the worst case their version of the finite-precision arithmetic coding is only 0.006 bits per symbol worse than the optimum.

An important problem in the arithmetic coding is to store in an efficient way the changing probabilities of symbol occurrences. Fenwick [64] presented an elegant tree-based structure for solving this problem. An improvement to this method was proposed by Moffat [114].

2.6 Classes of sources

2.6.1 Types of data

Before choosing the right compression algorithm we must know the sequence to be compressed. Ideally, we would know that in a particular case we have, e.g., a Dickens novel, a Picasso painting, or an article from a newspaper. In such a case, we can choose the modelling method that fits such a sequence best. If our knowledge is even better and we know that the sequence is an article from the New York Times we can choose the more suitable model corresponding to the newspaper style. Going further, if we know the author of the article, we can make even better choice. Using this knowledge one may choose a model well adjusted to the characteristics of the current sequence. To apply such an approach, providing different modelling methods for writers, painters, newspapers, and so on, is insufficient. It is also needed to provide different modelling methods for all newspapers and all authors publishing there. The number of modelling methods would be incredibly large in this case. The more so, we do not know the future writers and painters and we cannot prepare models for their works.

On the other side, we cannot assume nothing about the sequence. If we do so, we have no way of finding similarities. The less we know about the sequence, the less we can utilise.

To make a compression possible we have to make a compromise. The standard approach is to define the *classes of sources* producing sequences of different types. We assume that the possible sequences can be treated as an output of some of the sources. The goal is to choose the source's class which approximates the sequence best. Then we apply a universal compression algorithm that works well on the sources from the chosen class. This strategy offers a possibility of reduction the number of modelling methods to a reasonable level.

2.6.2 Memoryless source

Let us start the description of source types from the simplest one which is a *memoryless source*. We assume that $\Theta = \{\theta_0, \ldots, \theta_{k-1}\}$ is a set of probabilities of occurrence of all symbols from the alphabet. These parameters fully define the source.

Such sources can be viewed as finite-state machines (FSM) with a single state and k loop-transitions. Each transition is denoted by a different character, a_i, from the alphabet, and with each of them the probability θ_i is associated. An example of the memoryless source is presented in Figure 2.3.

The memoryless source produces a sequence of randomly chosen symbols according to its parameters. The only regularity in the produced sequence is

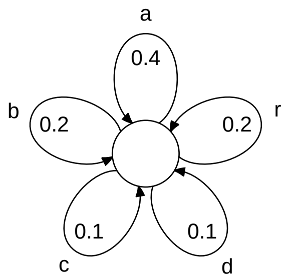

Figure 2.3: Example of the memoryless source

that the frequency of symbol occurrences is close to the probabilities being the source parameters.

2.6.3 Piecewise stationary memoryless source

The parameters of the memoryless source do not depend on the number of symbols it generated so far. This means that the probability of occurrence of each symbol is independent from its position in the output sequence. Such sources, which do not vary their characteristics in time, are called *stationary*.

The *piecewise stationary memoryless source* is a memoryless source, where the set of probabilities Θ depends on the position in the output sequence. This means that we assume a sequence $\langle \Theta_1, \ldots, \Theta_m \rangle$ of probabilities sets, and a related sequence of positions, $\langle t_1, \ldots, t_m \rangle$, after which every set Θ_i becomes actual to the source.

This source is *nonstationary*, because its characteristics varies in time. Typically we assume that the sequence to be compressed is produced by a stationary source. Therefore all other source classes considered in the dissertation are stationary. The reason to distinguish the piecewise stationary memoryless sources is that they are closely related to one stage of the Burrows–Wheeler compression algorithm, what we will discuss in Section 4.1.3.

2.6.4 Finite-state machine sources

A *Markov source* (Figure 2.4) is a *finite-state machine* which has a set of states $\mathcal{S} = \{s_0, \ldots, s_{m-1}\}$ and some transitions. There can be at most k transitions from each state and all of them are denoted by a different symbol. Each transition has some probability of choosing it. The next state is completely specified by the previous one and a current symbol.

Finite-order FSM sources are subsets of Markov sources. Every node is associated with a set of sequences of length not greater than d, where d is called an *order* of the source. The current state is completely specified by the last d sym-

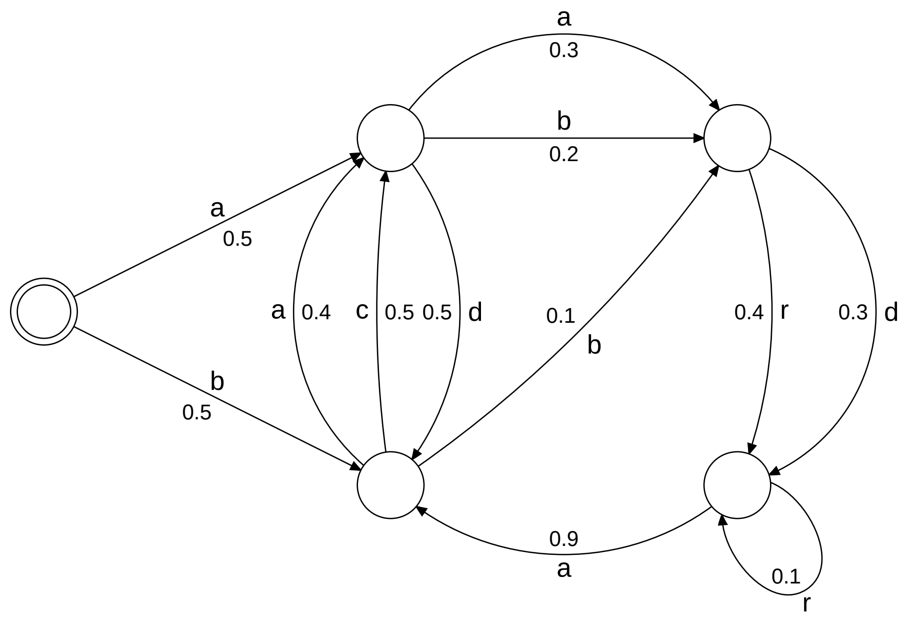

Figure 2.4: Example of the Markov source

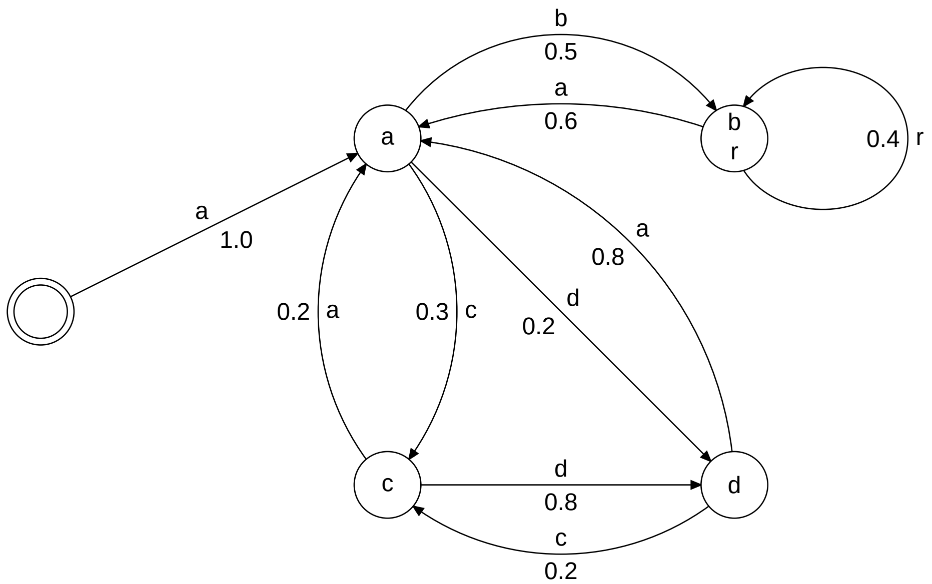

Figure 2.5: Example of the finite-order FSM source

bols. The next state is specified by the current one and the current symbol. An example of a finite-order FSM source is shown in Figure 2.5.

The *FSMX sources* [135] are subsets of finite-order FSM sources. The reduction is made by the additional assumption that sets denoting the nodes contain exactly one element.

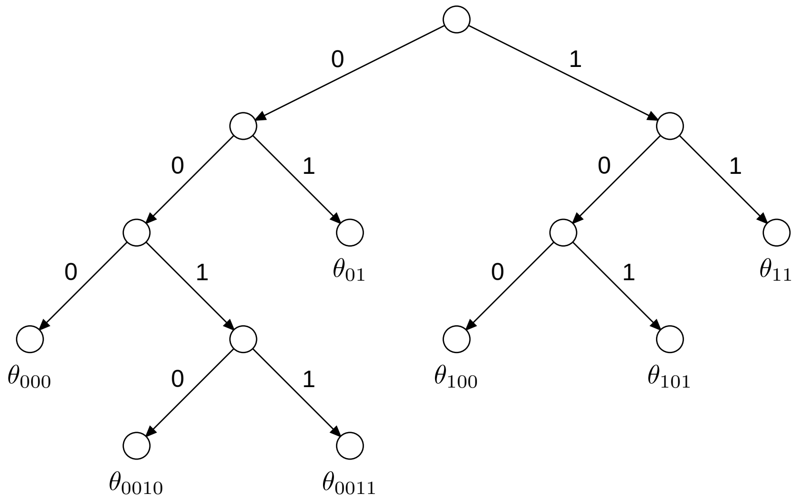

Figure 2.6: Example of the binary CT-source

2.6.5 Context tree sources

Context tree sources (CT-sources) were introduced by Willems *et al.* [185] in 1995. A finite memory CT-source ω is specified by a set $\mathcal{S}$ of contexts s (sequences over the alphabet $\mathcal{A}$) of length not greater than the maximum order d ($|s| \leq d$). The set S of contexts should be *complete*, what means that for every possible sequence x, of length not smaller than d, there exists a context $s \in \mathcal{S}$, being a suffix of x. Moreover, there should be exactly one such context, called a *proper* context. Each context $s \in \mathcal{S}$ has a conditional probability distribution $\{\Theta_s, s \in \mathcal{S}\} = \{\{\theta(a|s), a \in \mathcal{A}\}, s \in \mathcal{S}\}$ of producing the next symbol. A sample CT-source for binary alphabet is presented in Figure 2.6.

Context tree sources can be viewed as a generalisation of the FSMX sources. To specify when a relation between the FSMX source and the context tree source holds, we have to define some terms. The *generator* of the component $v_{1..i}$ is its prefix $v_{1..i-1}$. We name the set $\mathcal{S}$ *closed* if a generator of each suffix of $s \in \mathcal{S}$ belongs to the set $\mathcal{S}$ or it is a suffix of s. There exists an equivalent FSMX source to the context tree source if the set $\mathcal{S}$ if closed. The number of states in the FSMX source is the number of components in the set $\mathcal{S}$. There is also an FSMX source related to every context tree source but the number of states in the FSM is greater then $|\mathcal{S}|$ if the set $\mathcal{S}$ is not closed.

2.7 Families of universal algorithms for lossless data compression

2.7.1 Universal compression

We noticed that it is impossible to design a single compression method for all types of data without some knowledge of the sequence. It is also impossible to prepare a different compression algorithm for every possible sequence. The reasonable choice is to invent a data compression algorithm for some general source classes, and to use such an algorithm to the sequences which can be treated, with a high precision, as outputs of the assumed source.

Typical sequences appearing in real world contain texts, databases, pictures, binary data. Markov sources, finite-order FSM sources, FSMX sources, and context tree sources were invented to model such real sequences. Usually real sequences can be successfully approximated as produced by these sources. Therefore it is justified to call algorithms designed to work well on sequences produced by such sources *universal*.

Sometimes, before the compression process, it is useful to transpose the sequence in some way to achieve a better fit to the assumption. For example, image data are often decomposed before the compression. This means that every colour component (red, green, and blue) is compressed separately. We consider the universal compression algorithms which in general do not include such preliminary transpositions.

We are now going to describe the most popular universal data compression algorithms. We start from the classic Ziv–Lempel algorithms then we present the more recent propositions.

2.7.2 Ziv–Lempel algorithms

Main idea

Probably the most popular data compression algorithms are *Ziv–Lempel methods*, first described in 1977. These algorithms are *dictionary* methods: during the compression they build a dictionary from the components appeared in the past and use it to reduce the sequence length if the same component appears in the future.

Let us suppose that the input sequence starts with characters abracadabra. We can notice that the first four symbols are abra and the same symbols appear in the sequence once more (from position 8). We can reduce the sequence length replacing the second occurrence of abra by a special marker denoting a repetition of the previous component. Usually we can choose the subsequences to be replaced by a special marker in various ways (take a look for example at the sequence abracadabradab, where we can replace the second appearance of component abra or adab, but not both of them). At a given moment, we cannot find out which replacement will give better results in the future. Therefore the Ziv–Lempel algorithm use heuristics for choosing the replacements.

The other problem is how to build the dictionary and how to denote the replacements of components. There are two major versions of the Ziv–Lempel algorithms: LZ77 [202] and LZ78 [203], and some minor modifications.

The main idea of the Ziv–Lempel methods is based on the assumption that there are repeated components in the input sequence, x. This means that the probability of occurrence of the current symbol, x_i, after a few previous characters, a component $x_{i-j..i-1}$, is not uniform. The FSM, FSMX, and CT-source with nonuniform probability distribution in states fulfil this assumption.

LZ77 algorithm

We are going to describe the LZ77 algorithm using the sample sequence abracadabra (Figure 2.7). At the beginning we choose two numbers: l_s—the maximum length of the identical subsequences, and l_b—the length of the buffer storing the past characters. Let us set $l_s = 4$ and $l_b = 8$. The LZ77 algorithm works on the buffer sequence b that is composed of l_b previous and l_s future symbols. There are no previous symbols at the beginning of the compression because the current position i in the x sequence equals 1. Therefore we initialise the buffer b with l_b repetitions of the first symbol from the alphabet (a in our example), and the l_s first symbols from the sequence. Now we find the longest prefix of the current part of component $b_{(l_b+1)..(l_b+l_s)}$ (underlined characters in column "Buffer"), in the buffer starting not further than at the l_bth position. We find the prefix a of length $l_l = 1$ starting at the 8th position in our example. Then we output a triple $\langle 8, 1, \mathsf{b}\rangle$ describing the repeated part of the sequence. The first element of the triple, 8, is a position where the identical subsequence starts, the second, 1, is the length of the subsequence and the third, b, is the character following the repeated sequence, x_{i+l_l}. Then the buffer is shifted $l_l + 1$ characters to the left, filled from the right with the component $x_{(i+l_s+1)..(i+l_s+1+l_l)}$, and the current position, i, is changed to $i + l_l + 1$. The output is a sequence of triples; in our example: $\langle 8, 1, \mathsf{b}\rangle$, $\langle 1, 0, \mathsf{r}\rangle$, $\langle 6, 1, \mathsf{c}\rangle$, $\langle 4, 1, \mathsf{d}\rangle$, $\langle 2, 4, \square\rangle$. (The special character $\square$ denotes no character.)

The parameters l_b and l_s are much larger in real implementations, so we can find significantly longer identical components. Choosing l_b and l_s we have to remember that using large values entails the need to use much more space to store the triples. To achieve better compression, in modern versions of the LZ77 algorithm the sequence of triples is modelled by simple algorithms, and then encoded with the Huffman or arithmetic coder.

The LZ77 algorithm was intensively studied in last years. From many works at this field we should mention the algorithms LZSS by Storer and Szyman-

Remaining sequence	Buffer	Longest prefix	Code
abracadabra	aaaaaaaaabra	a	$\langle 8, 1, \mathrm{b} \rangle$
racadabra	aaaaaaabraca		$\langle 1, 0, \mathrm{r} \rangle$
acadabra	aaaaaabracad	a	$\langle 6, 1, \mathrm{c} \rangle$
adabra	aaaabracadab	a	$\langle 4, 1, \mathrm{d} \rangle$
abra	aabracadabra	abra	$\langle 2, 4, \square \rangle$

Figure 2.7: Example of the LZ77 algorithm processing the sequence abracadabra

Compressor		Dictionary	
Remaining sequence	Code	Index	Sequence
abracadabra	$\langle 0, \mathrm{a} \rangle$	1	a
bracadabra	$\langle 0, \mathrm{b} \rangle$	2	b
racadabra	$\langle 0, \mathrm{r} \rangle$	3	r
acadabra	$\langle 1, \mathrm{c} \rangle$	4	ac
adabra	$\langle 1, \mathrm{d} \rangle$	5	ad
abra	$\langle 1, \mathrm{b} \rangle$	6	ab
ra	$\langle 3, \mathrm{a} \rangle$	7	ra

Figure 2.8: Example of the LZ78 algorithm processing the sequence abracadabra

ski [159], LZFG by Fiala and Greene [70], and LZRW by Williams [193, 194]. Further improvements were introduced also by Bell [21], Bell and Kulp [23], Bell and Witten [24], Gutmann and Bell [79], and Horspool [82].

LZ78 algorithm

The second algorithm developed by Ziv and Lempel is the LZ78 method [203]. It uses a different approach to the problem of representing previous part of a sequence. Instead of the buffer that exists in LZ77, a dictionary storing the components that are encoded is built. We describe the method using the same sample sequence abracadabra (Figure 2.8).

The algorithm starts with an empty dictionary. During the compression, we search for the longest prefix of the subsequence starting at the current position, i, in the dictionary, finding the component $x_{i..(i+l_l-1)}$ of length l_l. Then we output a pair. The first element of the pair is an index of the found sequence in the dictionary. The second element is the next character in the input sequence, x_{i+l_l}. Next we expand the dictionary with a component $x_{i..(i+l_l)}$ being a concatenation of found subsequence and the next character in the sequence x. Then we change the current position i to $i + l_l + 1$. The steps described above are repeated until we reach the end of the sequence x. The output is a sequence of pairs; in our example: $\langle 0, \mathsf{a}\rangle$, $\langle 0, \mathsf{b}\rangle$, $\langle 0, \mathsf{r}\rangle$, $\langle 1, \mathsf{c}\rangle$, $\langle 1, \mathsf{d}\rangle$, $\langle 1, \mathsf{b}\rangle$, $\langle 3, \mathsf{a}\rangle$.

During the compression process the dictionary grows, so the indexes become larger numbers and require more bits to be encoded. Sometimes it is unprofitable to let the dictionary grow unrestrictedly and the dictionary is periodically purged. (Different versions of LZ78 use different strategies in this regard.)

From many works on LZ78-related algorithms the most interesting ones are the propositions by Miller and Wegman [112], Hoang *et al.* [80], and the LZW algorithm by Welch [187]. The LZW algorithm is used by a well-known UNIX *compress* program.

2.7.3 Prediction by partial matching algorithms

The *prediction by partial matching* (PPM) data compression method was developed by Cleary and Witten [48] in 1984. The main idea of this algorithm is to gather the frequencies of symbol occurrences in all possible contexts in the past and use them to predict probability of occurrence of the current symbol x_i. This probability is then used to encode the symbol x_i with the arithmetic coder.

Many versions of PPM algorithm have been developed since the time of its invention. One of the main differences between them is the maximum order of contexts they consider. There is also an algorithm, PPM* [49], that works with an unbounded context length. The limitation of the context length in the first versions of the algorithm was motivated by the exponential growth of the number of possible contexts together to the order, what causes a proportional growth of the space requirements. There are methods for reducing the space requirements and nowadays it is possible to work with long contexts (even up to 128 symbols). When we choose a too large order, we, however, often meet the situation that a current context has not appeared in the past. Overcoming this problem entails some additional cost so we must decide to some compromise in choosing the order.

Let us now take a look at the compression process shown in Figure 2.9. The figure shows a table of frequencies of symbol occurrences after encoding the sequence abracadabra. The values in column c are the numbers of symbol occurrences in each context. The values in column p are the estimated probabilities of occurrence of each symbol in the different contexts. The *Esc* character is a special symbol, called *escape code*, which means that the current character x_i has not occurred in a context so far. It is important that the escape code is present in every context, because there is always a possibility that a new character appears in the current context. The order -1 is a special order included to ensure that all symbols from the alphabet have a non-zero probability in some context.

The last two characters of the sample sequence are ra and this is the context of order 2. Let us assume that the current character x_i is a. The encoding proceeds as follows. First, we find the context $x_{(i-2)..(i-1)} =$ ra in the table. We see

Order 2

Ctx	Chars	c	p
ab	r	2	2/3
	Esc	1	1/3
ac	a	1	1/2
	Esc	1	1/2
ad	a	1	1/2
	Esc	1	1/2
br	a	2	2/3
	Esc	1	1/3
ca	d	1	1/2
	Esc	1	1/2
da	b	1	1/2
	Esc	1	1/2
ra	c	1	1/2
	Esc	1	1/2

Order 1

Ctx	Chars	c	p
a	b	2	2/7
	c	1	1/7
	d	1	1/7
	Esc	3	3/7
b	r	2	2/3
	Esc	1	1/3
c	a	1	1/2
	Esc	1	1/2
d	a	a	1/2
	Esc	1	1/2
r	a	2	2/3
	Esc	1	1/3

Order 0

Ctx	Chars	c	p
	a	5	5/16
	b	2	2/16
	c	1	1/16
	d	1	1/16
	r	2	2/16
	Esc	5	5/16

Order -1

Ctx	Chars	c	p
		1	$1/k$

Figure 2.9: Example of the PPMC algorithm (1)

that the symbol $x_i = \mathsf{a}$ has not appeared in this context so far. Therefore we need to choose the escape code and encode it with the estimated probability 1/2. Because we have not found the character a, we need to decrease the order to 1, achieving the context $x_{(i-1)..(i-1)} = \mathsf{a}$. Then we look at this context for a. We are still unsuccessful, so we need to choose the escape code and encode it with the estimated probability 3/7. Then we decrease the order to 0 and look at the frequency table. We see that the probability of occurrence of symbol a in the context of order 0 is 5/16 so we encode it with this probability. Next the statistics of all context from order 0 to 2 are updated (Figure 2.10). The code length of the encoded character is $-\log\frac{1}{2} - \log\frac{3}{7} - \log\frac{5}{16} \approx 3.900$ bits.

We can improve this result if we notice that after reducing the order to 1 we know that the current character cannot be c. If it were c we would encode it in the context of order 2, so we can correct the estimated probabilities discerning the occurrences of symbol c in the context a. Then we choose the escape code but with a modified estimated probability 3/6. Similarly, after limiting the order to 0 we can exclude occurrences of characters b, c, d and we can encode the symbol a with the estimated probability 5/12. In this case, the code length of the encoded character is $-\log\frac{1}{2} - \log\frac{3}{6} - \log\frac{5}{12} \approx 3.263$ bits. This process is called *applying exclusions*.

The second important difference between PPM algorithms is the way of estimating probabilities of the escape code. The large number of methods of probability estimation follows from the fact that no method was proved to be the best. Cleary and Witten [48] proposed two methods, and their algorithms are called PPMA, PPMB. The method C employed in PPMC algorithm was pro-

Order 2

Ctx	Chars	c	p
ab	r	2	2/3
	Esc	1	1/3
ac	a	1	1/2
	Esc	1	1/2
ad	a	1	1/2
	Esc	1	1/2
br	a	2	2/3
	Esc	1	1/3
ca	d	1	1/2
	Esc	1	1/2
da	b	1	1/2
	Esc	1	1/2
ra	a	1	1/4
	c	1	1/4
	Esc	2	1/4

Order 1

Ctx	Chars	c	p
a	a	1	1/9
	b	2	2/9
	c	1	1/9
	d	1	1/9
	Esc	4	4/9
b	r	2	2/3
	Esc	1	1/3
c	a	1	1/2
	Esc	1	1/2
d	a	a	1/2
	Esc	1	1/2
r	a	2	2/3
	Esc	1	1/3

Order 0

Ctx	Chars	c	p
	a	6	6/17
	b	2	2/17
	c	1	1/17
	d	1	1/17
	r	2	2/17
	Esc	5	5/17

Order −1

Ctx	Chars	c	p
		1	$1/k$

Figure 2.10: Example of the PPMC algorithm (2)

posed by Moffat [113]. (We use PPMC algorithm in our example.) The methods PPMD and PPME were proposed by Howard [83] and Åberg *et al.* [2] respectively. Other solutions: PPMP, PPMX, PPMXC were introduced by Witten and Bell [196].

We can realise that choosing a too large order may cause a need of encoding many escape codes until we find the context containing the current symbol. Choosing a small order causes, however, that we do not use the information of statistics in longer contexts. There are a number of works in this field. We mention here the PPM* algorithm [49] by Cleary and Teahan, Bunton's propositions [33, 34, 35, 36, 37, 38], and the most recent propositions by Shkarin [154, 155], which are the state of the art PPM algorithms today.

The way of proper choosing the order and effective encoding the escape codes are crucial. Recently such solutions as *local order estimation* (LOE) and *secondary escape estimation* (SEE) were proposed by Bloom [30] to overcome these problems. Both these strategies are used currently by Shkarin [154, 155, 156].

The PPM algorithms work well on the assumption that the probabilities of symbol occurrences in contexts are nonuniform. This assumption is fulfilled for the FSM, FSMX, and CT-sources with nonuniform probability distribution in the states. The additional assumption in the PPM algorithms is that the probability distribution in the context is similar to the probability distribution in contexts being its suffixes. The formulation of the source classes discussed in Section 2.6 does not give a justification for this assumption but typical sequences in the real world fulfil this additional requirement too.

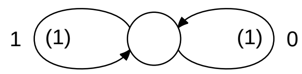

Figure 2.11: Initial situation in the DMC algorithm

At the end we notice that the PPM algorithms yield the best compression rates today. Unfortunately their time and space complexities are relatively high, because they maintain a complicated model of data.

2.7.4 Dynamic Markov coding algorithm

The *dynamic Markov coding* algorithm (DMC) was invented in 1987 by Cormack and Horspool [52]. The main idea of this method is to discover the Markov source that has produced the input sequence. For a clear presentation we illustrate the work of the DMC algorithm on a binary alphabet.

The DMC algorithm starts with an FSM with one state and two transitions as shown in Figure 2.11. Next it processes the input sequence going through the FSM and counting the frequency of using each transition. When some transition is used often, the DMC algorithm *clones* the destination state. Figure 2.12 shows the state s split into states s' and s''. All the outgoing transitions of s are copied to the new states, but the only transition to the state s'' is the one that caused the cloning process. Other incoming transitions to s are copied to the state s'. After cloning we have to assign counts to the outgoing transitions from s' and s''. We do that by considering two requirements. First, the ratio of counts related to the new transitions outgoing from the states s' and s'' should be as close to the one of the outgoing transitions from s as possible. The second, the sums of counts of all the incoming and outgoing transitions in the states s' and s'' should be the same. The result of the cloning is presented in Figure 2.13.

We described the DMC algorithm assuming the binary alphabet to simplify the presentation. It is of course possible to implement it on the alphabet of larger size. Teuhola and Raita investigated such an approach introducing a *generalised dynamic Markov coder* (GDMC) [169].

The DMC algorithm was not examined in the literature as deeply as the PPM algorithms were. One reason is that the implementation for alphabets of typical sizes becomes harder than for binary ones, and the programs employing the DMC algorithm work significantly slower than these using the PPM methods.

Interesting works on the DMC algorithm were carried out by Bell and Moffat [19], Yu [201], and Bunton [32, 33]. The latter author presents a variant of the DMC called Lazy DMC, that outperforms, in the terms of the compression ratio, the existing DMC methods. An interesting comparison of the best PPM and

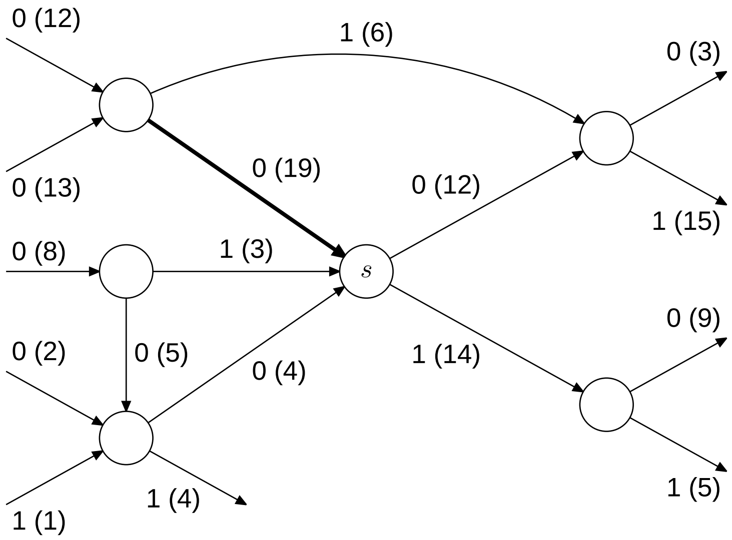

Figure 2.12: Before cloning in the DMC algorithm

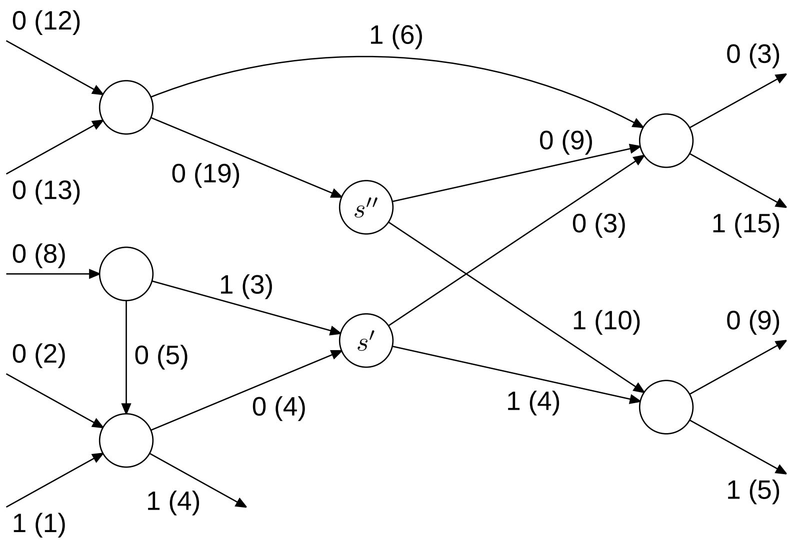

Figure 2.13: After cloning in the DMC algorithm

DMC algorithms showing the significant advantage in the compression ratio of the PPM algorithms is demonstrated in Reference [33].

2.7.5 Context tree weighting algorithm

The *context tree weighting* algorithm was introduced by Willems *et al.* [189]. Its main assumption is that the sequence was produced by a context tree source of an unknown structure and parameters.

For a clear presentation we follow the way that the authors used to present their algorithm, rather than describing it in work, what could be confusing without introducing some notions. The authors start with a simple binary memoryless source and notice that using the Krichevsky–Trofimov estimator [95] to estimate the probability for the arithmetic coding, we can encode every sequence produced by any such source with a small bounded inefficiency equals $1/2 \log n + 1$. Next they assume that the source is a binary CT-source of known structure (the set of contexts) and unknown parameters (the probabilities of producing 0 or 1 in each context). The authors show how the maximum redundancy of the encoded sequence grows in such a case. The result is strictly bounded only by the size of the context set and the length of the sequence. The last step is to assume that we also do not know the structure of the context tree source. The only thing we know is the maximum length d of the context from the set $\mathcal{S}$. The authors show how to employ an elegant weighting procedure over all the possible context tree sources of the maximum depth d. They show that, in this case, a maximum redundancy is also strictly bounded for their algorithm. The first idea of the CTW algorithm was extended in further works [171, 180, 181, 188, 190, 192], and the simple introduction to the basic concepts of the CTW algorithm is presented by Willems *et al.* [191].

The significant disadvantage of the CTW algorithm is the fact that the context tree sources are binary. The formulation of the CTW algorithm for larger alphabets is possible, but the mathematics and computations become much more complicated. Hence, to employ the CTW algorithm to the non-binary sequence, we have to decompose it into binary sequences first.

The results of Willems *et al.* are theoretical, and the authors do not supply experimental tests. The compression for text sequences was investigated by Åberg and Shtarkov [1], Tjalkens *et al.* [170], Sadakane *et al.* [140], Suzuki [161], and Volf [179]. Ekstrand [57, 58] as well as Arimura *et al.* [8] considered also the compression of sequences of grey scale images with the CTW algorithm. (The works mentioned in this paragraph contain also experimental results.)

2.7.6 Switching method

The *switching method* proposed by Volf and Willems [182, 183] is not in fact a new universal compression algorithm. This method employs two compression algorithms such as CTW, DMC, LZ77, PPM, or other. The input sequence is then compressed with both algorithms and then the switching procedure decides which parts of the sequence should be compressed with which algorithm to obtain the best compression ratio. The output sequence is composed of parts of output sequences produced by both algorithms and the information where to switch between them. This method gives very good compression ratios and this is the reason we mention it here. We, however, do so only for the possibility of comparing the experimental results.

2.8 Specialised compression algorithms

Sometimes the input sequence is very specific and applying the universal data compression algorithm does not give satisfactory results. Many specialised compression algorithms were proposed for such specific data. They work well on the assumed types of sequences but are useless for other types. We enumerate here only a few examples indicating the need of such algorithms. As we aim at universal algorithms, we will not mention specialised ones in subsequent chapters.

The first algorithm we mention is Inglis [88] method for scanned texts compression. This problem is important in archiving texts that are not available in an electronic form. The algorithm exploits the knowledge of the picture and finds consistent objects (usually letters) that are almost identical. This process is slightly relevant to the Optical Character Recognition (OCR), though the goal is not to recognise letters. The algorithm only looks for similarities. This approach for scanned texts effects in a vast improvement of the compression ratio with regard to standard lossless data compression algorithms usually applied for images.

The other interesting example is compressing DNA sequences. Loewenstern and Yianilos investigated the entropy bound of such sequences [104]. Nevill–Manning and Witten concluded that the genome sequence is almost incompressible [119]. Chen *et al.* [45] show that applying sophisticated methods based on the knowledge of the structure of DNA we can achieve some compression. The other approach is shown by Apostolico and Lonardi [5].

The universal algorithms work well on sequences containing text, but we can improve compression ratio for texts when we use more sophisticated algorithms. The first works on text compression was done by Shannon [152] in 1951. He investigated the properties of English text and bounded its entropy relating on experiments with people. This kind of data is specific, because the text has complicated structure. At the first level it is composed of letters that are grouped into words. The words form sentences, which are parts of paragraphs, and so on. The structure of sentences is specified by semantics. We can treat the texts as the output of a CT-source but we can go further and exploit more. The recent extensive discussion of how we can improve compression ratio for texts was presented by Brown *et al.* [31] and Teahan *et al.* [163, 164, 165, 166, 167, 197].

The last specific compression problem we mention here is storing a sequence representing a finite set of finite sequences (words), i.e., a *lexicon*. There are different possible methods of compressing lexicons, and we notice here only one of them by Daciuk *et al.* [53] and Ciura and Deorowicz [47], where the effective compression goes hand in hand with efficient usage of the data. Namely, the compressed sequence can be searched for given words faster than the uncompressed one.

Chapter 3

Algorithms based on the Burrows–Wheeler transform

If they don't suit your purpose as they are,
transform them into something more satisfactory.

— SAKI [HECTOR HUGH MUNRO]
The Chronicles of Clovis (1912)

3.1 Description of the algorithm

3.1.1 Compression algorithm

Structure of the algorithm

In 1994, Burrows and Wheeler [39] presented a data compression algorithm based on the Burrows–Wheeler transform (BWT). Its compression ratios were comparable with the ones obtained using known best methods. This algorithm is in the focus of our interests, so we describe it more precisely.

At the beginning of the discussion of the Burrows–Wheeler compression algorithm let us provide an insight description of its stages (Figure 3.1). The presentation is illustrated by a step-by-step example of working of the BWCA.

Burrows–Wheeler transform

The input datum of the BWCA is a sequence x of length n. First we compute the *Burrows–Wheeler transform* (BWT). To achieve this, n sequences are created

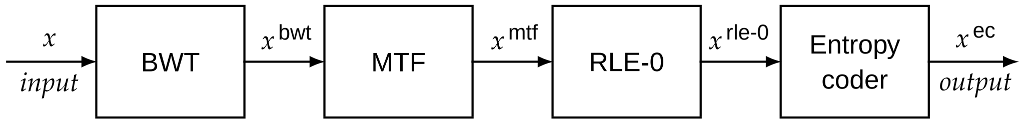

Figure 3.1: Burrows–Wheeler compression algorithm

in such a way that the ith sequence is the sequence x rotated by $i-1$ symbols. These sequences are put into an $n \times n$ matrix, $M(x)$:

$$M(x) = \begin{bmatrix} x_1 & x_2 & \cdots & x_{n-1} & x_n \\ x_2 & x_3 & \cdots & x_n & x_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{n-1} & x_n & \cdots & x_{n-3} & x_{n-2} \\ x_n & x_1 & \cdots & x_{n-2} & x_{n-1} \end{bmatrix}.$$

The matrix $M(x)$ is then transformed into a matrix $\widetilde{M}(x)$ by sorting its rows in the lexicographic order. Let $R(x)$ denote the row number of the sequence x in the matrix $\widetilde{M}(x)$. The result of the BWT is the pair comprising the last column of the matrix $\widetilde{M}(x)$, which we denote x^{bwt}, and $R(x)$.

Figure 3.2 shows the example for the sequence $x =$ abracadabra. In this case, the results of the BWT are $x^{\text{bwt}} =$ rdarcaaaabb and $R(x) = 3$. The example shows how the original procedure by Burrows and Wheeler works. Subsequent research has shown that a better relation to the classes of sources can be obtained if the input sequence is augmented with a special character \$, called a *sentinel*. It is the last character of the alphabet and it appears exactly once in the sequence x. In fact, the sentinel is not a part of the data which are compressed, and it is appended to the input sequence before the BWT stage. Later on we will discuss in depth the motivation of using the sentinel. Figure 3.3 shows an example of the Burrows–Wheeler transform for the sequence $x =$ abracadabra\$. Now the results are $x^{\text{bwt}} =$ \$drcraaaabba and $R(x) = 1$.

In this dissertation, the modified version of the BWCA, with the sentinel is considered. The difference between this version and the original one is small. When it is possible or important, we, however, notice how the results change comparing to the original version.

Let us return to the example. We can see that the $R(x)$ precisely defines where the sentinel appears in the sequence x^{bwt} and vice versa—the $R(x)$ is the number of the row where the sentinel is located in the last column. Therefore one of them is redundant (if the sentinel is not used, the $R(x)$ is necessary to compute the reverse BWT and can be omitted). The sentinel is, however, only an abstract concept. Usually all characters from the alphabet can be used in the sequence x and it will be necessary to expand the original alphabet by one

$$
M(x) = \begin{bmatrix}
\mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{c} & \mathsf{a} & \mathsf{d} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} \\
\mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{c} & \mathsf{a} & \mathsf{d} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{a} \\
\mathsf{r} & \mathsf{a} & \mathsf{c} & \mathsf{a} & \mathsf{d} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{a} & \mathsf{b} \\
\mathsf{a} & \mathsf{c} & \mathsf{a} & \mathsf{d} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{a} & \mathsf{b} & \mathsf{r} \\
\mathsf{c} & \mathsf{a} & \mathsf{d} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} \\
\mathsf{a} & \mathsf{d} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{c} \\
\mathsf{d} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{c} & \mathsf{a} \\
\mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{c} & \mathsf{a} & \mathsf{d} \\
\mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{c} & \mathsf{a} & \mathsf{d} & \mathsf{a} \\
\mathsf{r} & \mathsf{a} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{c} & \mathsf{a} & \mathsf{d} & \mathsf{a} & \mathsf{b} \\
\mathsf{a} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{c} & \mathsf{a} & \mathsf{d} & \mathsf{a} & \mathsf{b} & \mathsf{r}
\end{bmatrix}
$$

$$
\widetilde{M}(x) = \begin{bmatrix}
\mathsf{a} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{c} & \mathsf{a} & \mathsf{d} & \mathsf{a} & \mathsf{b} & \mathbf{r} \\
\mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{c} & \mathsf{a} & \mathbf{d} \\
\underline{\mathsf{a}} & \underline{\mathsf{b}} & \underline{\mathsf{r}} & \underline{\mathsf{a}} & \underline{\mathsf{c}} & \underline{\mathsf{a}} & \underline{\mathsf{d}} & \underline{\mathsf{a}} & \underline{\mathsf{b}} & \underline{\mathsf{r}} & \underline{\mathbf{a}} \\
\mathsf{a} & \mathsf{c} & \mathsf{a} & \mathsf{d} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{a} & \mathsf{b} & \mathbf{r} \\
\mathsf{a} & \mathsf{d} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathbf{c} \\
\mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{c} & \mathsf{a} & \mathsf{d} & \mathbf{a} \\
\mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{c} & \mathsf{a} & \mathsf{d} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathbf{a} \\
\mathsf{c} & \mathsf{a} & \mathsf{d} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathbf{a} \\
\mathsf{d} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{c} & \mathbf{a} \\
\mathsf{r} & \mathsf{a} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{c} & \mathsf{a} & \mathsf{d} & \mathsf{a} & \mathbf{b} \\
\mathsf{r} & \mathsf{a} & \mathsf{c} & \mathsf{a} & \mathsf{d} & \mathsf{a} & \mathsf{b} & \mathsf{r} & \mathsf{a} & \mathsf{a} & \mathbf{b}
\end{bmatrix}
$$

Figure 3.2: Example of the BWT for the sequence $x =$ abracadabra

symbol. The sizes of alphabets are typically powers of 2 and we would have to increase also the number of bits per symbol if the sentinel was to be stored explicitly. As it is rather complicated to store the sentinel, it is easier to remove it from the sequence x^{bwt} and store the value $R(x)$.

Move-to-front transform

When the BWT is completed, the sequence x^{bwt} is encoded using the move-to-front (MTF) transform [26]. The coding proceeds as follows. First the list $L = (a_0, a_1, \ldots, a_{k-1})$ consisting of the symbols of the alphabet $\mathcal{A}$ is created. Then to each symbol x_i^{bwt}, where $i = 1, 2, \ldots, n$, a number p_i is assigned, such that x_i^{bwt} is equal to the p_ith element of the list L, and then this element is moved to the beginning of the list L. As a result, a sequence x^{mtf} over the alphabet $\mathcal{A}^{\text{mtf}}$ consisting of integer numbers from the range $[0, k-1]$ is obtained. Figure 3.4 presents an example of the MTF transform for the sample sequence obtained from the BWT stage. The result of this stage is $x^{\text{mtf}} = 34413000401$.

$$M(x) = \begin{bmatrix}
a & b & r & a & c & a & d & a & b & r & a & \$ \\
b & r & a & c & a & d & a & b & r & a & \$ & a \\
r & a & c & a & d & a & b & r & a & \$ & a & b \\
a & c & a & d & a & b & r & a & \$ & a & b & r \\
c & a & d & a & b & r & a & \$ & a & b & r & a \\
a & d & a & b & r & a & \$ & a & b & r & a & c \\
d & a & b & r & a & \$ & a & b & r & a & c & a \\
a & b & r & a & \$ & a & b & r & a & c & a & d \\
b & r & a & \$ & a & b & r & a & c & a & d & a \\
r & a & \$ & a & b & r & a & c & a & d & a & b \\
a & \$ & a & b & r & a & c & a & d & a & b & r \\
\$ & a & b & r & a & c & a & d & a & b & r & a
\end{bmatrix}$$

$$\widetilde{M}(x) = \begin{bmatrix}
\underline{a} & \underline{b} & \underline{r} & \underline{a} & \underline{c} & \underline{a} & \underline{d} & \underline{a} & \underline{b} & \underline{r} & \underline{a} & \underline{\mathbf{\$}} \\
a & b & r & a & \$ & a & b & r & a & c & a & \mathbf{d} \\
a & c & a & d & a & b & r & a & \$ & a & b & \mathbf{r} \\
a & d & a & b & r & a & \$ & a & b & r & a & \mathbf{c} \\
a & \$ & a & b & r & a & c & a & d & a & b & \mathbf{r} \\
b & r & a & c & a & d & a & b & r & a & \$ & \mathbf{a} \\
b & r & a & \$ & a & b & r & a & c & a & d & \mathbf{a} \\
c & a & d & a & b & r & a & \$ & a & b & r & \mathbf{a} \\
d & a & b & r & a & \$ & a & b & r & a & c & \mathbf{a} \\
r & a & c & a & d & a & b & r & a & \$ & a & \mathbf{b} \\
r & a & \$ & a & b & r & a & c & a & d & a & \mathbf{b} \\
\$ & a & b & r & a & c & a & d & a & b & r & \mathbf{a}
\end{bmatrix}$$

Figure 3.3: Example of the BWT for the sequence $x =$ abracadabra$

Zero run length encoding

Zero is a dominant symbol in the sequence x^{mtf}. For some sequences x taken from the standard set of data compression test files, the Calgary corpus [20], the percentage of zeros in the sequence x^{mtf} may reach 90%. On the average, this sequence contains 60% zeros. Hence, there are many long runs in the sequence x^{mtf} consisting of zeros, so called *0-runs*. This may lead to some difficulties in the process of efficient probability estimation. Therefore a *zero run length* (RLE-0) transform was suggested by Wheeler (not published but reported by Fenwick [67]) to treat 0-runs in a special way. Note that the RLE-0 transform was not introduced in the original work by Burrows and Wheeler [39] but we describe it because of its usefulness. Figure 3.5 contains the symbols assigned by the RLE-0 transform to some integers. A general rule for computation of the RLE-0 code for the integer m is to use a binary representation of length $\lfloor \log(m+1) \rfloor$ of the number $m - 2^{\lfloor \log(m+1) \rfloor} + 1$, and substitute all 0s with 0_a and

L	a b c d r	d a b c r	r d a b c	c r d a b	r c d a b	a r c d b	a r c d b	a r c d b	b a r c d	b a r c d	a b r c d
x^{bwt}	d	r	c	r	a	a	a	a	b	b	a
x^{mtf}	**3**	**4**	**4**	**1**	**3**	**0**	**0**	**0**	**4**	**0**	**1**

Figure 3.4: Example of the move-to-front transform

0-run length	RLE-0 code
1	0_a
2	0_b
3	0_a0_a
4	0_a0_b
5	0_b0_a
6	0_b0_b
7	$0_a0_a0_a$
8	$0_a0_a0_b$
9	$0_a0_b0_a$
...	...

Figure 3.5: Example of the RLE-0 transform

all 1s with 0_b. A more detailed description of the RLE-0 was presented by Fenwick [67]. Applying the RLE-0 transform results in the sequence $x^{\text{rle-0}}$ over the alphabet $\mathcal{A}^{\text{rle-0}} = (\mathcal{A}^{\text{mtf}} \setminus \{0\}) \cup \{0_a, 0_b\}$. Experimental results indicate [16] that the application of the RLE-0 transform indeed improves the compression ratio.

For our sample sequence, the gain of using the RLE-0 cannot be demonstrated since the sequence is too short. After computing this transform we arrive at the result $x^{\text{rle-0}} = 344130_a0_b40_a1$.

Entropy coding

In the last stage of the BWCA, the sequence $x^{\text{rle-0}}$ is compressed using a universal entropy coder, which could be for example the Huffman or the arithmetic coder. In the sequel, we are going to discuss how the probability estimation is made for the sequence $x^{\text{rle-0}}$. Now we only emphasise that the first two stages of the BWCA (BWT and MTF) do not yield any compression at all. They are only transforms which preserve the sequence length. The third stage, RLE-0,

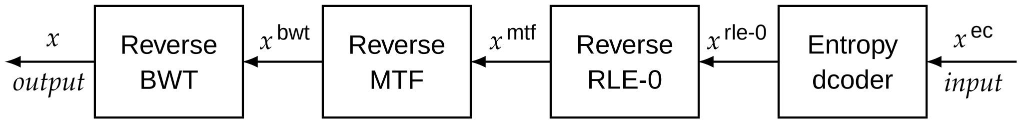

Figure 3.6: Burrows–Wheeler decompression algorithm

	a	d	r	c	r	a	a	a	b	b	a
	b	a	d	r	c	r	r	r	a	a	b
L	c	b	a	d	d	c	c	c	r	r	r
	d	c	b	a	a	d	d	d	c	c	c
	r	r	c	b	b	b	b	b	d	d	d
x^{mtf}	3	4	4	1	3	0	0	0	4	0	1
x^{bwt}	**d**	**r**	**c**	**r**	**a**	**a**	**a**	**a**	**b**	**b**	**a**

Figure 3.7: Example of the reverse move-to-front transform

gives some compression but its main task is to simplify the problem of probability estimation if a single symbol has a large frequency of occurrence. The proper compression is made only in the last stage, called entropy coding. The first three stages transform the input sequence, x, to the form for which the probability estimation can be effectively and simply determined.

3.1.2 Decompression algorithm

We have completed the discussion of the basis of the compression algorithm. Now let us describe how the decompression algorithm works. Its basic scheme is presented in Figure 3.6. The discussion will follow the order of running the stages of the decompression process.

The entropy decoder works similarly to the entropy coder, and it is ignored in this section. Let us only remark that the probability estimation is identical in both the entropy coder and the decoder (what is necessary for proper decoding).

The reverse RLE-0 stage is very simple. The only thing we need to do is to substitute the components of 0_as and 0_bs with the equivalent 0-runs. Let us assume that we decompress the same sequence which we have compressed in the previous sections. Therefore the $x^{\text{rle-0}}$ sequence is $344130_a0_b40_a1$. After the reverse RLE-0 transform we obtain $x^{\text{mtf}} = 34413000401$.

The reverse MTF stage is also simple and similar to the MTF stage. While processing the sequence x^{mtf}, we output the symbol that is at the position x_i^{mtf} at the list L and next we move the symbol to the beginning of the list L. Figure 3.7 contains an example. As a result of this stage we obtain $x^{\text{bwt}} =$ drcraaaabba.

$$\widetilde{M}(x) = \begin{bmatrix} \underline{a} & \cdots & \underline{\$} \\ \mathsf{a} & \cdots & \mathsf{d} \\ \mathsf{a} & \cdots & \mathsf{r} \\ \mathsf{a} & \cdots & \mathsf{c} \\ \mathsf{a} & \cdots & \mathsf{r} \\ \mathsf{b} & \cdots & \mathsf{a} \\ \mathsf{b} & \cdots & \mathsf{a} \\ \mathsf{c} & \cdots & \mathsf{a} \\ \mathsf{d} & \cdots & \mathsf{a} \\ \mathsf{r} & \cdots & \mathsf{b} \\ \mathsf{r} & \cdots & \mathsf{b} \\ \$ & \cdots & \mathsf{a} \end{bmatrix}$$

Row number	1	12	5	11	7	2	9	4	8	3	10	6
Last symbol in row	$	a	r	b	a	d	a	c	a	r	b	a
No. of symbol occurrences	1	5	2	2	2	1	4	1	3	1	1	1
New row number	12	5	11	7	2	9	4	8	3	10	9	1

Figure 3.8: Example of the reverse Burrows–Wheeler transform

A more interesting is the reverse Burrows–Wheeler transform. It is based on the observation that the sequence x^{bwt} is a permutation of the sequence x and sorting it yields the first column of the matrix $\widetilde{M}(x)$ which contains the first characters of the contexts by which the matrix $\widetilde{M}(x)$ is sorted. Therefore, we can find a symbol c located in the first column of ith row of the matrix $\widetilde{M}(x)$, using the symbol x_i^{bwt}. Knowing that this is the jth occurrence of symbol c in the first column of the matrix $\widetilde{M}(x)$ we find its jth occurrence in the last column. Moreover, the symbol c precedes the symbol x_i^{bwt} in the sequence x. Thus, if we know $R(x)$, then we also know the last character of the sequence x, i.e., $x_{R(x)}^{\mathrm{bwt}}$. Starting from this character, we can iterate in a described manner to restore the original sequence x in time $O(n)$.

Let us now return to the sample sequence and take a look at Figure 3.8, presenting the computation of the reverse BWT. Knowing the value $R(x)$, we insert the sentinel character at the appropriate position in the sequence x^{mtf}. At the beginning, we find the first and the last columns of the matrix $\widetilde{M}(x)$. Now the reverse Burrows–Wheeler transform starts from the row number $R(x) = 1$, whose last character is $. We check that this first occurrence of $ is in the last column, and we find the first occurrence of $ in the first column, which happens to be in the 12th row. Then we find the last symbol in this row, a, and notice that this is the 5th occurrence of this symbol in the last column. Thus we look for the 5th occurrence of a in the first column finding it in the 5th row. This procedure is repeated until the entire sequence x^{-1} is retrieved. Reversing the sequence

$x^{-1} =$ $arbadacarba, we obtain the sequence $x =$ abracadabra$. After removing the sentinel, we arrive at the sequence we have compressed.

3.2 Discussion of the algorithm stages

3.2.1 Original algorithm

So far, we have introduced the fundamentals of the Burrows–Wheeler compression algorithm. The BWCA was presented by the authors without relating it to the classes of sources. Over the years, the understanding of the properties of the Burrows–Wheeler transform was progressing. A number of researchers proposed also many improvements to the original work of Burrows and Wheeler. Now we take a closer look at them.

3.2.2 Burrows–Wheeler transform

Burrows–Wheeler computation method

Burrows and Wheeler [39] presented a method for the BWT computation based on sorting. Their approach was a direct implementation of the way of the BWT computation, which is presented in Section 3.1.1. Even though it is efficient enough in most practical cases, its worst-case time complexity is $O(n^2 \log n)$. Burrows and Wheeler also suggested using a suffix tree to calculate the transform faster. To this end a sentinel, $, is appended to the sequence x. For the sequence x ended by $, the problem of sorting the cyclic shifts reduces to the problem of suffix sorting. The latter problem can be solved by building a suffix tree and traversing it in the lexicographic order. The time of the traversing is linear to the length of the sequence.

Methods for suffix tree construction

A sample suffix tree for the sequence abracadabra$ is presented in Figure 3.9. There are several efficient methods for building it. First such an approach was presented by Weiner [186]. This was the first method that works in time $O(n)$, which is the minimum possible time complexity order since building a suffix tree requires processing all input symbols. Today the Weiner's method is important only from a historical point of view, because now we know methods that use less space and work faster, but it was a milestone in developing suffix tree methods.

Historically, a second linear-time approach was the method introduced by McCreight [111]. This method is significantly more effective than the Weiner's one, and because of its high efficiency it is often used nowadays. The most important disadvantage of the McCreight's approach is its off-line nature, which

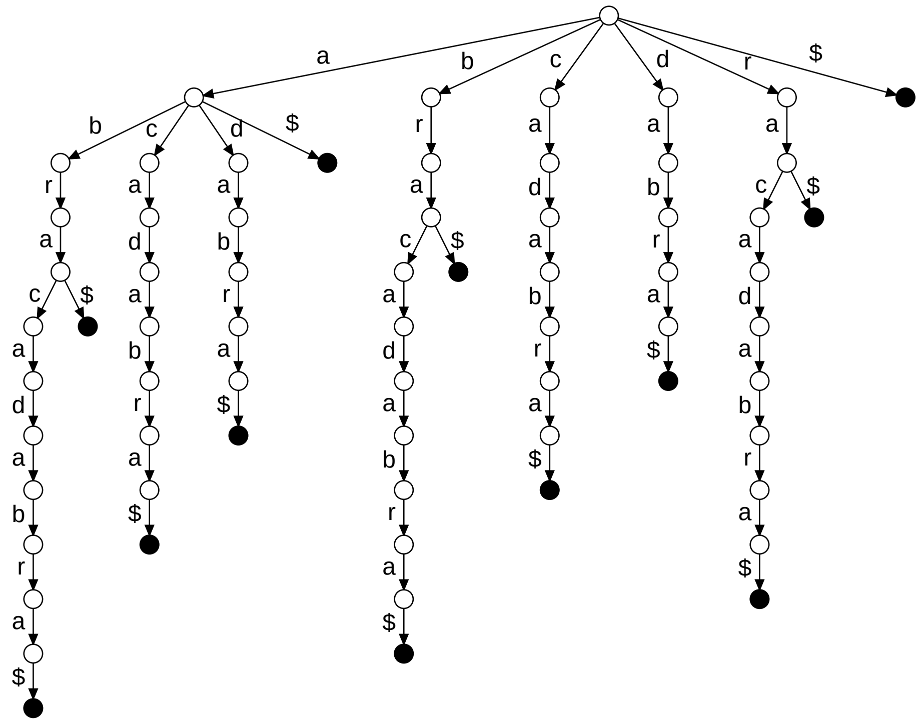

Figure 3.9: Suffix tree for the sequence abracadabra$

means that the whole sequence is to be known before building the suffix tree can start.

In 1995, Ukkonen [173] presented a method which also works in linear time, but is free of the disadvantage of McCreight's approach. His method is an online one, i.e., it builds the suffix tree from the incoming symbols. The Ukkonen's method maintains the proper suffix tree built from the already processed symbols all the time, while the McCreight's one only after processing the whole sequence yields a proper suffix tree. This difference is sometimes important. The advantage of the McCreight's method, however, is its practical efficiency which is better than that for the Ukkonen's method.

As shown by Geigerich and Kurtz [74] all these three methods are related. Recently Kurtz and Balkenhol [96] presented an improved version of the McCreight's method that works fast in practice and has reduced space requirements. Its space complexity is only about $10n$ in the average case and $20n$ in the worst case.

The time complexity of all the described methods is given with an assumption that the alphabet size is small and can be neglected. The method, whose time complexity is optimal when the size of the alphabet cannot be neglected was presented by Farach [63]. The time complexity of his solution is $O(n \log n)$.

Row no.	Start position of suffix	Suffix
1	1	abracadabra$
2	8	abra$
3	4	acadabra$
4	6	adabra$
5	11	a$
6	2	bracadabra$
7	9	bra$
8	5	cadabra$
9	7	dabra$
10	3	racadabra$
11	10	ra$
12	12	$

Figure 3.10: Suffix array for the sequence $x = \text{abracadabra\$}$

Methods for suffix array construction

Constructing a suffix tree is not the only effective method for computing the BWT. The other data structure that can be applied to this task is a suffix array (Figure 3.10). This data structure was introduced by Manber and Myers [106]. The suffix array preserves many properties of the suffix tree. The advantages are: construction time independent of the alphabet size and lower memory requirements. Therefore the suffix array construction methods are often faster in practice than the suffix tree ones. One of its disadvantages is that no direct method of constructing it in linear time is known, and the best methods work in time $O(n \log n)$. (There is a simple method for building the suffix array in time $O(n)$. It is enough to construct the suffix tree in linear time and convert it to the suffix array, what can also be done in linear time. We are interested in direct methods however, because when the suffix trees are employed, we lose the advantage of smaller memory requirements.)

The first method for suffix arrays construction was presented by Manber and Myers [106]. Its worst-case time complexity is $O(n \log n)$ and a space complexity is $8n$. An improved method was proposed by Sadakane [139] and its later version by Sadakane and Larsson [102]. The latest method has the same space and time complexity, but runs much faster in practical applications.

Computation by string-sorting

We mentioned that the original Burrows–Wheeler method for computing the BWT was based on sorting. Bentley and Sedgewick [25] invented for this task a quicksort-based method, which has a worst-case time complexity $O(n^2)$ and an average-case time complexity $O(n \log n)$. Its space requirements equal $5n$ plus the memory needed to store the quicksort's [81] stack. The most recent direct-sorting method was proposed by Seward [149]. The worst-case time complexity of his method is $O(n^2 \log n)$, but in the real applications it works quite fast.

Itoh–Tanaka's method

The method presented recently by Itoh and Tanaka [90] reduces the space requirements to $5n$, but its time complexity strongly depends on the input sequence and in the worst-case is higher than $O(n \log n)$. We describe this method in detail, because it is a point of departure for our research.

To analyse precisely this method let us first define some relations between two sequences:

$$
\begin{aligned}
y <_k z &\iff \exists_{j \le k} \forall_{1 \le i < j} (y_i = z_i \wedge y_j < z_j), \\
y \le_k z &\iff \exists_{j \le k} \forall_{1 \le i < j} (y_i = z_i \wedge y_j \le z_j), \\
y =_k z &\iff \forall_{1 \le i \le k}\, y_i = z_i, \\
y >_k z &\iff \exists_{j \le k} \forall_{1 \le i < j} (y_i = z_i \wedge y_j > z_j), \\
y \ge_k z &\iff \exists_{j \le k} \forall_{1 \le i < j} (y_i = z_i \wedge y_j \ge z_j).
\end{aligned}
\tag{3.1}
$$

The relations define the lexicographic ordering of the k initial symbols of sequences. We say that a sequence s of size n is *decreasing* if $s_{1..n} > s_{2..n}$, and *increasing* if $s_{1..n} < s_{2..n}$.

Itoh and Tanaka propose to split all the sequences being suffixes of the input sequence, x, into two types. A sequence $x_{i..n+1}$ is of:

- type A if $x_{i..(n+1)} >_1 x_{(i+1)..(n+1)}$,
- type B if $x_{i..(n+1)} \le_1 x_{(i+1)..(n+1)}$.

In the first step of the method, the suffixes are sorted according to their first character, using the bucket sorting procedure. Within a bucket, the suffixes of type A are decreasing sequences and they are lower in lexicographic order than all suffixes of type B. In the second step of the Itoh–Tanaka's method, the suffixes of type B are sorted in all the buckets, using a string-sorting procedure. The last step consists of sorting suffixes of type A, what can be done in linear time.

Let us trace the working of the method on the sequence $x =$ abracadabra\$ (Figure 3.11).* The first step is easy. It suffices to bucket sort the set of suffixes and find out which ones are of the type A and which ones are of the type B. In the second step, the suffixes of type B are sorted in each bucket. A vector *ptr*

*In the example, the sentinel character, \$, compares higher than all symbols from the alphabet, to be consistent with the formulation of the suffix trees. In the original paper [90] by Itoh and Tanaka, the sentinel symbol is the lowest character. As this difference is relatively unimportant in the BWCA, we decided to make this modification.

contains the starting indexes of the suffixes in the lexicographic order. In the last step, we are traversing the vector *ptr* from left to right as follows. The element $ptr[1]$ contains the value 1 and as no suffix starts at the earlier position (position 0) we do nothing. Then we see that $ptr[2] = 8$. The symbol x_7 is d, so the suffix $x_{7..(n+1)}$ is of type A and is unsorted. This suffix must be, however, the first one from the suffixes starting from d, so we put $ptr[9] = 7$. The next value of the vector *ptr* is 4. The symbol x_3 is r, so the suffix $x_{3..(n+1)}$ is of type A and is unsorted. Two suffixes start from r, but this one is the lowest, in lexicographic order, so we put $ptr[10] = 3$. In a similar way, we calculate from $ptr[4]$, the value $ptr[8]$, and from $ptr[5]$ the value $ptr[10]$. For $ptr[6] = 2$ we find that $x_1 = \mathsf{a}$, so the suffix $x_{1..(n+1)}$ is of type B and is sorted. Similarly, we do not have to do anything for further elements of the vector *ptr*. When we finish the traversal of this vector, the sorting procedure is completed. We call this method an Itoh–Tanaka's method of order 1, as we split suffixes according to only one symbol.

The advantage of splitting the suffixes into two types is a lower number of sequences to sort. The authors show how their basic method can be improved. They postulate to split the suffixes into buckets according to the following extended rule:

- type A if $x_{i..(n+1)} >_1 x_{(i+1)..(n+1)}$ or $x_{i..(n+1)} >_2 x_{(i+2)..(n+1)}$,
- type B otherwise.

To make this improvement possible, we have to divide the suffixes into buckets according to their two initial symbols. We call this version an Itoh–Tanaka's method of order 2. It is possible to extend this rule using also the relation $>_3$, but it would entail the usage of 2^{24} buckets, so it could be useful only for very long sequences (of size significantly exceeding 2^{24}).

The memory requirements of this method depend on the memory needed by the sorting procedure. Similarly, the time complexity is determined by the sorting step.

Summary of transform computation methods

It does not matter whether we employ a suffix tree construction method, a suffix array construction method, or yet another method for computation the sequence x^{bwt}. We need only to compute the last column of the matrix $\widetilde{M}(x)$, what can be accomplished in many ways. Figure 3.12 compares the methods used in practical applications for computing the BWT.

Transform relation to the context tree sources

At the first glance, the BWCA seems significantly different from the PPM algorithm. Cleary *et al.* [49, 50] shown however that it is quite similar to the PPM*

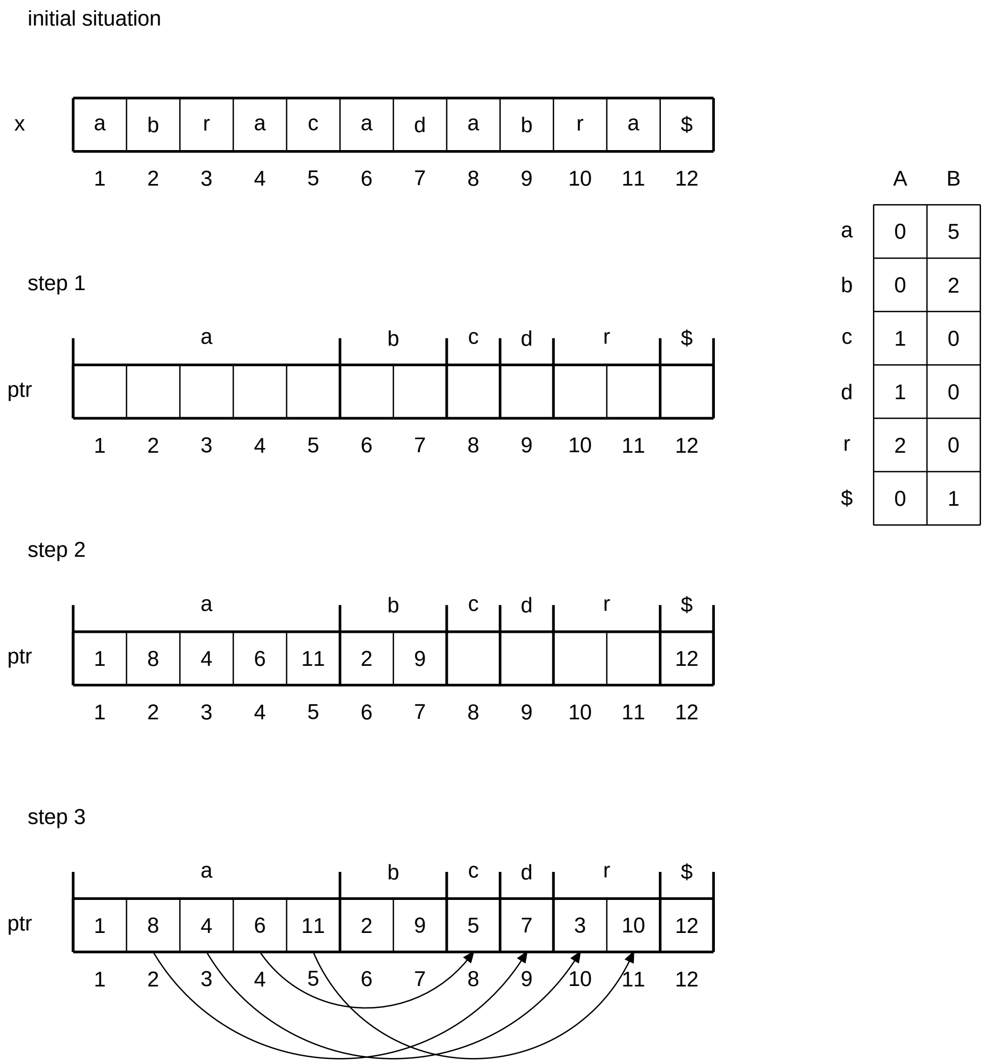

Figure 3.11: Example of working of the Itoh–Tanaka's method of order 1 for the sequence $x =$ abracadabra$

compression method. A more precise relation of the BWT to the context tree sources was shown by Balkenhol and Kurtz [16]. We discuss here this relation following their work.

Let us suppose that the sequence x was produced by a context tree source ω containing the set $\mathcal{S}$ of contexts s. Let us consider any context $s \in \mathcal{S}$. The sequence x contains some number, s, of components. We define the set $\mathcal{X}(s) = \{j_1, j_2, \ldots, j_w\}$ of positions where such components start. In the first step of the BWT computation method, a matrix $M(x)$ is formed of all cyclic shifts of the sequence x. The prefixes of the j_1th, $\ldots$, j_wth rows are s. There are no other rows with such a prefix and all these rows are grouped together after the sorting step. Since no assumption is made of what context has been chosen, we see that the

Method	Worst-case time complexity	Avg.-case time complexity	Avg.-case space complexity
Ukkonen's suffix tree construction	$O(n)$	$O(n)$	NE
McCreight's suffix tree construction	$O(n)$	$O(n)$	NE
Kurtz–Balkenhol's suffix tree construction	$O(n)$	$O(n)$	$10n$
Farach's suffix tree construction	$O(n \log n)$	$O(n \log n)$	NE
Manber–Myers's suffix array construction	$O(n \log n)$	$O(n \log n)$	$8n$
Sadakane's suffix array construction	$O(n \log n)$	$O(n \log n)$	$9n$
Larsson–Sadakane's suffix array construction	$O(n \log n)$	$O(n \log n)$	$8n$
Itoh–Tanaka's suffix array construction	$> O(n \log n)$	NE	$5n$
Burrows–Wheeler's sorting	$O(n^2 \log n)$	NE	NE
Bentley–Sedgewick's sorting	$O(n^2)$	$O(n \log n)$	$5n +$ stack
Seward's sorting	$O(n^2 \log n)$	NE	NE

Figure 3.12: Comparison of methods for the BWT computation

BWT groups together identical contexts in the matrix $\widetilde{M}(x)$. Therefore, all the characters preceding each context occupy successive positions in the x^{bwt} sequence. Such a component of the sequence x^{bwt} which is composed of symbols appearing in one context, is called a *CT-component*. Typically, the context tree source is defined in such a way that the symbols succeeding the contexts depend on it. In this situation, we talk about *preceding contexts*. In the BWT, we have *successive contexts* because the characters that precede the contexts are considered. It seems that this is a large difference, but let us notice that the reverse sequence, x^{-1}, provides the relation to the preceding contexts.

In the PPM compression algorithms, we choose a context which depends on the previous characters. We do not know, however, how long the current context is, because we do not know the context tree source which produced the sequence x. In the BWCA, we also do not know the context but we know that all the identical contexts are grouped. The main disadvantage of the BWT is that the sequence x^{bwt} contains no information when context changes. We even do not know when the first character of the context switches. This disadvantage is important, since the probability distribution of symbol occurrence in one context is determined by the parameters of the context tree source and does not change, but in the another context this probability differs significantly. Typically, the probability distribution of similar context is also similar. There are no grounds for such an assumption in the context tree source model, but this similarity is very frequent in real sequences. The length of the common prefix of two contexts is shorter, the difference of probability distribution is usually larger. The PPM scheme can exploit this phenomenon, because it knows this length. In the BWCA, however, we do not have such a precise knowledge. We only know that the CT-components related to similar contexts appear in the sequence x^{bwt} closely to each other.

There were a number of other theoretical analyses of the Burrows–Wheeler transform and similar transforms. A transform closely related to the BWT, the *lexical permutation sorting transform*, was introduced by Arnavut and Magliveras [10, 11]. Other theoretical analysis of the BWT-based algorithms was presented by Effros [56]. The author has shown the properties of the BWT output assuming the FSMX source model. Manzini [107, 108, 109] also discussed the properties of the BWT and has proven a bound of the compression ratio. Other interesting works on the properties of the BWT were presented by Arimura and Yamamoto [6, 7].

3.2.3 Run length encoding

The *run length encoding* (RLE) [75] was absent in the original implementation of the BWCA. Some researchers postulate, however, its usage before the BWT, so we describe here the RLE and present the arguments in favour of its inclusion into a BWT-based compression algorithm.

The main idea of the RLE is simple—the runs of length longer or equal to some fixed number, usually 3, are replaced by a pair: the current symbol and the number of its repetitions. This transform rarely improves the compression ratio significantly, but this is not the only motivation of using the RLE. Since the sequence x may contain many long runs, the time of the successive stages (especially, the computation of the Burrows–Wheeler transform) can be quite long. To overcome this problem various solutions were suggested. Burrows and Wheeler [39] introduced a special sorting procedure for such sequences. Fenwick [66, 68] used RLE, which eliminates long runs and decreases the length of the sequence x. The main advantage is reducing the time of computing the BWT.

Using the RLE destroys some contexts. In most cases, the RLE slightly worsens the compression ratio, by about 0.1%. If the sequence x contains many long runs, then it turns out that the sequence x^{mtf} comprises 90% of 0s. In such a case, the RLE allows for reducing substantially the number of zeros in the sequence x^{mtf} and better estimating the probability of non-zero symbols, thus improving the overall compression ratio. Taking this observation into account, Balkenhol and Kurtz [16] suggested that the RLE should be used only when it makes the length of the sequence $x^{\text{rle-0}}$ to be less than $0.7n$.

Note that if the method for the BWT computation based on constructing a suffix tree or a suffix array is used, then the occurrences of runs do not significantly influence its execution time. If the RLE-0 transform is employed, then the problem of improper probability estimation in the entropy coder also becomes insignificant. Therefore we conclude that the RLE should not be used.

3.2.4 Second stage transforms

Structure of the Burrows–Wheeler transform output sequence

As we mentioned, the sequence x^{bwt} is a concatenation of components corresponding to the separate contexts. Unfortunately, there is no information on where exactly each such a CT-component starts in the sequence x^{bwt}. By monitoring the probability distribution in the sequence x^{bwt}, however, we can try to uncover some of this information [18, 99, 101]. Surely the sequence x^{bwt} is a permutation of the sequence x. In order to exploit the properties of the sequence x^{bwt}, we have to transform its local structure into a global structure in some way. Alternatively, we have to find a way to rapidly adapt to a changing probability distribution in the sequence x^{bwt} without information on where the CT-components start. Several approaches have been proposed to solve this problem. Some of them are based on the observation that the problem is similar to the list update problem.

List update problem

The formulation of the *list update problem* (LUP) [110] states that there is a list of items and a sequence of requests. A request can be an insertion, a deletion, or an access to an item in the list. The method solving the problem must serve these requests. The cost of serving an access request to an item p on the ith position from the front of the list equals i, which is the number of comparisons needed to find p. After processing a request, the list can be reorganised in order to minimise the total cost of its maintenance. Once an item p is accessed, it may be moved free of charge to any position closer to the front of the list (*free transpositions*). Other transpositions, of elements located closer than p to the end of the list are called *paid*, and their cost equals 1. The methods solving the LUP should minimise the total cost of maintaining the list, i.e., the sum of all costs of serving the sequence of requests.

There are two main classes of on-line methods solving the LUP: deterministic and randomised. Here we present the theoretical bounds of their performance. Before we can establish these bounds, we need to introduce a new term. We say that a deterministic method A is *c-competitive* if there is a constant α such that

$$A(\sigma) - c \cdot OPT(\sigma) \leq \alpha, \tag{3.2}$$

for all possible sequences of requests σ. The $A(\cdot)$ denotes the total cost of maintenance performed by the method A, and $OPT(\cdot)$ is the total cost of maintenance done by the *optimal off-line algorithm*. Similarly we define the c-competitiveness for the randomised method as

$$E(A(\sigma)) - c \cdot OPT(\sigma) \leq \alpha, \tag{3.3}$$

where $E(\cdot)$ is the expected value taken with respect to the random choices made by the considered method.

The optimal off-line method knows the whole sequence of requests and can serve it with the minimal cost. The time complexity of such a method is exponential, what disqualifies it in practice. The best known optimal off-line method, proposed by Reingold and Westbrook [132], runs in time $O(l2^n(n-1)!)$, where n is the number of requests and l is the number of elements on the list L.

There are bounds of the competitiveness of the deterministic as well as randomised methods. Raghavan and Karp (reported by Irani [89]) proved that the lower bound for deterministic methods is $c = 2 - 2/(l+1)$. The randomised methods can improve this bound, and the best known lower bound for them is $c = 1.5$ [168]. The best known randomised method achieves $c = (1+\sqrt{5})/2 \approx 1.62$ [3].

A recent review of many methods for the LUP was presented by Bachrach and El-Yaniv [15]. The authors provide a brief description of over forty methods giving their proven competitiveness. They also compare these methods empirically.

In the second stage of the BWCA, we do not want to minimise the total cost of maintenance of the list L. When we apply the method solving the LUP to the list L and the sequence of requests x^{bwt}, we obtain the sequence x^{lup} which contains the integers being the number of positions on which the symbols from the sequence x^{bwt} appear in the list L. If we sum up the numbers from the sequence x^{lup}, we get the total cost of maintaining the list L (it is assumed here that the method for the LUP does not perform paid transpositions). The main goal of the second stage of the BWCA is not to minimise the total cost, even though typically when this cost is smaller, the distribution of probabilities of symbol occurrences in the sequence x^{lup} is less uniform. In the last stage of the BWCA, we apply the entropy coder to the sequence x^{lup}, and when the distribution of the symbols in this sequence is less uniform, then a better compression ratio is achieved. This observation is of course imprecise because the compression ratio depends on the probability distribution of symbols in the sequence x^{lup} in a more complicated way. When we use the RLE-0 transform, we in fact encode the sequence $x^{\text{rle-0}}$, whose probability distribution of symbols is slightly different. The overall compression ratio depends also on the probability estimation in the last stage of the BWCA. Nevertheless, minimising the total cost of maintaining the list L typically yields a better compression ratio.

Let us notice that the additional cost of paid transpositions can be neglected because the symbols in the sequence x^{lup} correspond to the cost of finding the current items, and there is no cost related to the reorganisation of the list L. In particular, we can say that in the BWCA the problem is similar to the modified LUP, in which the paid transpositions cost 0. The sequence x^{bwt} has also a specific structure, as it is composed of CT-components. In the LUP, we do not assume anything about the sequence of requests. Therefore, using the best methods specialised for the LUP does not always lead to the best compression results. Several modifications of these methods were proposed in order to exploit the properties of the BWT in a better way.

Move-to-front transform and its modifications

In the work introducing the BWCA [39], Burrows and Wheeler suggested using the move-to-front transform [26] as the second stage of the compression algorithm. The MTF transform is a method solving the LUP (it meets the lower bound for the deterministic methods), and maintains a character list L. When a character x_i^{bwt} appears, the list L is scanned and the position of the character x_i^{bwt} in the list L is assigned to x_i^{mtf}. Then the character is moved to the beginning of the list L. As a result we get the sequence x^{mtf} over the alphabet $\mathcal{A}^{\text{mtf}} = \{0, 1, \ldots, k-1\}$. This is a very simple strategy, but its results are quite good in practice. The usage of the MTF transform in the BWCA is motivated by the observation that the most likely symbols to appear are the most recent ones. This is because the more recent the last occurrence of the character is, the more likely it is in a different CT-component.

Burrows and Wheeler [39] suggested that it may be useful to refrain from moving the current character to the very first position of the list. Fenwick [65, 67, 68] and Schindler [144] explored such a possibility, but failed to obtain better compression results. Recently, Balkenhol *et al.* [17] proposed an improvement of the MTF called MTF-1, which improves the compression ratio. Its only modification of the MTF transform is that only the symbols from the second position in the list L are moved to top of the list. The symbols from the higher positions are moved to the second position. Balkenhol and Shtarkov [18] proposed a further modification of the MTF-1—the symbols from the second position are moved to the beginning of the list L only if the previous transformed symbol is at the first position (following the authors we call this version MTF-2). The illustration and the comparison of the MTF, the MTF-1, and the MTF-2 transforms is shown in Figure 3.13.

MTF	L	a	d	r	c	r	a	a	a	a	b	a
		b	a	d	r	c	r	r	r	r	a	b
		c	b	a	d	d	c	c	c	c	r	r
		d	c	b	a	a	d	d	d	d	c	c
		r	r	c	b	b	b	b	b	b	d	d
	x^{bwt}	d	r	c	r	a	a	a	a	b	b	a
	x^{mtf}	**3**	**4**	**4**	**1**	**3**	**0**	**0**	**0**	**4**	**0**	**1**

MTF-1	L	a	a	a	a	a	a	a	a	a	a	b
		b	d	r	c	r	r	r	r	r	b	a
		c	b	d	r	c	c	c	c	c	r	r
		d	c	b	d	d	d	d	d	d	c	c
		r	r	c	b	b	b	b	b	b	d	d
	x^{bwt}	d	r	c	r	a	a	a	a	b	b	a
	$x^{\text{mtf-1}}$	**3**	**4**	**4**	**2**	**0**	**0**	**0**	**0**	**4**	**1**	**1**

MTF-2	L	a	a	a	a	a	a	a	a	a	a	a
		b	d	r	c	r	r	r	r	r	b	b
		c	b	d	r	c	c	c	c	c	r	r
		d	c	b	d	d	d	d	d	d	c	c
		r	r	c	b	b	b	b	b	b	d	d
	x^{bwt}	d	r	c	r	a	a	a	a	b	b	a
	$x^{\text{mtf-2}}$	**3**	**4**	**4**	**2**	**0**	**0**	**0**	**0**	**4**	**1**	**0**

Figure 3.13: Comparison of the MTF, MTF-1, and MTF-2 transforms

Other modifications were proposed by Chapin [43], who introduced a Best x of $2x - 1$ transform. Its results are worse than the MTF ones, but he also suggests to use the switching procedure, originally introduced to join two universal compression algorithms in switching method [182, 183] (see also Section 2.7.6), to combine this transform and the MTF-2. The results are only slightly better or comparable to the MTF-2.

Time-stamp transform

One of the best methods for the LUP is *time-stamp* (TS) presented by Albers [3]. The deterministic version of this method—*time-stamp(0)* (TS(0)) [4]—scans for a processed character x_i^{bwt} in the list L and outputs its position. Then it moves the character in front of the first item in the list L which has been requested at most once since the last request of the character x_i^{bwt}. If the current character, x_i^{bwt},

	a	a	a	a	a	a	a	a	a	a	a
	b	b	b	b	b	b	b	b	b	b	b
L	c	c	c	c	r	r	r	r	r	r	r
	d	d	d	d	c	c	c	c	c	c	c
	r	r	r	r	d	d	d	d	d	d	d
x^{bwt}	d	r	c	r	a	a	a	a	b	b	a
x^{ts}	**3**	**4**	**2**	**4**	**0**	**0**	**0**	**0**	**1**	**1**	**0**

Figure 3.14: Example of the time-stamp(0) transform

has not been requested so far, it is left at its actual position.

The TS(0) transform was theoretically analysed by Albers and Mitzenmacher [4] who showed that theoretically the TS(0) is better than the MTF. The authors replaced the MTF transform in the BWCA with the TS(0), but the compression results obtained for the sequences from the Calgary corpus [20] were worse than those obtained with the MTF. The authors, however, do not provide explicit details of their experiments.

Inversion frequencies transform

A completely new approach to the problem of transforming the sequence x^{bwt} to a form which can be better compressed by an entropy coder was proposed by Arnavut and Magliveras [10]. They described a transform called *inversion frequencies* (IF). This transform does not solve the list update problem. Instead, it forms a sequence x^{if} over an alphabet of integers from the range $[0, n-1]$. For each character a_j from the alphabet $\mathcal{A}$, the IF transform scans the sequence x^{bwt}. When it finds the occurrence of the character a_j, it outputs an integer equal to the number of characters greater than a_j that occurred since the last request to the character a_j. This sequence, however, is not sufficient to recover the sequence x^{bwt} correctly. We also need to know the number of occurrences of each character from the alphabet in the sequence x^{bwt}. This disadvantage is especially important for short sequences.

The example of working of the IF transform is shown in Figure 3.15. The bottom line in the row denoted by x^{if} is only for better understanding and the complete result of the IF is the sequence $x^{\mathrm{if}} = 5211240002402000$.

An efficient forward implementation for the inversion frequencies was presented by Kadach [92] as well as the reverse transform. The author introduced that both transforms can work in time $O(n \log k)$.

x^{bwt}	d r c r a a a a b b a									
x^{if}	5	2	1	1	2	40002	40	2	0	00
	a	b	c	d	r	a	b	c	d	r

$$x^{\text{if}} = 5211240002402000$$

Figure 3.15: Example of the inversion frequencies transform

Distance coding transform

Recently a new proposition for the second stage, *distance coding* (DC), was suggested by Binder. This transform has not been published yet, and we describe it according to References [27, 29, 77]. For each character x_i^{bwt}, the DC finds its next occurrence in the sequence x^{bwt}, which is x_p^{bwt}, and outputs the distance to it, i.e., $p - i$. When there is no further occurrence of x_i^{bwt}, the DC outputs 0. To establish the sequence x^{bwt} correctly, we also have to know the first position of all the alphabet characters. The basic version of the DC, described above, is in fact a small modification of the *interval encoding* proposed by Elias [60].

To improve the compression ratio, Binder proposed three modifications of the basic transform. First, we can notice that in the sequence x^{dc} some of the ending zeroes are redundant. Second, while scanning the sequence x^{bwt} for the next occurrence of the current character, we may count only the characters that are unknown at this moment. Third, and most important, if the next character is the same as the current character, we do not need to encode anything, and we can simply proceed to the next character. The example of work of the DC is shown in Figure 3.16. The three improvements are also shown. The output of the DC is the sequence being the concatenation of 59312 (the initial positions of the symbols) and 01002 which is the output of the transform.

Balkenhol–Shtarkov's coding of the move-to-front transform output

Recently Balkenhol and Shtarkov [18] proposed a new approach to the coding of the sequence x^{bwt}. Their method is based on the observation that the entropy coder codes the sequence over the alphabet $\mathcal{A}^{\text{mtf}}$ (or $\mathcal{A}^{\text{rle-0}}$) consisting of integers. For typical sequences x, the probability distribution of the integers in the sequence x^{mtf} decreases monotonically for larger integers. Unfortunately, this distribution varies in the sequence and—what is worse—for a given integer, we do not know which character it represents. For example, two identical integers greater than 1 that appear at successive positions in the sequence x^{mtf} represent different characters.

DC ver. 0	x^{bwt}	d	r	c	r	a	a	a	a	b	b	a
	x^{dc}	0	2	0	0	1	1	1	3	1	0	0

DC ver. 1	x^{bwt}	d	r	c	r	a	a	a	a	b	b	a
	x^{dc}	0	2	0	0	1	1	1	3	1		

DC ver. 2	x^{bwt}	d	r	c	r	a	a	a	a	b	b	a
	init. known pos.	d	r	c	.	a	.	.	.	b	.	.
	x^{dc}	0	1	0	0	1	1	1	2	1		

DC ver. 3	x^{bwt}	d	r	c	r	a	a	a	a	b	b	a
	init. known pos.	d	r	c	.	a	.	.	.	b	.	.
	x^{dc}	0	1	0	0				2			

$$x^{\text{dc}} = 5931201002$$

Figure 3.16: Example of the distance coding transform

Balkenhol and Shtarkov suggest dividing the sequence x^{mtf} into two sequences.[†] The first sequence, $x^{\text{mtf},1}$, is over the alphabet $\mathcal{A}^{\text{mtf},1} = \{0, 1, 2\}$. It is constructed from the sequence x^{mtf} by replacing all occurrences of integers greater than 1 with 2. This means that the ternary sequence $x^{\text{mtf},1}$ holds only the information whether the transformed character was at the first (integer 0), the second (integer 1), or at some other position (integer 2) in the list L. The second sequence, $x^{\text{mtf},2}$, is over the alphabet $\mathcal{A}$. It is constructed from the sequence x^{bwt} by removing the characters for which the integers 0 or 1 appear in the sequence $x^{\text{mtf},1}$.

3.2.5 Entropy coding

Preliminaries

The last stage in the BWCA is the entropy coding. As we noticed, the first stage of the BWCA transforms the input sequence to the form from which the probability of next symbols occurrence can be effectively calculated. The entropy coding is the stage where the proper compression is made.

The BWT transforms the input sequence to the sequence composed of CT-components. It seems that the BWT removes any contextual information and usage of the higher orders may introduce additional redundancy because of the

[†]In fact, Balkenhol and Shtarkov use the MTF-2 transform instead of the MTF transform and the sequence $x^{\text{mtf-2}}$.

MTF symbol	Prefix code
0	0
1	10
2	110
3	1110
4	$1111\langle 4\rangle$
...	...
255	$1111\langle 255\rangle$

Figure 3.17: Encoding method of symbols in Fenwick's Shannon coder

larger number of contexts, in which probability is estimated. Therefore most authors propose using a simple order-0 coder. If not stated otherwise, such coders are used in the solutions described below.

Burrows and Wheeler's proposition

The very first proposition by Burrows and Wheeler [39] was to use the Huffman coder as the last stage. The Huffman coder is fast and simple, but the arithmetic coder is a better choice if we want to achieve better compression ratio. Nowadays, best compression ratios in the family of the BWT-based algorithm with the Huffman coder as the last stage can be obtained by the *bzip2* program by Seward [150].

Fenwick's Shannon coder

The first usage of the arithmetic coder in the BWCA was in Fenwick's works. He investigated several methods of probability estimation. His first proposal was *Shannon encoder* [68]. Each symbol x_i^{mtf} of the sequence x^{mtf} is transformed to the prefix code with respect to the rules presented in Figure 3.17. The code is composed of at most four binary numbers and, if necessary, a number of range $[4, 255]$ when the prefix is 1111. Each of the bits of the binary numbers is then encoded with a binary arithmetic coder in the separate contexts. Moreover, the first binary digit is encoded in one of two different contexts depending on whether x_{i-1}^{mtf} is zero or not. The last part of the code, an integer from the range $[4, 255]$, is encoded using the arithmetic coder in one context.

Fenwick's structured coder

The second interesting method of probability estimation invented by Fenwick is a *structured coding* model [68]. The observation that turned Fenwick to this solution was that the frequencies of occurrences of different symbols in the x^{mtf} se-

MTF symbol	Number of entries in the group
0	1
1	1
2–3	2
4–7	4
8–15	8
16–31	16
32–63	32
64–127	64
128–255	128

Figure 3.18: Grouping method of symbols in Fenwick's structured coder

quence can differ by four or five orders of magnitude for input sequences from real world. This causes the difficulties in effective probability estimation of symbol occurrences. The idea is to overcome these difficulties by grouping symbols in classes of similar frequency of occurrence, where the probability can be effectively estimated. Fenwick proposed nine groups of different sizes (Figure 3.18). The encoding of the symbol x_i^{mtf} is then split into two steps. First, we encode the number of group the current symbol belongs to, then, if necessary, we encode the symbols within the group. In this approach, both statistics for all groups and the frequencies of symbol occurrences within the groups do not differ significantly. The results obtained using this model are slightly worse than the ones with the Fenwick's Shannon coder, but employing also the RLE-0 transform and encoding the sequence $x^{\text{rle-0}}$ instead of the sequence x^{mtf}, better results can be obtained.

Rapid changes in the Burrows–Wheeler transform output sequence

We have mentioned before that large differences of frequencies of symbols occurrence are very unfavourable. We have, however, abstained from an in-depth discussion. Now we take a closer look at this problem.

The statistics of fragments of the sequence x^{mtf} placed in short distance from one to the other are similar, because those components are related to the same or similar CT-components in the x^{bwt} sequence. The larger, however, the distance between the symbols in the x^{mtf} sequence, the more likely these symbols come from different CT-components. Therefore the probability of their occurrence differs significantly. One more observation is that when we cross the boundary of successive CT-components, some symbols from far positions in the list L usually introduce to the sequence x^{mtf} a small number of large integers. To solve the problem of effective estimation of these large integers, the statistics of symbol oc-

MTF symbols	Number of entries in the group
$0, 1, 1^+$	3
$2, 2^+$	2
$3, 4, 4^+$	3
$5, 6, 7, 8, 8^+$	5
$9, \ldots 16, 16^+$	9
$17, \ldots, 32, 32^+$	17
$33, \ldots, 64, 64^+$	33
$65, \ldots, 128, 128^+$	65
$129, \ldots, 255$	127

Figure 3.19: Grouping method in a hierarchical coder by Balkenhol *et al.*

currences should be highly adaptive. If, however, the encoder is highly adaptive to tune to rapid changes of local properties of the sequence x^{mtf}, the difference of a few orders of magnitude of symbol frequencies causes many problems and the estimation is rather poor.

Balkenhol–Kurtz–Shtarkov's hierarchical coder

Balkenhol *et al.* proposed a *hierarchical coder* [17] to overcome the problem of large differences between frequencies of small and large integers in the x^{mtf} sequence. They proposed to divide the symbols of range $[0, 255]$ into nine groups. The way of grouping the symbols is presented in Figure 3.19. The encoding proceeds as follows. First, the symbol x_i^{mtf} is encoded within the first group. If the symbol is 0 or 1 that is all, but if the symbol is higher than 1 it is encoded using the second group. Similarly, if the symbol is 2 then it is the last step, and in the other case the third group is used, and so on until the current symbol is completely processed.

One of the main differences between the above-described hierarchical and the Fenwick's structured models is the number of steps needed to encode a symbol. Using the hierarchical model, we may need nine encodings in the worst case while in the structured one we need at most two encodings. Balkenhol *et al.* proposed also using the higher than order-0 arithmetic coder in some contexts. That proposal is justified by the improvements of compression ratios.

Balkenhol and Shtarkov's semi-direct encoding

The proposition of probability estimation introduced by Balkenhol and Shtarkov [18] is a hybrid approach. We have discussed the concept of splitting the sequence x^{mtf} into sequences $x^{\mathrm{mtf},1}$ and $x^{\mathrm{mtf},2}$ in Section 3.2.4, however we have not seen how these sequences are then processed. The sequence $x^{\mathrm{mtf},1}$ is treated as a Markov chain and is encoded using the standard universal coding scheme. To encode the sequence $x^{\mathrm{mtf},2}$, the alphabet is divided into four groups of changing sizes and contents. The symbols from the sequence $x^{\mathrm{mtf},2}$ are encoded with the probability estimation from one of these groups. The groups are maintained to work as sliding windows and contain statistics of symbol occurrences in these windows. Because the last symbols have similar probability distribution, such an estimation helps in obtaining good compression results.

Direct coding of the Burrows–Wheeler transform output

Fenwick examined also one more solution which refrains from usage of the MTF transform. He investigated what could be obtained if the arithmetic coder were employed to encode the x^{bwt} sequence. The main problem in such a situation is that the symbols occurring in successive CT-components can be completely different, and the statistics of symbol occurrences vary rapidly. When we use the MTF, we transform the local structure of the x^{bwt} sequence to the global one, obtaining the sequence x^{mtf} in which the distribution of symbol frequencies decreases roughly monotonically. The sequence x^{bwt} is a permutation of the sequence x, so the distribution of symbol frequencies is much more uniform. These were the reasons why Fenwick [68] obtained significantly worse results.

The approach of refraining from usage of the MTF transform was continued recently by Wirth and Moffat [195]. They employed the idea of the hierarchical coder by Balkenhol and Shtarkov and modified it to work without the MTF stage. Wirth and Moffat propose a probability estimation with exponential forgetting. The authors improve the basic approach using the concepts of exclusion and update exclusion. They also decide to use a higher than order-0 arithmetic coder in some contexts. The obtained results are very good in the group of the BWT-based algorithms that do not use MTF transform or similar second stage methods. The results, however, are still significantly worse than the best ones in the Burrows–Wheeler transform-based algorithms family.

Encoding the distance coder and the inversion frequencies outputs

The MTF transform and related methods are not the only possibility of the second stage in the BWCA. In Section 3.2.4, we mentioned also the inversion frequencies and the distance coder. The sequences x^{if} and x^{dc} consist of integers from the range $[0, n]$, so we need to apply a different approach to the entropy coder when these transforms are used.

Various solutions to encode such numbers may be considered. One of them is to use an exponential hierarchy and to group the integers into sets $\{0\}, \{1\}, \{2,3\}, \{4,\ldots,7\}, \ldots, \{2^{\lfloor \log n \rfloor}, \ldots, n\}$. In the first step, we encode the number of the group containing the current symbol, then we encode the symbol within this group. The other approach is to use a binary prefix code, encode the current symbol using it, and finally encode each bit separately using the binary arithmetic encoder. A good choice for this task can be an Elias γ code [59]. This solution, presented in Reference [55], was invented by the author of the dissertation, and will be discussed also in Section 4.1.6. This approach is reported to be the best one of those examined by Arnavut [9].

The distance coder transform was not published so far and unfortunately we do not know the internals of the algorithms using it, so we could not describe the way of encoding the integers in the compression programs, like the *dc* program [28]. A solution similar to the one used for coding the output of the IF can be, however, used for the output of the DC.

Variable-length integer codes

Recently, an alternative way as the last stage of the BWT-based compression algorithm was proposed by Fenwick [69]. He postulates to use a variable-length integer codes instead of the Huffman or the arithmetic coder if the speed of compression is crucial. Fenwick investigates Elias γ codes [59] and Fraenkel–Klein Fibonacci codes [71]. The gain in the compression speed is, however, occupied by a significant lost in compression ratio.

3.2.6 Preprocessing the input sequence

Why to preprocess the input sequence?

We observed that the BWCA is similar to the PPM algorithms. We also pointed out a significant difference between them—in the BWCA the information where the context is changed is lost after the BWT stage. Here we want to point out the second important difference.

The compression ratio in most data compression algorithms, such as PPM, LZ77, LZ78, or DMC, does not depend on the order of the symbols in the alphabet. We can permute the alphabet and the compression ratio will be only slightly different (theoretically it will be identical). The behaviour of the BWCA is different. In some algorithms, like PPM, we know where the context starts and we can exploit this knowledge. In the BWCA, we do not know this, and in the entropy coding stage, for the probability estimation, we also use symbols from the CT-components close to the current one, what can be harmful.

Alphabet reordering

If we drastically change the ordering of the alphabet symbols, it may cause the contexts grouped together with the current one have significantly different probability distributions. Chapin and Tate [44] showed that if the alphabet ordering is changed randomly, the compression ratio decreases highly (even by 20%).

(We have to stress here that in the whole dissertation we assume the default ASCII ordering of the alphabet.) They suggested that we can try to employ this disadvantage to our benefit and reorganise the alphabet in such a way that the compression ratio will improve.

Chapin and Tate suggested a few approaches—first, an *ad hoc* reordering of the alphabet, grouping vowels and consonants. The other approaches are more complicated. They create histograms of symbols appearing in all one-character contexts. Then they try to reorganise the alphabet in such a way that the total cost of changing the context will be lower. The first problem is to choose which way of finding the difference between two histograms is the best for this task. Chapin and Tate have presented some proposals in this regard. The second problem is how to find the best ordering. They used the algorithms solving the Travelling Salesman Problem (TSP) [103] to find the solution. The similarity to the TSP may by not clear at the first glance. If we notice, however, that we deal with k one-character contexts, the costs of changing to the all other one-character contexts, and we have to produce a permutation of the k contexts, the similarity comes out more clear.

The experimental results obtained by Chapin and Tate are, however, rather disappointing. The gain is significant only when we try all the proposed methods to improve the compression ratio and choose the best one for each single file. It is impractical to compress a file with a dozen or so versions of the transform, and choose the best result if the gain is less than 1%. Moreover, we have to consider also the cost of solving the TSP.

The alphabet reordering was also discussed by Balkenhol and Shtarkov [18]. They modified the ordering proposed by Chapin and Tate. The gain is very small, and the proposed order is adapted to the text files only, so we consider that this artificial reorganisation is unjustified.

Reversing the input sequence

Balkenhol *et al.* [17] suggested that if the number of symbols used in the input sequence is the maximum number of characters which can be represented with an assumed number of bits (typically 256 characters), it should be assumed that the sequence stores a binary file. For such sequences, the succeeding contexts are much better than the preceding ones. Therefore they propose to use in such a case the reversed sequence, x^{-1}. This heuristics gives gains for binary files so we decided to use it in our research to provide a thorough comparison to the best existing algorithms.

Preliminary filtering

Other preliminary improvements (before the BWT) were proposed by Grabowski [76]. His proposals are reversible filters that are used before the compression.

Some of them are independent on the compression algorithm and can be used also with the different compression schemes, not only with the BWCA. All the improvements are based on the assumption that text files are to be compressed.

The first proposition is to use a *capital conversion* what means that the words starting with a capital letter (only the first letter should be a capital) are changed in such a way that the first letter is transformed to the lowercase and a special flag denoting the change is inserted. (Till the end of the current section we will be using the intuitive terms *words* and *letters* to emphasise that we consider only the sequences being texts.) The second improvement is motivated by the observation that most text files contain the lines starting with a letter. This means that the previous character (the context) of this letter is the *end-of-line* (EOL) character. It appears only because of formating rules and typically it has no other meaning. Grabowski proposes to insert an additional space character after the EOL, so the context of the first letter of the word is the space symbol, what is a typical situation. The other proposition is a *phrase substitution*. This idea is not new, also Teahan [163] proposed such a solution to improve the compression ratios. The main idea is to replace the groups of letters with a new symbol. It can be simply done in the text files, because the standard ASCII code uses only 128 first characters of the 256 possibilities. Grabowski provides a list of substitutions which improve the compression ratio. This list was established experimentally, because it is very hard to provide an exact rule which groups of letters to replace in order to improve the compression ratio. Therefore it is clear that for a different set of files other list of substitutions can give better results. We have mentioned only the most important propositions of Grabowski which give significant improvements. These experiments show that by using his modifications we can achieve a gain in the compression ratio about 3% on the text files from the Calgary corpus. This gain is significant and much greater than can be obtained with the alphabet reordering. We have to remember, however, that these improvements are motivated by the assumption that text files are compressed, and applying these modifications to binary files worsens the compression ratio.

Yet another preprocessing method was proposed by Awan and Mukherjee [13, 14]. The authors introduce a reversible *length index preserving transform* (LIPT), based on the dictionary prepared before the compression and the corpus of files, used to order the words in the dictionary. If we have the dictionary, where for each word a special code is prepared, we process the input sequence and replace each known word (the word that belongs to the dictionary) with a related code. Applying the LIPT before the compression gives a gain in the compression ratio of about 5% on the text files from the Calgary corpus. The main disadvantages of the LIPT are: we have to know that we compress a text file, what is the language of the file, and we need a pre-built dictionary with ready-made encodings.

We have mentioned only preprocessors for the text files. There are also filters for different types of binary data. The number of types of binary data is, however, large, and we want only to notice that such filters exist.

Since this dissertation concerns the universal compression algorithms, we do not make any assumptions regarding the type of the compressed sequence. As a consequence we discuss the above data-specific improvements only as a notification of works on the BWCA, but we do not use them in our research.

Chapter 4

Improved compression algorithm based on the Burrows–Wheeler transform

It is a bad plan that admits of no modification.

— PUBLIUS SYRUS
Maxim 469 (42 B.C.)

4.1 Modifications of the basic version of the compression algorithm

4.1.1 General structure of the algorithm

In Chapter 3, we discussed many possibilities of modifications of the basic version of the Burrows–Wheeler compression algorithm. Now we introduce solutions for some of the stages of the algorithm and show how the other stages can be modified to obtain an improved algorithm based on the Burrows–Wheeler transform. Its general structure is presented in Figure 4.1.*

The first stage of the algorithm is the Burrows–Wheeler transform. The BWT output sequence undergoes, in the second stage, a weighted frequency count transform which we introduce in Section 4.1.5. The third stage is the zero run

*The source codes and executable Win32 files for the compression algorithm are available at http://www-zo.iinf.polsl.gliwice.pl/~sdeor/deo03.html.

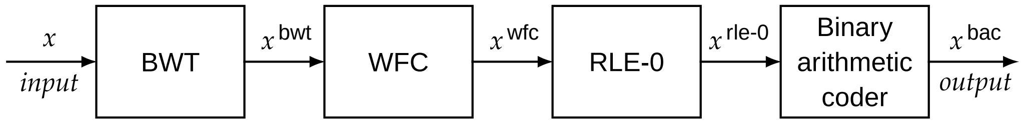

Figure 4.1: Improved compression algorithm based on the BWT

length coding, which reduces the length of 0-runs. In the final stage, the sequence $x^{\text{rle-0}}$ is encoded by a binary arithmetic coder. The arithmetic coding is a well established method for entropy coding, and the most important part of this stage is the probability estimation method. The proposed way of the probability estimation is described in Section 4.1.6.

For the purposes of the comparison, we implemented also the compression algorithm with the inversion frequencies transform as the second stage. In this algorithm, we use a different method of probability estimation in the binary arithmetic coder because the sequence $x^{\text{rle-0}}$ consists of symbols from a different alphabet. The way of the probability estimation is described in Section 4.1.6.

As the dissertation deals with the universal lossless compression, we must not assume too much about the sequence that is compressed. A reasonable choice is to assume that the sequence x is produced by an unknown context tree source. If not stated otherwise, our further discussion is based on this assumption.

4.1.2 Computing the Burrows–Wheeler transform

In Section 3.2.2, we considered the BWT computation methods. The Itoh–Tanaka's approach [90] is one of the methods for building a suffix array. It is fast for most sequences, however, if the sequences contain long runs, its behaviour is poor, because in the sorting step comparisons of such sequences are slow. Here we propose some improvements to the versions of order 1 and 2.

The first improvement is simple, and is motivated by the observation that we always sort suffixes of type B that are not decreasing. There is no obstacle, however, to sort only the suffixes that are not increasing. To this end, it suffices to reverse the lexicographic order of sequences. As the Itoh–Tanaka's method reduces the number of suffixes that are sorted, we can choose the forward or reverse lexicographic order to sort the smaller number of suffixes. This improvement can be applied to both order-1 and order-2 versions.

The second innovation is more complicated and can be applied only to the order-1 version. After the split of the suffixes, the groups of type A contain only decreasing sequences. The groups of type B contain mainly increasing sequences, but also some number of decreasing ones—a part of the sequences starting from two identical symbols, as we cannot easily find out if the sequence starting from the run is increasing or not, without analysing more than two symbols.

In the Itoh–Tanaka's approach, we need only to sort the suffixes that are increasing. In their original work, some suffixes are sorted excessively. Moreover, the suffixes starting from a run of length 2 may not be sorted at all by a string-sorting procedure. We postulate to split the suffixes into three groups:

- type D, if $x_{i..(n+1)} >_1 x_{(i+1)..(n+1)}$,
- type E, if $x_{i..(n+1)} =_1 x_{(i+1)..(n+1)}$,
- type I, if $x_{i..(n+1)} <_1 x_{(i+1)..(n+1)}$.

The suffixes of type I have to be sorted by a string-sorting procedure, and the ordering of the suffixes of type D can be easily discovered during a linear pass over the vector *ptr*. What can we say about the suffixes of type E? Those that are decreasing can be also handled in the traversal over the vector *ptr*, and those that are increasing can be sorted in a special way. Let us suppose that all the suffixes of type I are sorted. Traversing each bucket (storing sequences starting from the same symbol) from right to left, we analyse the suffixes $x_{i..(n+1)}$. For each such a sequence we check if the symbol x_{i-1} is equal to x_i. If it is equal, then the suffix $x_{(i-1)..(n+1)}$ is of type E, and moreover it is the last suffix, in the lexicographic order, in all the unsorted suffixes from this group. Therefore we can put the right position of it to the vector *ptr*, and mark it as sorted. We continue the passing till we analyse sorted suffixes. When this process is finished, all increasing suffixes from the bucket are sorted.

After the author of this dissertation improved the Itoh–Tanaka's approach, he found the work by Tsai–Hsing [172], also presenting a way to improve this method. Tsai–Hsing postulates to split the suffixes into four groups. The main idea is similar, as he also refrains from sorting the suffixes starting from a run of length 2. He divides, before sorting, our group of type E into two groups: first of the increasing sequences, and second of the decreasing sequences.

In our proposition, the preliminary split is unnecessary, as after sorting the suffixes of type I, in a pass from right to left over the bucket, we simply place the increasing suffixes from group of type E. The decreasing suffixes of type E are sorted together with the suffixes of type D in the left-to-right pass over the vector *ptr*.

Owing to this improvements, we always have to sort at most a half of the suffixes. What is more important in practice, the suffixes of type E, which previously were hard to sort, are now easy to process.

4.1.3 Analysis of the output sequence of the Burrows–Wheeler transform

Sequence anatomy

In Section 3.2.2, we mentioned that it is known that the sequence x^{bwt} is composed of the CT-components. This means that the probability of symbol occurrence inside a CT-component is specified by the parameter of the corresponding leaf in the context tree. We, however, did not investigate if something more can be said about the CT-components. The analysis of the output sequence of the BWT we start from proving several lemmas.

Lemma 4.1. *In the non-sentinel version of the BWT, the lowest character of the alphabet in the sequence x is the first character of the sequence x^{bwt} if and only if the sequence is a single run consisting of this character.*

Proof. Let us denote by c the lowest character of the alphabet that appears in the sequence x. The first character of the sequence x^{bwt} is the last character of the first row of the matrix $\widetilde{M}(x)$. The rows of the matrix $\widetilde{M}(x)$ are ordered in the lexicographic order, so no other row precedes the first one in the lexicographic order. What if the last character of the first row is c? The sequence being the one-character-to-right rotation of the first row will be not greater than the first row. The only case when it will not be also lower is when the rotated sequence equals the initial one, which entails that the sequence x is a single run of the character c. □

Lemma 4.2. *In the sentinel version of the BWT, the lowest character of the alphabet in the sequence x cannot be the first character of the sequence x^{bwt}.*

Proof. At the beginning, let us consider what happens if the last character of the first row of the matrix $\widetilde{M}(x)$ is a character, c, which is the lowest character appearing in the sequence x. It cannot happen because the one-character-to-right rotation of the first row will be not greater than this row. It cannot be equal to the first row, for, due to the sentinel, all rotations of the sequence x are different. Therefore, the first character of the x^{bwt} sequence can be the sentinel or some other character greater than c. □

In the above discussion, we did not assume anything on the source that produced the sequence x. Let us now assume that the sequence processed by the BWT is produced by a context tree source.

Lemma 4.3. *The position of symbol occurrence in the CT-component influences the probability of its occurrence.*

Proof. We only want to show that the probability of symbol occurrence is dependent on its position in the CT-component. Therefore, we will consider only one case, for which the calculation is the simplest.

Let us consider the first CT-component of the sequence x^{bwt}. Independently on the parameters of the related context tree leaf, the first character of this component can be the first character appearing in the x sequence, c, with a very small probability in the non-sentinel version of the BWT (confer Lemma 4.1) or with zero probability in the sentinel version of the BWT (confer Lemma 4.2). The characters of the CT-component were produced by the context tree with predefined parameters. These parameters also specify the frequency of symbol occurrences of the all characters in the CT-component. Therefore the probability of occurrence of the character c at some other positions at the CT-component differs from the assumed parameter. □

Corollary 4.4. *At least one CT-component is not an output of a memoryless source with the parameter of the related leaf in the context tree source.*

Proof. From Lemma 4.3 we see that at least in some CT-components the probability of character occurrence depends on its position. This fact contradicts the assumed stationarity of the memoryless source. □

Number of different output sequences of the transform

Let us leave the assumption of the source that produced the input sequence, to investigate the properties of the BWT from an other side. At first, we will find the total number of possible sequences x^{bwt}. To this end we have to prove some lemmas.

Lemma 4.5. *For any two sequences $y_{1..n}$ and $z_{1..n}$, the equality $y^{bwt} = z^{bwt}$ holds if and only if there exists an integer i such that $y_{1..n} = z_{(1+i)..n}z_{1..i}$.*

Proof. For convenience we denote by $M(\cdot)^j$ the sequence in the jth row of the matrix $M(\cdot)$. Let us assume that

$$\exists_i \, y_{1..n} = z_{(1+i)..n}z_{1..i}. \tag{4.1}$$

Therefore

$$M(y)^1 = M(z)^{1+i} \tag{4.2}$$

since, from the formulation of the BWT computation, the $(i+1)$th row of the matrix $M(z)$ starts with the $(i+1)$th character of the sequence z. The $(j+1)$th row is produced from the jth row in the matrix $M(\cdot)$ by cyclically shifting the sequence one character to the left, so

$$\forall_{1\le j\le n} \, M(y)^j = M(z)^{((i+j-1) \bmod n)+1}, \tag{4.3}$$

which means that the matrices $M(y)$ and $M(z)$ contain the same sets of rows. In the second step of the BWT, the rows from the matrix $M(\cdot)$ are sorted. From the above we achieve

$$\widetilde{M}(y) = \widetilde{M}(z), \tag{4.4}$$

and

$$y^{\mathrm{bwt}} = z^{\mathrm{bwt}}. \tag{4.5}$$

Let us now recall that if $x = uvw$ for some, possible empty sequences u, v, w, then we call v a component of x. Similarly, we call a sequence v of length m (not greater than n), a *cyclic component of x of length m* if, for some, possible empty, sequences u and w, holds $uvw = x_{1..n}x_{1..(m-1)}$.

Let us now assume that

$$y^{\mathrm{bwt}} = z^{\mathrm{bwt}}. \tag{4.6}$$

It means that the last columns of the matrices $\widetilde{M}(y)$ and $\widetilde{M}(z)$ are equal:

$$\forall_{1\le j\le n}\; \widetilde{M}(y)^j_n = \widetilde{M}(z)^j_n. \tag{4.7}$$

The first column of both matrices $\widetilde{M}(y)$ and $\widetilde{M}(z)$ contains all the cyclic components of length 1 of the sequences y^{bwt} and z^{bwt}, sorted lexicographically. From Equation 4.6 we conclude that their first columns are also identical:

$$\forall_{1\le j\le n}\; \widetilde{M}(y)^j_1 = \widetilde{M}(z)^j_1. \tag{4.8}$$

Let us now assume that the last one and m initial columns of the matrices $\widetilde{M}(y)$ and $\widetilde{M}(z)$ are equal:

$$\forall_{1\le j\le n}\; \left(\widetilde{M}(y)^j_n = \widetilde{M}(z)^j_n \wedge \widetilde{M}(y)^j_{1..m} = \widetilde{M}(z)^j_{1..m}\right), \tag{4.9}$$

for some $m \in [1, n-1]$. For each row number j, a concatenation of a character $\widetilde{M}(x)^j_n$ and a cyclic component $\widetilde{M}(x)^j_{1..m}$ of length m forms a cyclic component of length $m+1$. Therefore $\widetilde{M}(x)^j_n\widetilde{M}(x)^j_{1..m}$ for all $1 \le j \le n$ is a set of all possible cyclic components of length $m+1$ of the sequence x. As the m initial and one last column of the matrices $\widetilde{M}(y)$ and $\widetilde{M}(z)$ are equal, then the sets of cyclic components of length $m+1$ for the sequences y and z must also be equal. All these components appear in consecutive rows of the matrices $\widetilde{M}(y)$ and $\widetilde{M}(z)$ in the initial $m+1$ columns. As the matrices are sorted lexicographically, they must be identical on the first $m+1$ columns. Therefore we can write:

$$\begin{aligned}&\forall_{1\le j\le n}\; \left(\widetilde{M}(y)^j_n = \widetilde{M}(z)^j_n \wedge \widetilde{M}(y)^j_{1..m} = \widetilde{M}(z)^j_{1..m}\right) \Longrightarrow \\ &\forall_{1\le j\le n}\; \left(\widetilde{M}(y)^j_{1..(m+1)} = \widetilde{M}(z)^j_{1..(m+1)}\right),\end{aligned} \tag{4.10}$$

for $m \in [1, n-1]$. From this we obtain

$$\widetilde{M}(y) = \widetilde{M}(z). \tag{4.11}$$

As $y = \widetilde{M}(y)^{R(y)}$ and Equation 4.11 holds, then $y = \widetilde{M}(z)^{R(y)}$. Therefore we can conclude that:

$$\exists_i \, y_{1..n} = z_{(1+i)..n} z_{1..i}. \tag{4.12}$$

□

It will be useful for our investigations to define a new term. We call two sequences *BWT-equivalent* if and only if for these sequences we achieve the same output sequence of the BWT. As we have proven, such sequences must be mutual cyclic shifts. Of course, for the BWT-equivalent sequences, the equality $R(y) = R(z)$ holds only when $y = z$. In this section, however, we consider only the output sequences, so the value of $R(.)$ is neglected.

Corollary 4.6. *The number of different sequences after the BWT is the number of non-BWT-equivalent sequences.*

The total number of different sequences of length n over the alphabet of size k is clearly k^n. It is not obvious, however, how many non-BWT-equivalent sequences are there. Let us consider a sequence x such that

$$\forall_{1 \le i < n} \, x \ne x_{(1+i)..n} x_{1..i}. \tag{4.13}$$

For such a sequence there exist n different BWT-equivalent sequences. Each of them is a cyclic shift of the sequence x.

Let us now consider a sequence x, such that

$$\exists_{1 \le i < n} \, x = x_{(1+i)..n} x_{1..i}, \tag{4.14}$$

and i reaches the smallest possible value. Now there are only i different sequences (including the sequence x) that are BWT-equivalent to the sequence x. We also notice that

$$\exists_{1 \le i < n} \, x = x_{(1+i)..n} x_{1..i} \implies \forall_{1 \le j \le n \,\wedge\, ij \le n} \, x = x_{(1+ij)..n} x_{1..(ij)}, \tag{4.15}$$

and

$$\exists_{1 \le i < n} \, x = x_{(1+i)..n} x_{1..i} \implies i | n. \tag{4.16}$$

From the above, we can conclude that if the sequence x is identical to some its cyclic shift, then it is composed of identical components:

$$\exists_{1 \le i < n} \, x = x_{(1+i)..n} x_{1..i} \implies x = \underbrace{y_{1..i} y_{1..i} \cdots y_{1..i}}_{n/i}. \tag{4.17}$$

Now we are ready to find the total number of different non-BWT-equivalent sequences. Let us denote by $U(n)$ the total number of sequences of length n that are different from cyclic shifts of themselves, and by $G(n)$ the number of non-BWT-equivalent sequences.

We also notice that the results to be proven depend on the version of the BWT considered. First we consider the classical BWT (without the sentinel).

Theorem 4.7. *The number of non-BWT-equivalent sequences in the non-sentinel version of the BWT is*

$$G(n) = \frac{k^n}{n} + O\left(\sqrt{n}\, k^{n/2}\right). \tag{4.18}$$

Proof. The total number of the sequences of length n that are not identical to cyclic shifts of themselves is

$$U(n) = k^n - \sum_{\substack{1 \le i < n \\ i | n}} U(i). \tag{4.19}$$

The total number of different non-BWT-equivalent sequences is

$$\begin{aligned} G(n) &= \frac{U(n)}{n} + \sum_{\substack{1 \le i < n \\ i | n}} \frac{U(i)}{i} = \\ &= \frac{k^n}{n} - \sum_{\substack{1 \le i < n \\ i | n}} \frac{U(i)}{n} + \sum_{\substack{1 \le i < n \\ i | n}} \frac{U(i)}{i} = \\ &= \frac{k^n}{n} + \sum_{\substack{1 \le i < n \\ i | n}} U(i) \left(\frac{1}{i} - \frac{1}{n} \right). \end{aligned} \tag{4.20}$$

The recursive Equations 4.19 and 4.20 contain a sum over the divisors of n. To the best of our knowledge, the sum $G(n)$ defies reduction and we do not present its compact form also.

The asymptotical behaviour of $G(n)$ can, however, be analysed. For this, let us first bound the second term of the sum of Equation 4.20. It is obvious that $1 \le U(n) \le k^n$, so we can introduce the following bound:

$$0 < \sum_{\substack{1 \le i < n \\ i | n}} \underbrace{U(i)}_{\le k^i} \underbrace{\left(\frac{1}{i} - \frac{1}{n} \right)}_{<1} < 2 \left\lceil \sqrt{n} \right\rceil k^{n/2}. \tag{4.21}$$

Now the function $G(n)$ can be bound:

$$\frac{k^n}{n} \le G(n) \le \frac{k^n}{n} + 2 \left\lceil \sqrt{n} \right\rceil k^{n/2}. \tag{4.22}$$

Thus

$$G(n) = \frac{k^n}{n} + O\left(\sqrt{n}\, k^{n/2}\right). \tag{4.23}$$

□

We also can find some of the very first values of $G(n)$, which are presented in Figure 4.2.

n	$U(n)$	$G(n)$
1	k	k
2	$k^2 - k$	$\frac{k^2+k}{2}$
3	$k^3 - k$	$\frac{k^3+2k}{3}$
4	$k^4 - k^2$	$\frac{k^4+k^2+2k}{4}$
5	$k^5 - k$	$\frac{k^5+4k}{5}$
6	$k^6 - k^3 - k^2 + k$	$\frac{k^6+k^3+2k^2+2k}{6}$
7	$k^7 - k$	$\frac{k^7+6k}{7}$
8	$k^8 - k^4$	$\frac{k^8+k^4+2k^2+4k}{8}$
9	$k^9 - k^3$	$\frac{k^9+2k^3+6k}{9}$
10	$k^{10} - k^5 - k^2 + k$	$\frac{k^{10}+k^5+4k^2+4k}{10}$
...	...	...

Figure 4.2: Number of different non-BWT-equivalent sequences for small values of n

The number of different sequences that we obtain after the BWT is asymptotically k^n/n. In Section 3.1.1, we showed that after the BWT we have a little expansion of the data because besides the sequence x^{bwt} of length n we achieve an integer $R(x) \in [1, n]$. Here we see that the entropy of the additional number $R(x)$ is compensated by the smaller number of possible sequences x^{bwt}.

The number of different sequences after the classical BWT depends on the sequence length in such a complicated way, because there may exist sequences composed of identical components. When we extend the sequence x with the additional character, the sentinel, we achieve the sequence $x\$$.

Theorem 4.8. *The number of non-BWT-equivalent sequences in the sentinel version of the BWT is*

$$G(n) = k^n. \tag{4.24}$$

Proof. The sentinel character does not occur in the sequence x. The number of different sequences $y = x\$$ is k^n.

We should notice here that now

$$\nexists_{1\le i\le n}\, y_{1..(n+1)} = y_{(1+i)..(n+1)} y_{1..i}, \tag{4.25}$$

because otherwise it would imply that the symbol y_{n+1}, i.e., the sentinel should appear also in some other position, y_i, what contradicts our assumption that the sentinel does not appear in the sequence x. For this reason, no sequences are BWT-equivalent to the sequence $x\$$. From Corollary 4.6 we conclude that the number of different sequences after the BWT is in this case

$$G(n) = k^n. \tag{4.26}$$

□

We obtained also a justification for the statement that the integer $R(x)$ can be neglected in the sentinel version of the BWT. For every sequence x we achieve a different x^{bwt} sequence.

Let us now again assume that the sequence x is produced by the context tree source. We know that the sequence x^{bwt} is a concatenation of the CT-components. If we assume that the CT-components are produced by a memoryless source of parameters from the context tree, we then treat the sequence x^{bwt} as the output of the piecewise stationary memoryless source. This assumption is superfluous, because at least some CT-components cannot be strictly treated as the output of the context tree source (see Corollary 4.4). A similar corollary can be also concluded from the observation that the number of different x^{bwt} sequences is smaller than the total number of possible sequences of length equal to the length of the sequence x^{bwt} (i.e., n in the non-sentinel version and $n+1$ in the sentinel version). These difference is, however, so small that can be neglected, and in practice we can assume that the sequence x^{bwt} is the output of the piecewise stationary memoryless source.

The conclusion similar to the one that the sequence x^{bwt} can be asymptotically treated as an output of the piecewise stationary memoryless source was derived independently in a different way by other authors, as reported by Visweswariah *et al.* [175].

4.1.4 Probability estimation for the piecewise stationary memoryless source

Relation of the source to the Burrows–Wheeler transform

The sequence x^{bwt} is a concatenation of the CT-components. After the BWT, the boundaries between successive CT-components in the sequence x^{bwt} are, however, unknown. In this section, we examine various ways of probability estimation in such a case, what will be useful in our further investigations. To make the considerations possible, we have to make some simplifications. The investigations for large alphabets are possible, but using the binary alphabet makes calculations much easier. Therefore we assume that the elements of the sequence come from a binary alphabet.

Simple estimation

Let us focus on some position in the sequence x^{bwt}. We assume that the length of the current CT-component is m, and the parameter of it is θ_j, however we do not know the position in this component. The parameter of the previous CT-component is θ_{j-1}.

During the compression, we have to estimate the probability of occurrence of all possible characters as the next character in the sequence. To this end, we introduce an estimator $P_e(a,b)$. The inputs of the estimator denote: a—the number of 0s, b—the number of 1s in the last $a+b$ characters, while the its output is an estimation of the probability that the next character be 1.

The probability that memoryless source with parameter θ_j produces one specific sequence of a zeros and b ones (of fixed order) is

$$P_0(j,a,b) = (1-\theta_j)^a \theta_j^b. \tag{4.27}$$

When we consider all the possible sequences of a 0s and b 1s, then the probability is

$$P(j,a,b) = P_0(j,a,b)\binom{a+b}{b} = (1-\theta_j)^a \theta_j^b \binom{a+b}{b}. \tag{4.28}$$

The entropy of a memoryless source of known parameter is

$$H_j = -\theta_j \log \theta_j - (1-\theta_j)\log(1-\theta_j). \tag{4.29}$$

The entropy, H_j, is the lower bound of the expected code length that can be assigned to the symbol. For a given estimator $P_e(a,b)$, the expected code length is

$$L_e(a,b) = -\theta_j \log P_e(a,b) - (1-\theta_j)\log(1-P_e(a,b)). \tag{4.30}$$

The estimators usually predict the future character better when they take more previous characters into account. Here we assume that $a+b=d$ where d is constant. Now we can find the expected redundancy of the estimation, i.e., the difference between the actual code length and the entropy of the source. It is

$$\begin{aligned} R_1(j,d) &= \sum_{a=0}^{d} \big(L_e(a,d-a) - H_j\big) \cdot P(j,a,b)\big) = \\ &= \sum_{a=0}^{d} L_e(a,d-a)P(j,a,d-a) - H_j. \end{aligned} \tag{4.31}$$

So far we have assumed that the estimator bases on the symbols from one CT-component. Let us now investigate what happens, when we assume that only d_j symbols come from the current CT-component, while the remaining $d_{j-1} = d - d_j$ symbols come from the previous CT-component. The expected redundancy equals in this case

$$\begin{aligned} R_2(j, d_{j-1}, d_j) &= \sum_{a_{j-1}=0}^{d_{j-1}} \sum_{a_j=0}^{d_j} ((L_e(a_{j-1}+a_j, d_{j-1}-a_{j-1}+d_j-a_j) - H_j) \times \\ &\times P(j-1, a_{j-1}, d_{j-1}-a_{j-1}) P(j, a_j, d_j - a_j)) = \\ &= \sum_{a_{j-1}=0}^{d_{j-1}} \sum_{a_j=0}^{d_j} L_e(a_{j-1}+a_j, d_{j-1}-a_{j-1}+d_j-a_j) \times \\ &\times P(j-1, a_{j-1}, d_{j-1}-a_{j-1}) P(j, a_j, d_j - a_j) - H_j. \end{aligned} \tag{4.32}$$

We assumed nothing about the position of the current symbol within the current CT-component, so to find the expected redundancy we need to weight it over the all m possible positions. Considering this we arrive at

$$R(j, d) = \frac{1}{m} \left(\sum_{d_j=0}^{d-1} R_2(j, d-d_j, d_j) + (m-d) R_1(j, d) \right). \tag{4.33}$$

Equation 4.33 holds only when $d \leq m$, but it is obvious that when $d > m$, then the estimator uses superfluous symbols that come from another CT-component, and the prediction will not be better than in the case when $d = m$.

Better estimation

When we assume that the sequence is composed of components produced by a memoryless sources, and we try to estimate the probability of the current symbol, then we base on the previous symbols. So far we have neglected the intuitive observation that the longer the distance to the previous symbol is, the more likely it comes from the component produced by a source with different parameter. If a symbol is from a different CT-component, then it introduces an additional redundancy to the estimation. We claim that the importance of the symbol in the probability estimation should depend on the distance between them and the estimated one.

Now we are going to rewrite Equations 4.31, 4.32, and 4.33 introducing a *weight function* $w(\cdot)$, which specifies the "importance" of the symbol depending on the distance. The redundancy in the case when all the symbols come from the same CT-component is

$$\begin{aligned} R_1^*(j,d) &= \sum_{0\le i_1,\ldots,i_d\le 1} \left(L_e\left(\sum_{1\le j\le d} w(j)(1-i_j), \sum_{1\le j\le d} w(j)i_j\right) - H_j\right) \times \\ &\times \ P_0\left(j, \sum_{1\le j\le d}(1-i_j), \sum_{1\le j\le d} i_j\right) = \\ &= \sum_{0\le i_1,\ldots,i_d\le 1} L_e\left(\sum_{1\le j\le d} w(j)(1-i_j), \sum_{1\le j\le d} w(j)i_j\right) \times \\ &\times \ P_0\left(j, \sum_{1\le j\le d}(1-i_j), \sum_{1\le j\le d} i_j\right) - H_j. \end{aligned} \tag{4.34}$$

If symbols from the previous CT-component are also considered, then the redundancy is

$$\begin{aligned} R_2^*(j,d_{j-1},d_j) &= \sum_{0\le i_1,\ldots,i_{d_{j-1}+d_j}\le 1} \left(L_e\left(\sum_{j=1}^{d_{j-1}+d_j} w(j)(1-i_j), \sum_{j=1}^{d_{j-1}+d_j} w(j)i_j\right) - H_j\right) \times \\ &\times \ P_0\left(j-1, \sum_{1\le j\le d_{j-1}}(1-i_j), \sum_{1\le j\le d_{j-1}} i_j\right) \times \\ &\times \ P_0\left(j, \sum_{1\le j\le d_j}(1-i_{d_{j-1}+j}), \sum_{1\le j\le d_j} i_{d_{j-1}+j}\right) = \\ &= \sum_{0\le i_1,\ldots,i_{d_{j-1}+d_j}\le 1} L_e\left(\sum_{j=1}^{d_{j-1}+d_j} w(j)(1-i_j), \sum_{j=1}^{d_{j-1}+d_j} w(j)i_j\right) \times \\ &\times \ P_0\left(j-1, \sum_{1\le j\le d_{j-1}}(1-i_j), \sum_{1\le j\le d_{j-1}} i_j\right) \times \\ &\times \ P_0\left(j, \sum_{1\le j\le d_j}(1-i_{d_{j-1}+j}), \sum_{1\le j\le d_j} i_{d_{j-1}+j}\right) - H_j. \end{aligned} \tag{4.35}$$

Finally, as we do not know the position within the CT-component, we have to weight the redundancy over all possibilities. Fortunately for our calculations every position in the CT-component is equally probable, so the result is

$$R^*(j,d) = \frac{1}{m}\left(\sum_{d_j=0}^{d-1} R_2^*(j,d-d_j,d_j) + (m-d)R_1^*(j,d)\right). \tag{4.36}$$

So far we have assumed nothing about the weight function. If we put

$$w(t) = 1, \qquad \text{for } t > 0, \tag{4.37}$$

then Equation 4.36 reduces to Equation 4.33.

Equation 4.36 specifies the expected redundancy in the case when the sequence is composed of two adjacent components produced by different memoryless sources. The estimation is complex even in the situation when we assume the simplest weight function (Equation 4.37). An exact analysis of the redundancy and finding the value d for which $R^*(j,d)$ is minimal is a hard task. We should also remember that we have made no assumption of parameters of the source, so the values θ_j and θ_{j-1} are unknown. In general, we can weight the expected redundancy, $R^*(\cdot)$, over these parameters, obtaining

$$R_w^*(j,d) = \int_0^1 \int_0^1 \frac{1}{m} \left(\sum_{d_j=0}^{d-1} R_2^*(j,d-d_j,d_j) + (m-d)R_1^*(j,d) \right) d\theta_{j-1}\, d\theta_j. \quad (4.38)$$

The expected redundancy is weighted, assuming a uniform distribution of the parameters θ_{j-1} and θ_j. There is no justification for such an assumption, but there are also no justifications for different distributions so we have chosen the simplest one. Since solving Equation 4.38 is very hard, we analyse it numerically, comparing different weight functions.

Numerical analysis of the weight functions

Before we proceed with the numerical analysis, we have to specify the way of probability estimation, i.e., the function P_e. Simple estimators are usually defined as

$$P_e(1|\ a \text{ zeros and } b \text{ ones}) = \frac{b+\alpha}{a+b+2\alpha}. \quad (4.39)$$

The choice of α determines the estimator, e.g., the Laplace estimator [174] is obtained for $\alpha = 1$ and the Krichevsky–Trofimov estimator (KT-estimator) [95] for $\alpha = 0.5$. The KT-estimator is often used because the upper bound of its redundancy is known. Using it for the weight function specified by Equation 4.37 is well justified, but the grounds are not so strong for other weight functions. In the following analysis, we decided to use the KT-estimator for all the examined weight functions to make the comparison independent on the estimator. (In fact, choosing the value of α can be usually compensated by rescaling the $w(\cdot)$ values.) The examined weight functions are presented in Figure 4.3.

First we have to assume the length of the CT-component. For now, we assume $m = 20$. Such a length is typical in real sequences. In the first experiment, we investigate the redundancy $R(j,d)$ as expressed by Equation 4.36. The values θ_{j-1} and θ_j lie anywhere in the range $[0,1]$, but we assume here four typical pairs: $(\theta_{j-1} = 0.15, \theta_j = 0.65)$, $(\theta_{j-1} = 0.25, \theta_j = 0.85)$, $(\theta_{j-1} = 0.55, \theta_j = 0.30)$, and $(\theta_{j-1} = 0.90, \theta_j = 0.60)$. The examined weight functions have one or two parameters that determine the expected redundancy. At the beginning we examine

$$w_1(t) = 1, \quad \text{for } t \geq 1$$

$$w_2(t) = \begin{cases} q^t, & \text{for } 1 \leq t \leq d \\ 0, & \text{for } t > d \end{cases}$$

$$w_3(t) = \begin{cases} \frac{1}{pt}, & \text{for } 1 \leq t \leq d \\ 0, & \text{for } t > d \end{cases}$$

$$w_4(t) = \begin{cases} 1, & \text{for } t = 1 \\ \frac{1}{pt}, & \text{for } 1 < t \leq d \\ 0, & \text{for } t > d \end{cases}$$

$$w_5(t) = \begin{cases} 1, & \text{for } t = 1 \\ pt^q, & \text{for } 1 < t \leq d \\ 0, & \text{for } t > d \end{cases}$$

$$w_6(t) = \begin{cases} \frac{1}{pt}q^t, & \text{for } 1 \leq t \leq d \\ 0, & \text{for } t > d \end{cases}$$

$$w_7(t) = \begin{cases} t^p q^t, & \text{for } 1 \leq t \leq d \\ 0, & \text{for } t > d \end{cases}$$

Figure 4.3: Examined weight functions

each of the weight functions separately by finding the best values of the parameters. Figure 4.4 illustrates the obtained results. For each pair of probabilities θ_{j-1} and θ_j we have found the best parameters. The parameters are considered to be the best if they lead to the minimal redundancy for any value of d. We notice that different weight functions achieve the best result for different pairs of probabilities. Also, the parameters for which the weight functions yield the best result strongly depend on the values of θ_{j-1} and θ_j. Detailed results of examinations of the weight function are presented graphically in Appendix D.

From the experiment described above we conclude that there is no single best weight function. The next step in the numerical analysis is to investigate the expected redundancy integrated over all the possible values θ_{j-1} and θ_j (Equation 4.38).

Figures 4.5 and 4.6 present the results for different weight functions. The best parameters for the weight functions were found and the comparison of the results is shown in Figure 4.7. The results show that the weight functions $w_2, \ldots, w_7$ outperform the w_1 function significantly. This result confirms our in-

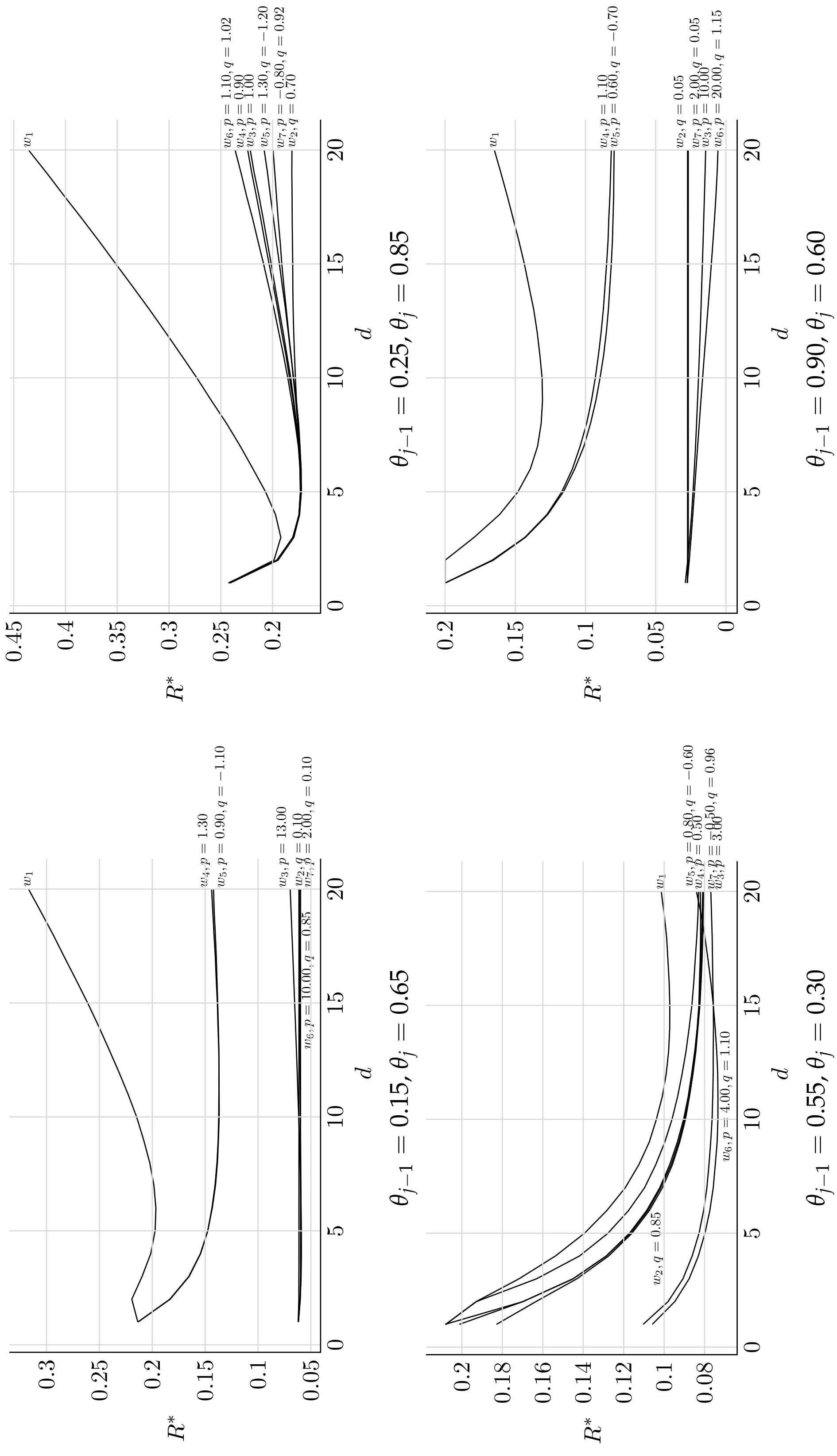

Figure 4.4: Distance from the entropy rate for some weight functions (best results for $m = 20$)

tuitive expectation that older symbols should come with lower importance to the estimator. The other conclusion is that it is hard to choose the best weight function. We should also remember that the investigations were carried out in a simplified case, where the alphabet size was 2, and the length of the CT-component was known.

Non-binary alphabets

We can go one step further and rewrite Equation 4.38 to refrain from the assumption of the binary alphabet. First we need to assume a memoryless source defined by a k-tuple of parameters $\Theta_j = \langle \theta_{j,0}, \theta_{j,1}, \ldots, \theta_{j,k-1} \rangle$, specifying the probabilities of producing all characters from the alphabet. (The elements of the $k-$tuple Θ_j sum to 1.) Now we can rewrite the entropy rate as:

$$H_j^k = -\sum_{i=0}^{k-1} \theta_{j,i} \log \theta_{j,i}. \tag{4.40}$$

The probability that a memoryless source produces a sequence of length d containing character i exactly c_i times for $0 \le i < k$ is (for the clear presentation we define $C = \langle c_0, c_1, \ldots, c_{k-1} \rangle$):

$$P_0^k(j, C) = \prod_{i=0}^{k-1} \theta_{j,i}^{c_i}. \tag{4.41}$$

When we consider all possible sequences of length d containing c_i characters i (in any ordering) for all possible characters, the probability is:

$$P^k(j, C) = P_0^k(j, C) \prod_{i=0}^{k-1} \binom{\sum_{l=i}^{k-1} c_l}{c_i}. \tag{4.42}$$

We also need to reformulate the expression for the expected code length:

$$L_e^k(C) = -\sum_{i=0}^{k-1} \theta_{j,i} \log P_e^k(c_i, C). \tag{4.43}$$

The estimator is defined analogically to the binary estimator:

$$P_e^k(a, C) = \frac{a + \alpha}{c_0 + c_1 + \cdots + c_{k-1} + k\alpha}. \tag{4.44}$$

Having defined the terms, we can rewrite the proper equations, achieving

$$\begin{aligned} R_1^*(j, d) \quad = \quad & \sum_{0 \le i_1, \ldots, i_d < k} L_e \left(\left\langle \sum_{1 \le j \le d} w(j) [i_j = l] \right\rangle \right) \times \\ \times \quad & P_0^k \left(j, \left\langle \sum_{1 \le j \le d} [i_j = l] \right\rangle \right) - H_j^k. \end{aligned} \tag{4.45}$$

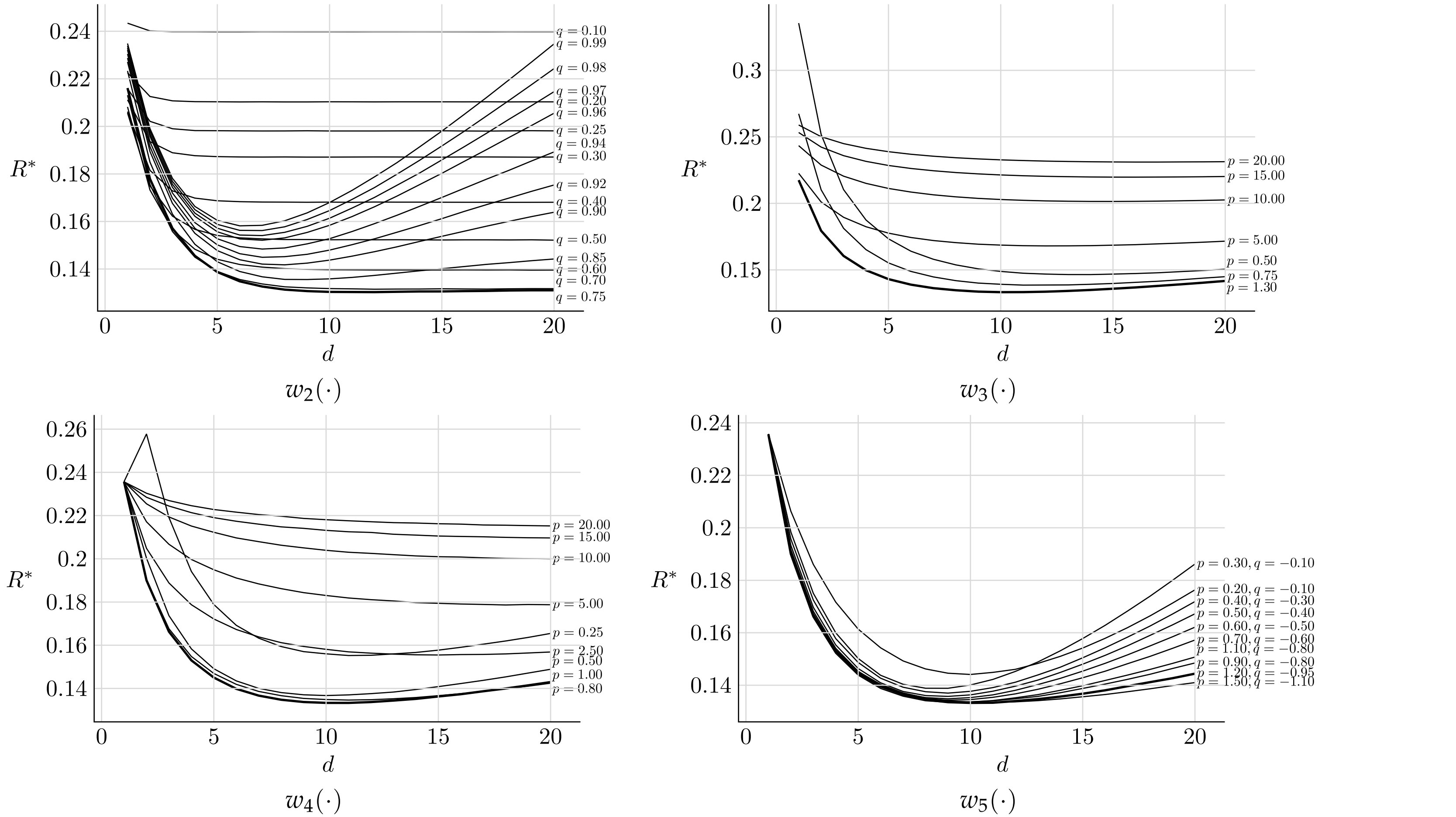

Figure 4.5: Distance from the entropy for weight functions w_2, w_3, w_4, and w_5 (weighted over all possible values of θ_{j-1}, θ_j for $m = 20$)

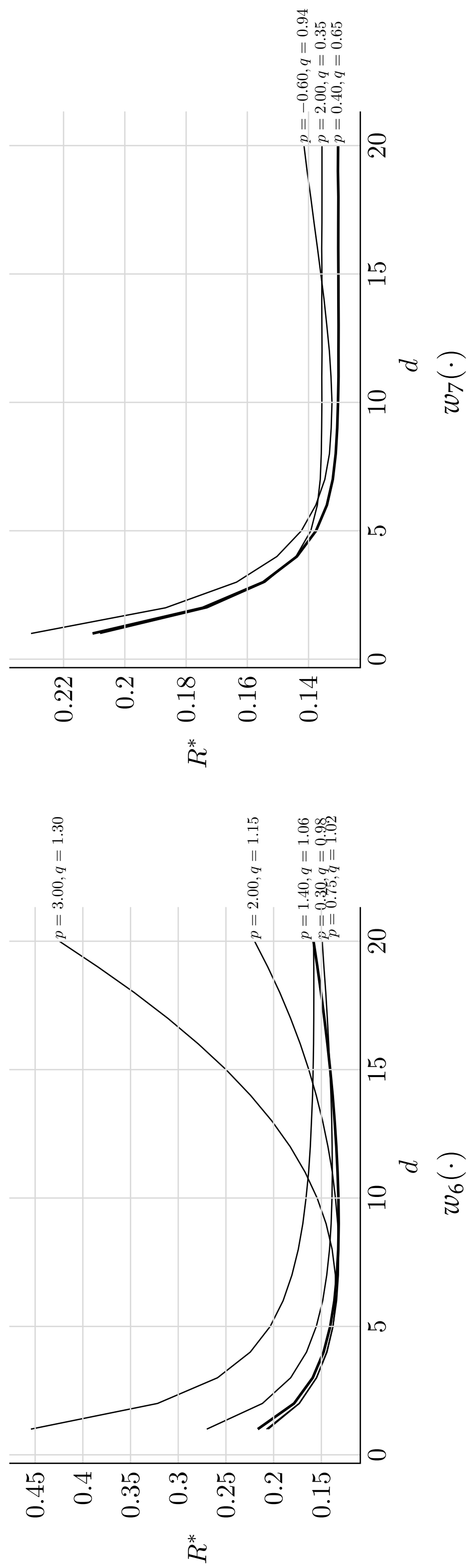

Figure 4.6: Distance from the entropy for weight functions w_6 and w_7 (weighted over all possible values of θ_{j-1}, θ_j for $m = 20$)

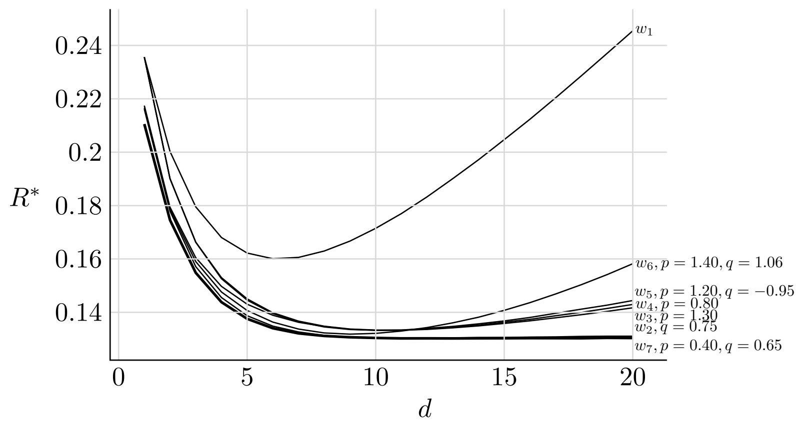

Figure 4.7: Comparison of the best results for different weight functions

$$
\begin{aligned}
R_2^{*k}(j, d_{j-1}, d_j) \quad &= \sum_{0 \le i_1, \ldots, i_{d_{j-1}+d_j} < k} L_e\left(\left\langle \sum_{j=1}^{d_{j-1}+d_j} w(j)[i_j = l] \right\rangle\right) \times \\
&\times \quad P_0^k\left(j-1, \left\langle \sum_{1 \le j \le d_{j-1}} [i_j = l] \right\rangle\right) \times \\
&\times \quad P_0^k\left(j, \left\langle \sum_{1 \le j \le d_j} [i_{j+d_{j-1}} = l] \right\rangle\right) - H_j^k. \qquad (4.46)
\end{aligned}
$$

$$
R^{*k}(j, d) = \frac{1}{m}\left(\sum_{d_j=0}^{d-1} R_2^{*k}(j, d-d_j, d_j) + (m-d)R_1^{*k}(j, d)\right). \qquad (4.47)
$$

$$
\begin{aligned}
R_w^{*k} \quad &= \underbrace{\int \cdots \int}_{\Theta_{j-1}} \underbrace{\int \cdots \int}_{\Theta_j} \frac{1}{m} \times \\
&\times \left(\sum_{d_j=0}^{d-1} R_2^{*k}(j, d-d_j, d_j) + (m-d)R_1^{*k}(j, d)\right) \times \\
&\times \quad d\theta_{j-1,0} \ldots d\theta_{j-1,k-1}\, d\theta_{j,0} \ldots d\theta_{j,k-1}. \qquad (4.48)
\end{aligned}
$$

Equation 4.48 is quite complicated and very hard to solve numerically. Even if we had solved it, the solutions would have not been valuable because in practice we do not know the size of the alphabet—it is usually less than the typically assumed 2^8. The probability distribution of symbols is also usually distant from the assumed one. Because of the above, we will not try to solve this equation and we will base on the results obtained for the binary alphabet.

4.1.5 Weighted frequency count as the algorithm's second stage

Weighted frequency count

The sequence x^{bwt} is a concatenation of the CT-components, and can be treated as the output of a piecewise stationary memoryless source. In Section 4.1.4, we discussed how to estimate the probability in such a case.

In typical implementations of the BWCA, the second stage is the MTF transform. This transform keeps the symbols that appeared recently in the front of the list L. This rule is strong—for every two symbols, the one that has appeared more recently is at a lower position. The numbers of occurrences of characters are not used in the MTF. Now we introduce a solution, a *weighted frequency count* (WFC) transform, which can be considered as a generalisation of the well known *frequency count* transform (FC). (It was introduced first by the author in Reference [55].) The WFC transform makes use of more information about the previous symbols than the MTF.

We first formulate the FC transform in an alternative way. To each character a_j appearing prior to the ith position in the sequence x^{bwt} we assign a sum

$$W_i(a_j) = \sum_{\substack{1 \le p < i \\ a_j = x_p}} 1, \tag{4.49}$$

and sort the list L according to the decreasing values of counters $W_i(a_j)$.

Next, we note that instead of summing 1s for all the characters, we can sum the numbers depending on their relative position in the sequence x^{bwt}. To this end, we introduce a *weight function* $w(\cdot)$ and reformulate the sum as

$$W_i(a_j) = \sum_{\substack{1 \le p < i \\ a_j = x_p}} w(i-p). \tag{4.50}$$

If some two characters have the same value of $W_i(\cdot)$, then we find their relative order using the values $W_{i-1}(\cdot)$, $W_{i-2}(\cdot)$, and so on, until the counters are different. For completeness we define $W_0(a_j) = -j$. The procedure outputting the position of processed characters in the list L and maintaining the list in the described way is called the weighted frequency count transform.

The usage of the weight functions is motivated by the results of the investigations in Section 4.1.4. We should notice that we examined the weight functions for the binary alphabet only. The equations for larger alphabets were elaborated, but they were unsolved because of their complexity. We also assumed some arbitrarily chosen length of the CT-components. For real sequences, the length of the successive CT-components can vary strongly. Finally, we examined the expected redundancy estimating the probability of symbol occurrence, while now this estimation is used to give ranks to the symbols. The last difference was motivated by the observation that the CT-components are rather short, so they contain only a fraction of characters from the alphabet. This is a similar problem to the zero frequency problem that exists in the PPM algorithms. As mentioned in Sections 3.2.5 and 3.2.5, some researchers investigated methods for direct entropy coding of the sequence x^{bwt} without transforming them in order to overcome these problems [67, 195]. Other used a semi-direct encoding and transform some part of the sequence, but the other part is encoded directly [18]. The compression results with a direct entropy encoding method are still significantly worse than the results with some second stage transform, so we decided to introduce the WFC transform as the second stage in the improved BWCA.

Because of the described reasons, we treat previous investigations only as suggestions how the symbols in the second stage should be transformed. Unfortunately, providing the more formal justification of the usage of the WFC transform is a complex task.

From the formulation of the WFC, it is clear that if we set

$$w(t) = 1, \qquad \text{for } t > 0, \tag{4.51}$$

then we obtain the FC transform, and if we set

$$w(t) = \begin{cases} 1, & \text{for } t = 1, \\ 0, & \text{for } t > 1, \end{cases} \tag{4.52}$$

then we obtain the MTF transform.

The *sort-by-time* (SBT) method—proposed by Schulz [146]—is also a special case of the WFC. We achieve it by setting

$$w(t) = q^t, \qquad \text{for } t > 0. \tag{4.53}$$

The theoretical properties of this method for different values of q have been examined by Schulz. He has shown that one obtains the MTF transform by establishing $0 < q \leq 0.5$.

Relation to context tree sources

As mentioned before, the sequence x^{bwt} is a concatenation of the CT-components. Therefore, a character that has occurred at the previous position is more likely to be at the same context than the penultimate and so on. In general, the character that has appeared at recent positions is more likely described by the same probability distribution as the current one, than by the distribution of the characters from more distant positions. Unfortunately, we do not know the current position in the context tree, how long the CT-component of the current leaf is, and where in that CT-component we are. We examined all these problems in Section 4.1.4.

There is also one more property to consider: similar context typically have only a slightly different probability distribution. (In fact, the similarity of probability distributions of similar contexts is one of the bases of the PPM algorithms.) It may be useful to explore some of the information regarding the probability distribution of the previous contexts. Because the formulation of the context tree source does not give a way to represent this fact, it is impossible to incorporate it in the theoretical analysis, when the context tree source is assumed. All in all, the values $w(i)$ of the weight function should decrease with increasing i. It is not clear, however, how fast it should decrease.

We theoretically examined some weight functions for the binary alphabet. Now, concerning the results of the theoretical analysis, we will examine how the weight functions work for real-world sequences. The experimental results show that for different sequences from the Calgary corpus different functions w give the best results. This is what one would expect. (We made the same observation in the theoretical analysis.) The CT-components of short sequences are typically shorter than those of longer sequences. Also, the sequences x are generated by different sources with a different number of contexts.

Efficient implementation

Prior to the empirical comparison of the weight functions, let us discuss its implementation. The formulation of the WFC transform does not give us a way for computing the values of $W_i(\cdot)$ quickly. When we move to the next character, we have to recalculate the values of all counters $W_i(\cdot)$. To this end, we have to rescan the encoded part of the sequence x^{bwt}. This rescanning makes the time complexity of the transform $O(n(n + k \log k))$.

We can improve the time complexity of the WFC transform by sacrificing its precision. One possibility is to quantise the values of the weight function to integer powers of 2. This quantisation decreases the number of different values of w to at most $l = \log w(1)/w(t_{max} + 1)$ (we assume that the weight function is non-increasing), which is typically small. For such values of the weight function, we can obtain the values of $W_i(\cdot)$ from $W_{i-1}(\cdot)$ by updating only the counters for the characters where the values $w(\cdot)$ are changing (for all such t that $w(t) \neq w(t-1)$). Using this approach, we obtain a method of the worst-case time complexity $O(nlk)$, which is not much greater than $O(nk)$ for transforms like the MTF. In practice, the characters on the list L move only by a few positions at a time. With a version of the insertion sort procedure the cost of maintaining the list is small (e.g., for the function w_{8q} the average number of swaps of characters in the list L per input character does not exceed 6 for almost all files from the Calgary corpus, except for binary files such us `geo`, where it is larger).

The disadvantage, in the compression ratio, of using the quantisation depends on the weight function w and properties of the sequence. One can double the number of different values of w if necessary by using also the powers of 2 for half exponents. It is also possible to introduce even faster methods for some weight functions, e.g., for the function w_2. We will not discuss, however, here such improvements, which are specific to weight functions.

We examined a number of weight functions. Here we present (Figure 4.8) only some of them (the best ones and those that we find interesting). Figure 4.9 shows an example of working of the WFC transform and Section 4.3.2 contains the experimental results we obtained for these weight functions on standard data compression sets.

4.1.6 Efficient probability estimation in the last stage

Binary arithmetic coder

The last stage of the BWCA is the entropy coding of the sequence $x^{\text{rle-0}}$. Different solutions to this task were discussed in Section 3.2.5. In this section, we introduce yet another method. The way of probability estimation described in this section was introduced by the author of this dissertation in Reference [54] and partially (for the IF transform) in Reference [55].

As the motivation of this work is to improve the compression ratio, the arithmetic coder is chosen as the entropy coder. In contrast to other researchers, we choose, however, a binary arithmetic coder. This coder is highly efficient, and no complicated model is needed to store the statistics of symbol occurrences. Many experiments with different methods for probability estimation of symbols from the sequence $x^{\text{rle-0}}$ were carried out and the simplicity of the binary arithmetic coder is its important asset.

Decomposing a sequence into binary codes

Since a binary arithmetic coder is chosen, the symbols from the sequence $x^{\text{rle-0}}$ have to be decomposed to binary codes before the coding. The decomposition of the symbols from the alphabet $\mathcal{A}^{\text{rle-0}}$ proceeds as follows. In the first step, the symbols are grouped into nine subsets: $\{0_a\}$, $\{0_b\}$, $\{1\}$, $\{2,\ldots,7\}$, $\{8,\ldots,15\}$, $\ldots$, $\{128,\ldots,255\}$. Then all the symbols are encoded using the binary prefix code presented in Figure 4.10. The code for each symbol consists of two parts: a unique prefix that distinguishes the subsets from each other, and a suffix, which is a binary representation of the symbol (b_i denotes the ith bit of the binary representation of the symbol). The second part of the code (not always present) contains these bits, which are indispensable to distinguish all the characters in a given subset. For example, for the subsets $\{8,\ldots,15\}$, $\ldots$, $\{128,\ldots,255\}$ it is unnecessary to encode the most significant bit of the symbol.

$$w_1(t) = \begin{cases} 1, & \text{for } t = 1 \\ 0, & \text{for } t > 1 \end{cases}$$

$$w_2(t) = \begin{cases} q^t, & \text{for } 1 \le t \le t_{max} \\ 0, & \text{for } t > t_{max} \end{cases}$$

$$w_3(t) = \begin{cases} \frac{1}{pt}, & \text{for } 1 \le t \le t_{max} \\ 0, & \text{for } t > t_{max} \end{cases}$$

$$w_4(t) = \begin{cases} 1, & \text{for } t = 1 \\ \frac{1}{pt}, & \text{for } 1 < t \le t_{max} \\ 0, & \text{for } t > t_{max} \end{cases}$$

$$w_5(t) = \begin{cases} 1, & \text{for } t = 1 \\ pt^q, & \text{for } 1 < t \le t_{max} \\ 0, & \text{for } t > t_{max} \end{cases}$$

$$w_6(t) = \begin{cases} \frac{1}{pt}q^t, & \text{for } 1 \le t \le t_{max} \\ 0, & \text{for } t > t_{max} \end{cases}$$

$$w_7(t) = \begin{cases} t^p q^t, & \text{for } 1 \le t \le t_{max} \\ 0, & \text{for } t > t_{max} \end{cases}$$

$$w_8(t) = \begin{cases} 1, & \text{for } t = 1 \\ \frac{1}{pt}, & \text{for } 1 < t \le 64 \\ \frac{1}{2pt}, & \text{for } 1 < t \le 256 \\ \frac{1}{4pt}, & \text{for } 1 < t \le 1024 \\ \frac{1}{8pt}, & \text{for } 1 < t \le t_{max} \\ 0, & \text{for } t > t_{max} \end{cases}$$

$$w_9(t) = \begin{cases} 1, & \text{for } t = 1 \\ \frac{1}{pt}q^t, & \text{for } 1 < t \le t_{max} \\ 0, & \text{for } t > t_{max} \end{cases}$$

Figure 4.8: Weight functions examined in the WFC transform

L	a (0)	d (0.700)	r (0.700)	c (0.700)	r (1.043)	r (0.730)	a (1.190)	a (1.533)	a (1.773)	a (1.241)	b (1.190)
	b (−1)	a (0.000)	d (0.490)	r (0.490)	c (0.490)	a (0.700)	r (0.511)	r (0.358)	r (0.250)	b (0.700)	a (0.869)
	c (−2)	b (0.000)	a (0.000)	d (0.343)	d (0.240)	c (0.343)	c (0.240)	c (0.168)	c (0.118)	r (0.175)	r (0.123)
	d (−3)	c (0.000)	b (0.000)	a (0.000)	a (0.000)	d (0.168)	d (0.118)	d (0.082)	d (0.058)	c (0.082)	c (0.058)
	r (−4)	r (0.000)	c (0.000)	b (0.000)	b (0.000)	b (0.000)	b (0.000)	b (0.000)	b (0.000)	d (0.040)	d (0.028)
x^{bwt}	d	r	c	r	a	a	a	a	b	b	a
x^{wfc}	**3**	**4**	**4**	**1**	**3**	**1**	**0**	**0**	**4**	**1**	**1**

Figure 4.9: Example of the WFC transform with the weight function w_2 and parameters $q = 0.7$, $t_{max} = 2048$. The values of counters $W(\cdot)$ are given in parenthesis.

Symbol	Code
0_a	0 0
0_b	0 1
1	1 0
$2, \ldots, 7$	1 1 0 b_2 b_1 b_0
$8, \ldots, 15$	1 1 1 0 b_2 b_1 b_0
$16, \ldots, 31$	1 1 1 1 0 b_3 b_2 b_1 b_0
$32, \ldots, 63$	1 1 1 1 1 0 b_4 b_3 b_2 b_1 b_0
$64, \ldots, 127$	1 1 1 1 1 1 0 b_5 b_4 b_3 b_2 b_1 b_0
$128, \ldots, 255$	1 1 1 1 1 1 1 b_6 b_5 b_4 b_3 b_2 b_1 b_0

Figure 4.10: Encoding the alphabet $\mathcal{A}^{\text{rle-0}}$

Another decomposition method is needed if the IF or the DC transform is used as the second stage in the BWCA. The symbols of the sequence x^{if} can be integers from 0 up to n in the original proposition by Arnavut and Magliveras [10, 11], and from 0 up to kn in our implementation. We work with higher symbols, because instead of encoding also the number of occurrences of all characters from the alphabet, which is necessary in Arnavut's approach, we have modified the IF transform as follows. When a character a_j is processed and the end of the sequence x^{bwt} is reached, the current character is incremented to a_{j+1} and further counting from the beginning of the sequence x^{bwt} is performed. Therefore we can sometimes obtain a number greater than n.

The advantage of using the binary arithmetic coder is that each bit of the code is encoded separately and the size of the symbol does not cause any problem. The decomposition of the symbols from the sequence x^{if} is quite simple: for each number x_i^{if} we calculate the Elias γ code [59] of length $1 + \lfloor 2 \log x_i^{\text{if}} \rfloor$.

Probability estimation

Another task in the entropy coding stage is to encode the string of bits obtained after encoding symbols with the described binary code. In order to improve the probability estimation, we calculate separate statistics for different contexts in which consecutive bits occur. The probabilities of bits in different contexts may be completely different, and thus it is essential to choose the context correctly. We should remember that a small redundancy with each context is introduced. If the number of contexts is too large, then the gain of separating contexts can be lost because of the additional redundancy.

Figure 4.1.6 details the calculation of the context c for successive bits of code representing the symbols from the $\mathcal{A}^{\text{rle-0}}$ alphabet. For the symbols from the sequence x^{if} each consecutive bit is encoded in a separate context.

Code bit	No. of contexts	Possible contexts
first bit	6	the last character is 0, and the 0-run is not longer than 2[†], the last character is 0, and the 0-run is longer than 2, the last character is 1, and the last but one character is 0, the last character is 1, and the last but one character is not 0, the last character is greater then 1, and the last but one character is 0, the last character is greater then 1, and the last but one character is not 0
second bit (first bit = 0)	$\log n$	the length of 0-run forms a context
second bit (first bit = 1)	3	the last character is 0 or 1, the last character is in range $[2, 7]$, the last character is not in range $[0, 7]$
successive code bits	254	all previous code bits form a context

Figure 4.11: Calculating the context in the improved BWCA

Weighted probability

It was assumed that the structure of the output sequence of the BWT is simple and there is no reason for using an arithmetic coder with an order higher than 0. Some results that confirm this postulate were obtained by Fenwick [67]. Successive research [18, 195] showed, however, that carefully choosing the order of the arithmetic coder in some contexts can improve the compression ratio.

In the algorithm proposed in this dissertation, we treat the sequence of bits occurring in context c as a Markov chain of order d, and use the Krichevsky–Trofimov estimator [95] to estimate the probability of bit occurrence. This means that we encode the subsequent bits with the conditional probability

$$P_c^d(a|s_c^d) = \frac{t_c^d(a|s_c^d) + \frac{1}{2}}{t_c^d(0|s_c^d) + t_c^d(1|s_c^d) + 1}, \tag{4.54}$$

where s_c^d are the last d bits encoded in the context c, and $t_c^d(a|s_c^d)$ is the number of occurrences of bit a in the context c, provided the last d coded bits were s_c^d. As mentioned before, the probability estimation improves when the order of the Markov chain increases. Unfortunately, as d grows, the number of states s for

[†]The symbol 0 denotes either 0_a or 0_b.

which the probability is estimated, grows exponentially (as 2^d). Thus increasing the order d adds redundancy caused by estimating probability in many states.

Using a higher order of the Markov chain can give some gains. We go even further and postulate to use a weighted probability estimation for different orders, calculated as

$$P_c^{d_1,d_2}(a|s_c^{\max(d_1,d_2)}) = \frac{1}{2}P_c^{d_1}(a|s_c^{d_1}) + \frac{1}{2}P_c^{d_2}(a|s_c^{d_2}) \tag{4.55}$$

for $d_1 \neq d_2$. It follows from Equation 4.55 that we treat the sequence of bits as the Markov chains of order d_1 and of order d_2, and estimate the probabilities in both models. Then we use the average of these two probabilities. Since we do not know which model describes the encoded sequence of bits in a particular context, and a model may depend on context c and a current position of a symbol in the sequence, using such an estimation may give good results. As the experiments described in Section 4.3.4 show, this method of calculating the probability indeed improves the compression ratios. The best results were obtained when setting $d_1 = 0$ and $d_2 = 2$, i.e., when estimating the probability as

$$P_c^{0,2}(a|s_c^2) = \frac{1}{2}\left(\frac{t_c^0(a|s_c^0) + \frac{1}{2}}{t_c^0(0|s_c^0) + t_c^0(1|s_c^0) + 1} + \frac{t_c^2(a|s_c^2) + \frac{1}{2}}{t_c^2(0|s_c^2) + t_c^2(1|s_c^2) + 1}\right). \tag{4.56}$$

We want to point out that Equation 4.56 is based on the practical experiments in which we achieved a significant improvement in the compression ratios. We cannot support this equation by a precise theoretical argument due to still incomplete knowledge of the structure of the sequence of bits occurring in context c, which would prove the concept of using this model, and the choice of $d_1 = 0$ and $d_2 = 2$ as well. We suspect that justifying this model formally will be a hard task because we assume that the input sequence is produced by any context tree source, while the real sequences for which the best values were chosen are rather special cases of the context tree sources.

Other modifications

The analysis of the sequence x^{mtf} indicates (Figure 4.12) that for most data files from the Calgary corpus the average value of symbols in the sequence x^{mtf} does not exceed 15. For some binary files (geo, obj2), however, there are fragments for which this value grows significantly, even exceeding 100. It can be seen in Figure 4.12 that the fragments with a large average value are preceded by the fragments with a very small value. In such a case, the adaptive properties of the arithmetic coder are crucial to achieve a good adjustment to the local properties of a coded sequence. The second observation is that for the fragments with a high symbol average, our model is less effective because each symbol has a

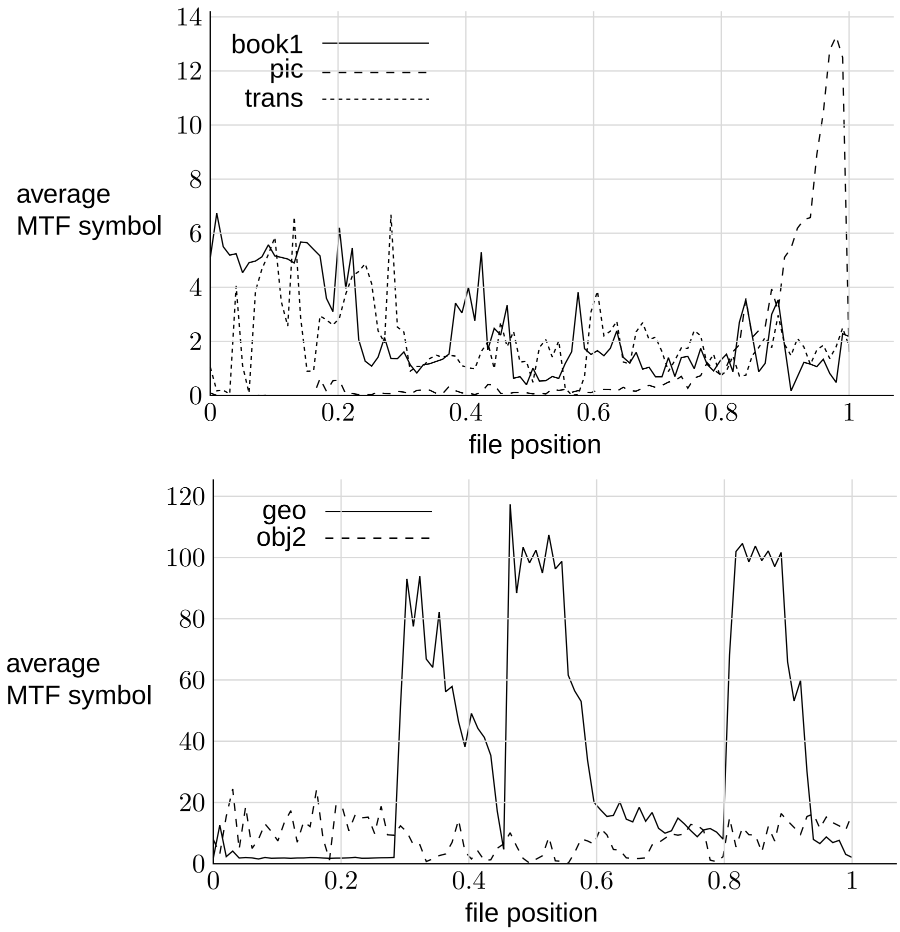

Figure 4.12: Average value of symbols in the consecutive fragments of the sequence x^{mtf} for different data files

relative long code. During the arithmetic coding, a running average is calculated as

$$\mathrm{avg}(t) = \mathrm{avg}(t-1) \cdot (1-\epsilon) + x_t^{\text{rle-0}} \cdot \epsilon. \tag{4.57}$$

If it exceeds a certain threshold, then the symbols from the alphabet $\mathcal{A}^{\text{rle-0}}$ are encoded with the code presented in Figure 4.13. Experimental results show that good results can be obtained if we assume $\epsilon = 0.15$ and the threshold 64.

Updating the statistics

The sequence produced by the Burrows–Wheeler transform is composed of the CT-components only if the input sequence is generated by a context tree source. Unfortunately, the model of the source describing the sequence x is unknown during a compression process. Furthermore, it is unknown where the change of

Symbol	Code
0_a	0 0
0_b	0 1
$1, \ldots, 255$	$1\ b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$

Figure 4.13: Encoding the alphabet $\mathcal{A}^{\text{rle-0}}$ for a high average symbol value

Contexts	t^0_{max}	t^2_{max}
first bit second bit (first bit = 1) remaining bits of code prefix (except b_i)	20	150
first bit b_i, when the average ≤ 64 first four bits b_i, when the average > 64	30	300
second bit (first bit = 0) all except the first bit b_i, when the average ≤ 64 last four bits b_i, when the average > 64	300	700

Figure 4.14: Thresholds for counter halving in different contexts

the context takes place in the sequence x^{bwt}. The probability distribution, as well as the number of symbols, varies significantly from one context to another, and thus the probability distribution in subsequent fragments of the sequence $x^{\text{rle-0}}$ may also vary a lot. Therefore, in order to adjust the coder to the local symbol distribution in the best possible manner, the counters $t^{d_1}_c(a|s^{d_1}_c)$ and $t^{d_2}_c(a|s^{d_2}_c)$ are halved according to the formulas

$$t^{d_1}_c(a|s^{d_1}_c) = \left\lfloor \frac{t^{d_1}_c(a|s^{d_1}_c)}{2} \right\rfloor, \qquad t^{d_2}_c(a|s^{d_2}_c) = \left\lfloor \frac{t^{d_2}_c(a|s^{d_2}_c)}{2} \right\rfloor. \tag{4.58}$$

The halving occurs when the sum of the counters for 0 and 1 exceeds the thresholds in a given context $t^{d_1}_{max}$ and $t^{d_2}_{max}$. Because of different speeds at which the probability distribution change in different contexts, the threshold values $t^{d_1}_{max}$ and $t^{d_2}_{max}$ differ from context to context (see Figure 4.14).

A small improvement in the compression ratio can also be achieved by initialising the counters $t^{d_1}_c$ and $t^{d_2}_c$ with values $t^{d_1}_{max}/32$ and $t^{d_2}_{max}/32$ respectively, and by incrementing these counters by 2 (in this case the values $t^{d_1}_{max}$ and $t^{d_2}_{max}$ are multiplied by 2).

4.2 How to compare data compression algorithms?

4.2.1 Data sets

Choosing the test data

In Section 4.1, we introduced an improved compression algorithm based on the Burrows–Wheeler transform. In the following sections, we examine its practical efficiency.

To compare the compression algorithms we need a set of files. In general, there are two ways of choosing the test files. The first way is to use a well-known data set. The second way is to prepare new set for testing. It is convenient to use a standard corpus, because it should be easy to compare new results to those previously published. Sometimes, however, existing corpora do not give us an answer to the question about the behaviour of the algorithms in all situations we are interested in, because it could miss the files from some categories.

Standard data sets

Three well-known data sets are used by researchers in the universal lossless data compression field. The first one, the *Calgary corpus*, was introduced in 1989 by Bell *et al.* [20, 22]. The files in the corpus were chosen to cover up the typical types of data used in computer processing. A description of the contents of the corpus is given in Table 4.1.[‡] This corpus is rather old, and it contains some types of data which went out of use, but the corpus is still a good benchmark used by many authors.

In 1997, Arnold and Bell proposed [12] a replacement for the Calgary corpus. The authors reviewed the types of data used contemporarily, examined many files and proposed a new corpus, nicknamed the *Canterbury corpus* (Table 4.2). The corpus is newer than the Calgary corpus, but some files were chosen in a rather unfortunate manner. The most troublesome file is kennedy.xls. Its specific structure causes different algorithms to achieve strange results. There are also simple filters which applied to this file before compression can rapidly improve the compression ratio. The difference in compression ratio for this file shows no correlation to the efficiency of algorithms in practical usage. The more so, the differences in the compression ratio on this file are large enough to dominate the overall average corpus ratio. The second disadvantage of this corpus is its usage of very small files. A small difference in the size of the compressed file makes a significant difference in the compression ratio and causes important, and disproportional, participation to the average ratio. The last reservation to

[‡]The corpus presented in Reference [22] contains more papers (paper3 ... paper6) but these files were later removed, and the established contents is as in the table.

File	Size [B]	Description	Type of data
bib	111,261	Bibliographic files (refer format)	text data
book1	768,771	Text of the book *Far from the madding crowd* by Thomas Hardy	English text
book2	610,856	Text of the book *Principles of computer speech* by Ian H. Witten in Unix troff format	English text
geo	102,400	Geophysical (seismic) data	binary data
news	377,109	A Usenet news batch file	English text
obj1	21,504	Compiled code for Vax: compilation of progp	executable
obj2	246,814	Compiled code for Apple Macintosh: Knowledge support system	executable
paper1	53,161	A paper *Arithmetic coding for data compression* by Witten, Neal and Cleary in Unix troff format	English text
paper2	82,199	A paper *Computer (in)security* by Witten in Unix troff format	English text
pic	513,216	Picture number 5 from the CCITT Facsimile test files (text + drawings)	image
progc	39,611	C source code: compress program version 4.0	C source
progl	71,646	Lisp source code: system software	Lisp source
progp	43,379	Pascal source code: prediction by partial matching evaluation program	Pascal source
trans	93,695	Transcript of a session on a terminal using the EMACS editor	English text
Total	3,141,622		

Table 4.1: Description of the Calgary corpus

the corpus is that it does not reflect the tendency of the computer files to grow. Because of the above, we decided not to use it.

The disadvantage of the absence of large files in the Canterbury corpus was lately partially removed by the proposition of the *large Canterbury corpus* [98] (Table 4.3). This corpus contains three files larger than the files in both the Calgary and the Canterbury corpora. Two of these files are English texts and one is the binary data of genome, e.coli, that is very hard to compress. The recent file is composed of symbols from the alphabet of size 4 and no algorithm currently known outperforms the obvious bound of 2.0 bpc by more than 5%.

Silesia corpus

The existing corpora are widely used for testing the compression algorithms. Because of the very fast progress in computer science their contents do not reflect,

File	Size [B]	Description	Type of data
alice29.txt	152,089	A book *Alice's Adventures in Wonderland* by Lewis Carroll	English text
asyoulik.txt	125,179	A play *As you like it* by William Shakespeare	English text
cp.html	24,603	Compression Pointers	HTML
fileds.c	11,150	C source code	C source
grammar.lsp	3,721	LISP source code	Lisp source
kennedy.xls	1,029,774	Excel Spreadsheet	binary data
lcet10.txt	426,754	*Workshop on electronic texts, Proceedings* edited by James Daly	English text
plrabn12.txt	481,861	*Paradise Lost* by John Milton	English text
ptt5	513,216	Picture number 5 from the CCITT Facsimile test files (text + drawings)	image
sum	38,240	SPARC executable	executable
xargs.1	4,227	GNU manual page	text data
Total	2,810,784		

Table 4.2: Description of the Canterbury corpus

File	Size [B]	Description	Type of data
e.coli	4,6380690	Complete genome of the *Esxherichia coli* bacterium	binary data
bible	4,047,392	The King James version of the bible	English text
world192.txt	2,473,400	The CIA world fact book	English text
Total	11,159,482		

Table 4.3: Description of the large Canterbury corpus

however, contemporary files. Nevertheless, many papers on universal lossless data compression algorithms contain comparisons for these data sets, and, in our opinion, these corpora will be still valuable candidates as benchmarks in a few years.

Over the years of using of these corpora some observations have proven their important disadvantages. The most important in our opinion are:

- the lack of large files—at present we work with much larger files;
- an over-representation of English-language texts—there are only English files in the three corpora, while in practice many texts are written in different languages;
- the lack of files being a concatenation of large projects (e.g., programming projects)—the application sizes grow quite fast and compressing each of

File	Size [B]	Description	Type of data
dickens	10,192,446	Collected works of Charles Dickens (from Project Gutenberg)	English text
mozilla	51,220,480	Tarred executables of Mozilla 1.0 (Tru64 Unix edition) (from Mozilla Project)	executable
mr	9,970,564	Medical magnetic resonance image	3D image
nci	33,553,445	Chemical database of structures	database
ooffice	6,152,192	A dynamic linked library from Open Office.org 1.01	executable
osdb	10,085,684	Sample database in MySQL format from Open Source Database Benchmark	database
reymont	6,625,583	Text of the book *Chłopi* by Władysław Reymont	PDF in Polish
samba	21,606,400	Tarred source code of Samba 2-2.3 (from Samba Project)	executable
sao	7,251,944	The SAO star catalogue (from Astronomical Catalogues and Catalogue Formats)	bin database
webster	41,458,703	The 1913 Webster Unabridged Dictionary (from Project Gutenberg)	HTML
xml	5,345,280	Collected XML files	XML
x-ray	8,474,240	X-ray medical picture	image
Total	211,938,580		

Table 4.4: Description of the Silesia corpus elaborated within the dissertation

the source files separately is impractical presently; a more convenient way is to concatenate the whole project and to compress the resulting file;

- absence of medical images—the medical images must not undergo a lossy compression because of law regulations;
- the lack of databases that currently grow considerably fast—databases are perhaps the fastest growing type of data.

In this dissertation, we decided to introduce a corpus which solves the problems observed with the existing corpora. We think that nowadays the most important matter is to create the corpus of large files. Because of the above, we propose to use two corpora together: the Calgary corpus and the *Silesia corpus* introduced in this section. Each of these data sets should be used separately to enable comparing the new compression results to the existing ones.

Our intention is to propose a set of files that are significantly bigger than the ones in the Calgary and the Canterbury corpora. We have chosen the files to be of such sizes that should not prove too small in several years (Table 4.4). The chosen files are of different types and come from several sources. In our opinion, nowadays the two fastest growing types of data are multimedia and databases. The former are typically compressed with lossy methods so we do not include them in the corpus. The database files, osdb, sao, nci, come from three different fields. The first one is a sample database from an open source project that is intended to be used as a standard, free database benchmark. The second one, sao, is one of the astronomical star catalogues. This is a binary database composed of records of complex structure. The last one, nci, is a part of the chemical database of structures.

The sizes of computer programs are also growing rapidly. The standard corpora include only single, small routines, both in source and object code. Today it is almost impractical to compress every single source code file separately. The projects are composed of hundreds or thousands files, so it is a common habit to compress it all together. We often can achieve a better compression ratio if we compress a concatenated file of similar contents than the small separate ones. This trend is reflected in including a samba file. Besides the source codes, there is also a need to store the executables. We decided to include two files: ooffice and mozilla. The first one is a single medium-sized executable for the Windows system. The second is a concatenation of the whole application for Tru64 Unix system composed of executables, archives, texts, HTML files, and other.

We mentioned before that there are types of images that cannot be compressed loosely—the medical images. The sizes of such files are also huge and we include two examples of them in the corpus. The first file, x-ray, is an X-ray picture of a child's hand. The second file, mr, is a magnetic resonance, three dimensional image of a head.

The standard corpora contain text files. Moreover, these files are typically the largest files of them, but in our opinion there is a need to test the compression efficiency also on the larger ones stored in different file types. We propose three such files. The first, dickens, is a collection of some works by Charles Dickens that can be found in the Project Gutenberg. This is a plain text file. The second one, reymont, is a book *Chłopi* [133] by Władysław Reymont stored in a PDF file. The PDF files can be internally-compressed but the quality of this build-in compression is rather poor, and much better results can be obtained when we compress an uncompressed PDF file. Because of this we enclose the uncompressed version. The last text file, webster, is an electronic version of *The 1913 Webster Unabridged Dictionary* [130] taken from the Project Gutenberg [131]. The file is stored in the HTML format. The last file of the new corpus, xml, is a concatenation of 21 XML files. The XML standard is designed to be a universal file format for storing documents, so we decided to enclose it.

A more detailed description of the contents of the corpus can be found in Appendix A. The whole corpus can be downloaded from the URL: http://www-zo.iinf.polsl.gliwice.pl/~sdeor/corpus.htm.

4.2.2 Multi criteria optimisation in compression

Compression is a process of reducing the size of a sequence of characters to save the cost of transmission or storage. One could think that the better the compression ratio, the better the algorithm is. This is true if we neglect time, but time also matters. Let us suppose that we can choose between two compression methods: one of them gives good compression ratio, but is slow; the other is fast but offers a poorer compression ratio. If our goal is to transmit the sequence over a communication channel, then it is not obvious which method should be used. The slower one will produce a shorter compressed sequence, but the compression process is slow, and maybe the compression time surpass the transmission time and we could not use the full communication speed available, because we would have to wait for the data to transmit. The faster method will produce a longer compressed sequence, but the communication could be done with full speed. Without knowledge of the speed of compression methods on particular sequences, the sequence length, and the parameters of the communication channel, we cannot say, which method leads us to faster transmission.

We should not forget also the decompression process. The speed of the decompression in many cases significantly differs from the compression speed. We can imagine a situation typical in the multimedia world. We usually have much time to compress a DVD film, and we often have powerful computers for this task. The compression will be done, however, only once. The decompression is done in every DVD player many times. The computation power of the DVD player is lower. The files distributed via Internet are also compressed only once, but they are downloaded and decompressed many times. Often the decompression speed is more important than the compression speed.

We should not forget, however, the situation in the data transmission, when the data are compressed before the transmission, and decompressed after it. In such a case, the compression and decompression speed is usually of equal importance. An opposite situation takes place in backup utilities. The data are compressed many times, in every backup, but almost never decompressed. In this case, the decompression speed is of low importance.

The situation described before is typical in the real world. We often have many criteria, which can be exclusive, and we cannot optimise all of them. We have to opt for a compromise, and without a detailed knowledge of the case at which the compression will be used, we cannot choose the best compression method. In such situations, we talk about *multi criteria optimisation*. The first research in this field was conducted by Pareto [122, 123] in 1896–1897, who investigated the problem of satisfying many exclusive criteria. In 1906, in his famous book, *Manuale di economia politica, conuna introduzione alla scienca Sociale* [124], Pareto introduced a concept of the *non-dominated solutions*. Pareto formulated such a solution in economical terms as a solution, where no individual could be more satisfied without satisfying others less. Currently, we call such solutions Pareto-optimal. The solutions that are not non-dominated are called *dominated solutions*.

We formulate the problem of multi criteria optimisation in modern terms, following the works by von Neumann and Morgenstern [184]. There are some number of criteria $Q_1, Q_2, \ldots, Q_m$ dependent on some variables:

$$Q_i = Q_i(q_1, q_2, \ldots, q_r), \qquad \text{for } i = 1, 2, \ldots, m. \tag{4.59}$$

A tuple of variables:

$$q = \langle q_1, q_2, \ldots, q_r \rangle \tag{4.60}$$

is called a point in the optimisation. A point q is called *Pareto-optimal* if there is no other point p such that

$$\forall_{1 \le i \le m}\ Q_i(p) \ge Q_i(q). \tag{4.61}$$

The goal of multi criteria optimisation is to find the set of all the Pareto-optimal points. The formulation of the Pareto-optimal set (Equation 4.61) is for the case, in which all the criteria Q_i are maximised. In general case, all or some criteria can be minimised. Therefore, we should reformulate Equation 4.61 to:

$$\forall_{1 \le i \le m}\ Q_i(p) \,\Box_i\, Q_i(q), \tag{4.62}$$

where $\Box_i$ is the $\le$ relation if the criterion Q_i is minimised and the $\ge$ relation if it is maximised.

There are at least three criteria of compression quality: the compression ratio, the compression speed, and the decompression speed. We measure the compression ratio by the mean number of output bits per input symbol (bpc), so the smaller is the compression ratio, the better the compression is. Therefore we minimise this criterion. The speed of compression and decompression is measured in kB/s, so these criteria are maximised.

4.3 Experiments with the algorithm stages

4.3.1 Burrows–Wheeler transform computation

The Burrows–Wheeler transform can be computed in many ways. A brief summary of the possibilities is shown in Figure 3.12. In Section 4.1.2, we introduced an improvement to the Itoh–Tanaka's computation method. Here we examine the existing approaches in practice. Tables 4.5 and 4.6 contain the experimental results of several BWT computation methods on the files from the Calgary and the Silesia corpora.

We compare the methods for which either the source code or the in-depth description of the implementation are available. The columns of Tables 4.5 and 4.6

File	Size [B]	% runs	Sew	iSew	LS	MM	IT1S	IT2S	IT1iS	IT2iS	iIT1	iIT2	CIT
bib	111,261	2.26	0.06	0.06	0.08	0.16	0.05	0.05	0.04	0.05	0.05	0.05	0.05
book1	768,771	2.17	0.54	0.52	1.08	2.25	0.04	0.36	0.38	0.34	0.37	0.34	0.34
book2	610,856	2.17	0.40	0.38	0.80	1.71	0.29	0.28	0.27	0.25	0.28	0.25	0.25
geo	102,400	4.11	0.06	0.06	0.08	0.11	0.05	0.06	0.05	0.05	0.05	0.06	0.05
news	377,109	6.20	0.21	0.20	0.43	1.05	0.19	0.18	0.16	0.16	0.15	0.16	0.15
obj1	21,504	20.69	0.03	0.03	0.01	0.03	0.15	0.12	0.08	0.08	0.03	0.08	0.03
obj2	246,814	5.42	0.13	0.13	0.24	0.55	0.11	0.11	0.10	0.11	0.10	0.10	0.10
paper1	53,161	2.35	0.03	0.04	0.03	0.05	0.03	0.04	0.03	0.04	0.03	0.03	0.03
paper2	82,199	1.85	0.05	0.05	0.06	0.10	0.03	0.04	0.04	0.04	0.04	0.04	0.04
pic	513,216	85.20	0.15	0.12	0.59	1.16	183.97	186.51	224.50	202.46	0.10	204.24	0.10
progc	39,611	7.65	0.04	0.03	0.02	0.04	0.03	0.03	0.03	0.03	0.03	0.03	0.03
progl	71,646	15.61	0.04	0.04	0.05	0.09	0.05	0.05	0.04	0.05	0.04	0.05	0.04
progp	43,379	15.48	0.04	0.04	0.03	0.06	0.03	0.04	0.04	0.04	0.03	0.07	0.03
trans	93,695	8.86	0.06	0.06	0.07	0.13	0.05	0.06	0.05	0.05	0.04	0.05	0.04

Table 4.5: Comparison of different BWT computation methods for the Calgary corpus. The computation times are expressed in seconds.

File	Size [B]	% runs	Sew	iSew	LS	MM	IT1S	IT2S	IT1iS	IT2iS	iIT1	iIT2	CIT
dickens	10,192,446	2.22	12.63	12.29	20.96	68.30	9.96	7.99	9.38	7.55	9.22	7.14	7.14
mozilla	51,220,480	18.32	64.27	58.68	107.63	362.16	> 500	> 500	> 500	> 500	45.91	> 500	45.95
mr	9,970,564	28.34	10.15	9.50	25.07	62.66	> 500	> 500	> 500	> 500	7.49	> 500	7.50
nci	33,553,445	33.33	188.68	66.96	138.95	299.80	> 500	> 500	> 500	> 500	56.20	> 500	56.22
ooffice	6,152,192	10.54	5.53	5.38	9.08	31.68	22.06	21.01	23.75	22.03	3.86	22.10	3.86
osdb	10,085,684	5.89	14.48	12.95	21.81	48.05	11.70	10.09	9.73	8.64	9.49	8.64	8.64
reymont	6,627,202	3.34	8.17	7.42	15.55	33.54	6.58	5.11	5.50	4.34	5.79	4.31	4.31
samba	21,606,400	13.33	26.80	23.70	51.73	130.05	244.39	235.92	53.03	46.35	18.53	46.59	18.56
sao	7,251,944	1.10	8.88	8.75	11.50	24.44	6.63	7.06	6.23	6.60	6.45	6.42	6.42
webster	41,458,703	1.25	87.98	64.65	118.57	346.86	85.48	59.69	57.29	42.14	53.85	42.09	42.16
xml	5,345,280	2.89	7.18	5.41	12.13	28.58	8.95	9.20	6.59	6.11	4.72	5.82	5.82
x-ray	8,474,240	0.45	8.37	7.87	12.77	29.90	6.25	5.99	5.86	5.75	6.01	5.68	5.68

Table 4.6: Comparison of different BWT computation methods for the Silesia corpus. The computation times are expressed in seconds.

contain the time of the BWT computation expressed in seconds for the following methods:

- Sew—the *copy* method by Seward [149]; this method is implemented in the *bzip2* program [150]; our implementation is based on the original one by Seward,
- iSew—the improved version of Seward's method; the improvements were done in implementation only (a brief description of the improvements can be found in Appendix B),
- LS—the Larsson–Sadakane's method [102],
- MM—the Manber–Myers's method [106],
- IT1S—the basic Itoh–Tanaka's method [90] of order 1 with Seward's [149] sorting suffixes method,
- IT2S—the basic Itoh–Tanaka's method [90] of order 2 with Seward's [149] for sorting suffixes method,
- IT1iS—the basic Itoh–Tanaka's method [90] of order 1 with improved Seward's method for sorting suffixes,
- IT2iS—the basic Itoh–Tanaka's method [90] of order 2 with improved Seward's method for sorting suffixes,
- iIT1—the improved Itoh–Tanaka's method (Section 4.1.2) of order 1 with improved Seward's method for sorting suffixes,
- iIT2—the improved Itoh–Tanaka's method (Section 4.1.2) of order 2 with improved Seward's method for sorting suffixes,
- CIT—the combined method which chooses between the iIT1 and the iIT2 method depending on contents of the sequence to process.

The computation times of the methods for the Calgary corpus are small. This is caused by the short test files. The most interesting observation for this corpus is the slowness of some methods processing the pic file. The main problem of all the sorting-based methods (all of the examined except for LS and MM) is sorting strings with long identical prefixes. The simplest way to find such files is to count the number of runs occurrence. This method is not perfect, however it is easy to calculate and, as we can observe from the experiments, it is a good *ad hoc* rule to choose files hard to process. The column denoted % *runs* contains the percentage number of runs of length 2. As we can see, this ratio is over 85% for the pic file. The other file for which this value is large is obj1. The computation times for this file are small, because of its size, but we can notice that most IT-based methods work on it 5 times slower than the Sew method. When we focus on the methods for suffix arrays construction, whose good worst-case complexities are known (LS and MM methods), we can see, that in practice these methods are slower than the sorting-based ones.

The files in the Silesia corpus are larger. Therefore the differences between the computation times for different methods are higher. As we can see, the Larsson–Sadakane's and the Manber–Myers's methods are significantly slower than the Seward's method. There are three files, mozilla, mr, and nci, for which some IT-based methods work very slow. In fact, we stopped the experiments after 500 seconds, because we decided this time to be impractical, as there are much faster other methods. The Itoh–Tanaka-based methods employ the Seward's method for sorting suffixes, but there is one important difference between the employed method and the Seward's method. Seward introduced a highly efficient method for sorting runs. In the original Itoh–Tanaka's method, we cannot use this improvement. Therefore, for the files with long runs (all the mentioned files have the ratio of runs of length 2 over 15%) this method is slow. One of the improvements proposed in Section 4.1.2 is a special method for sorting runs. Unfortunately, this special treating of runs cannot be applied in the Itoh–Tanaka's method of order 2. As we can notice, the improvement of speed gained by this innovation is significant for all files, and for files hard to sort it is huge. The improved Itoh–Tanaka's method of order 2 works, however, faster for sequences with small number of runs. Therefore, we propose to use the order-2 version if the ratio of runs of length 2 is lower than 7%. Such a combined method (column CIT), choosing between iIT1 and iIT2, is used in further experiments.

4.3.2 Weight functions in the weighted frequency count transform

In this experiment, we examine the weight functions w. The comparison of these functions is made using the files from the Calgary corpus. The results are presented in Table 4.7.[§] For each of the examined weight functions, the best set of parameters were found. We see that the weight functions w_8 and w_9 achieve the best compression ratios. The usage of different parameters for different ranges in the weight function w_8 is motivated by the observation that typically characters in these ranges are from different, but similar, contexts. It is useful to exploit the information on the probability distribution in such contexts, but it should not dominate the probability distribution of the current context. The parameter q for the weight function w_9 is the most complex. In this weight function, the number of different contexts of length 4 that appear in the input sequence is used (the parameter C_4) to exploit some information on the structure of the sequence x.

[§] The results in this dissertation are slightly different from the ones that were presented in References [54, 55] because a slightly modified probability estimation method was used.

File	Size [B]	w_1	w_2 $q = 0.7$	w_3 $p = 4$	w_4 $p = 4$	w_5 $p = 0.5$ $q = -1.25$	w_6 $p = 1.0$ $q = 0.95$	w_7 $p = -1.40$ $q = 0.999$	w_8 $p = 4$	w_9 $p = 3$ $q = 1 - 100/C_4$	w_{9q} $p = 3$ $q = 1 - 100/C_4$
bib	111,261	1.914	1.917	1.969	1.914	1.898	1.908	1.895	1.896	**1.891**	1.892
book1	768,771	2.343	2.311	2.283	2.282	2.278	2.279	2.278	2.272	2.267	**2.266**
book2	610,856	1.998	1.980	2.000	1.972	1.962	1.962	1.957	1.958	**1.954**	1.957
geo	102,400	4.234	4.229	**4.114**	4.119	4.142	4.204	4.171	4.147	4.157	4.139
news	377,109	2.463	2.462	2.464	2.415	2.410	2.437	2.415	**2.408**	**2.408**	2.409
obj1	21,504	3.760	3.754	3.722	3.691	**3.690**	3.725	3.702	3.692	3.705	3.701
obj2	246,814	2.436	2.448	2.489	2.430	2.414	2.433	2.415	2.411	**2.410**	2.417
paper1	53,161	2.420	2.422	2.487	2.424	2.405	2.415	2.401	2.403	**2.400**	**2.400**
paper2	82,199	2.381	2.370	2.405	2.363	2.350	2.355	2.344	2.347	**2.343**	2.345
pic	513,216	0.759	0.741	**0.703**	0.706	0.716	0.726	0.721	0.718	0.718	0.714
progc	39,611	2.453	2.461	2.518	2.449	2.430	2.449	2.429	2.430	**2.427**	2.428
progl	71,646	1.684	1.699	1.768	1.681	1.672	1.701	1.678	1.670	**1.669**	1.671
progp	43,379	**1.667**	1.691	1.784	1.690	1.673	1.701	1.679	1.672	1.669	1.670
trans	93,695	**1.451**	1.487	1.608	1.467	1.457	1.504	1.474	**1.451**	**1.451**	1.454
Average		2.283	2.284	2.308	2.257	2.250	2.271	2.254	2.248	2.248	**2.247**
Std. dev.		0.880	0.877	0.843	0.857	0.861	0.869	0.865	0.862	0.866	0.862

Table 4.7: Comparison of the weight functions for the Calgary corpus. For all functions the value $t_{max} = 2048$ was used. The compression ratios are expressed in bpc.

File	Size [B]	MTF	MTF-1	MTF-2	TS(0)	IF	DC	BS99	WM01	A02	WFC
bib	111,261	1.914	1.906	1.906	2.012	1.963	1.930	1.91	1.951	1.96	**1.892**
book1	768,771	2.343	2.320	2.306	2.309	2.239	**2.224**	2.27	2.363	**2.22**	2.266
book2	610,856	1.998	1.985	1.977	2.028	1.964	**1.927**	1.96	2.013	1.95	1.957
geo	102,400	4.234	4.221	4.221	4.186	4.190	4.497	4.16	4.354	4.18	**4.139**
news	377,109	2.463	2.453	2.451	2.587	2.459	**2.392**	2.42	2.465	2.45	2.409
obj1	21,504	3.760	3.741	3.743	3.900	3.889	3.948	3.73	3.800	3.88	**3.701**
obj2	246,814	2.436	2.429	2.431	2.637	2.548	2.448	2.45	2.462	2.54	**2.417**
paper1	53,161	2.420	2.414	2.413	2.589	2.454	**2.398**	2.41	2.453	2.45	2.400
paper2	82,199	2.381	2.373	2.367	2.458	2.366	**2.334**	2.36	2.416	2.36	2.345
pic	513,216	0.759	0.742	0.738	0.733	0.706	0.713	0.72	0.768	**0.70**	0.714
progc	39,611	2.453	2.450	2.453	2.644	2.500	2.469	2.45	2.469	2.50	**2.428**
progl	71,646	1.684	1.680	1.683	1.853	1.747	1.689	1.68	1.678	1.74	**1.671**
progp	43,379	1.667	**1.666**	1.671	1.889	1.745	1.700	1.68	1.692	1.74	1.670
trans	93,695	1.451	**1.449**	1.453	1.710	1.557	1.473	1.46	1.484	1.55	1.454
Average		2.283	2.274	2.272	2.395	2.309	2.296	2.26	2.312	2.30	**2.247**
Std. dev.		0.880	0.878	0.879	0.866	0.885	0.954	0.868	0.901	0.885	0.862

Table 4.8: Comparison of the second stage methods for the Calgary corpus. The compression ratios are expressed in bpc.

We see that the weight function w_7, for which the best results were obtained in the numerical analysis of a simplified case (see Section 4.1.4), does not lead to the best compression ratios. The weight function w_9 is, however, only a slightly modified version of the weight function w_6, which was the second best in the numerical analysis.

In subsequent experiments, we use the weight function w_{9q}, which is the quantised version of the function w_9, and led to the best compression ratios among the quantised weight functions. As we can see, the disadvantage caused by the quantisation can be neglected in this case (in fact, we have achieved a slight improvement in the average ratio). For the other weight functions this difference may not be, however, that small.

4.3.3 Approaches to the second stage

In the next experiment, we compare the various second stage methods (Tables 4.8 and 4.9). The compression results in the columns denoted by MTF, MTF-1, MTF-2, TS(0), and WFC are obtained using the compression algorithm introduced in this dissertation and presented in Figure 4.1, where the WFC transform is replaced by the mentioned transforms. For the rest of the transforms, a different probability estimation is needed and we cannot replace only the second stage. The results for the IF transform are obtained using the algorithm in which also a different probability estimation method is applied.

The other second stage approaches are not described in the literature precisely enough, and it is hard to implement them to achieve such results as pre-

File	Size [B]	MTF	MTF-1	MTF-2	TS(0)	IF	DC	WFC
dickens	10,192,446	1.904	1.883	1.868	1.852	1.794	**1.781**	1.816
mozilla	51,220,480	2.629	2.603	2.600	2.631	2.616	2.641	**2.543**
mr	9,970,564	1.826	1.823	1.822	1.780	1.776	1.806	**1.770**
nci	33,553,445	0.301	0.299	0.299	0.306	0.298	0.316	**0.297**
ooffice	6,152,192	3.484	3.461	3.459	3.505	3.473	3.465	**3.380**
osdb	10,085,684	1.839	1.830	1.828	**1.815**	1.933	2.002	1.835
reymont	6,627,202	1.258	1.246	1.237	1.243	**1.218**	1.222	1.224
samba	21,606,400	1.518	**1.517**	1.518	1.594	1.554	1.574	1.518
sao	7,251,944	5.367	5.298	5.294	5.226	5.248	5.306	**5.195**
webster	41,458,703	1.332	1.319	1.309	1.308	1.263	1.325	**1.276**
xml	5,345,280	0.596	**0.595**	**0.595**	0.639	0.619	0.606	0.602
x-ray	8,474,240	3.577	3.576	3.575	3.521	**3.500**	3.606	3.518
Average		2.136	2.121	2.117	2.118	2.108	2.138	**2.081**
Std. dev.		1.425	1.411	1.411	1.390	1.397	1.410	1.380

Table 4.9: Comparison of the second stage methods for the Silesia corpus. The compression ratios are expressed in bpc.

sented by their authors. Therefore the reasonable choice is to use the published results. The compression ratios for the distance coder transform (DC) are taken from the experiments with its currently best implementation, *ybs* [200]. The results for the BS99 solution are presented following the work by Balkenhol and Shtarkov [18]. The column denoted by WM01 contains the compression ratios of the best compression algorithm by Wirth and Moffat [195] where no second stage transform is used. The column denoted by A02 presents the results for the other implementation of the IF transform following Arnavut [9].

The comparison shows that different transforms are the best for different files from the Calgary and the Silesia corpora, but most of the top results are achieved using the WFC transform. One should, however, remember that improving the probability estimation for the IF, DC, BS99, WM01, or A02 methods may change this result.

An interesting observation is that the orderings of the performance of the transforms for the Calgary corpus and for the Silesia corpus are different. For the Calgary corpus the WFC transform is the best, but the next places take MTF-2, MTF-1, and MTF transforms. The IF and TS(0) transforms give worse results with respect to the compression ratio. For the Silesia corpus the transforms IF and TS(0) are better or almost equal with respect to the compression ratio to the MTF family transforms. It suggests that these transforms work better for long sequences.

Method	Calgary corpus	Silesia corpus
0-order	2.271	2.096
1-order	2.265	2.091
2-order	2.264	2.088
3-order	2.266	2.088
4-order	2.270	2.088
0-1-weighted	2.249	2.083
0-2-weighted	**2.247**	**2.081**
0-3-weighted	2.248	**2.081**
0-4-weighted	2.249	**2.081**
1-2-weighted	2.252	2.083
1-3-weighted	2.251	2.082
1-4-weighted	2.251	**2.081**
2-3-weighted	2.256	2.084
2-4-weighted	2.256	2.083
3-4-weighted	2.261	2.086

Table 4.10: Average compression ratios for various methods of probability estimation. The compression ratios are expressed in bpc.

4.3.4 Probability estimation

In Section 4.1.6, we introduced a method for probability estimation based on weighting between estimations of two context lengths. Now we are going to examine for which context lengths for the compression ratios are the best. The empirical results are shown in Table 4.10.

The first five rows of the table show that the best choice of order is $d = 2$, if no weighting is used (Equation 4.54). Better results can be, however, obtained if the weighted probability is used (Equation 4.55). The experiments show that we obtain the best results when the orders are set to $d_1 = 0$ and $d_2 = 2$. We can also notice that the improvement in the compression ratio is larger for the Calgary corpus than for the Silesia corpus. The proper choice of orders d_1 and d_2 is also more important for the Calgary corpus.

4.4 Experimental comparison of the improved algorithm and the other algorithms

4.4.1 Choosing the algorithms for comparison

Perhaps the most interesting experiment is to compare the efficiency of the improved algorithm proposed in this dissertation and some of the state of the art algorithms. We examine in this section many compression methods. Some of them are described in the literature, but no executables for them are available.

For these algorithms we cannot provide the experimental compression results for the Silesia corpus. The other ones are available for downloading so we can examine them on both corpora.

The dissertation concerns universal compression, so we have to exclude from the comparison the compression programs that are highly tuned to files from the Calgary corpus. Some of such compressors use a different compression method depending on what they recognise as a file to process. For example, the pic and geo files can be better compressed if they undergo a special transformation before the compression. There exist compressors that monitor the size of the file and if it is exactly the same as the size of files from the Calgary corpus, they make such a transformation. Other programs monitor the contents of the file and if they recognise that the file is an English text, they run some preliminary filters, which are similar to those described in Section 3.2.6. Many files of the Calgary corpus are English texts, so such a heuristics gives an advantage for this corpus. There are also methods for improving the compression ratio for executable files, and some compression programs also use a heuristic to recognise such files.

When it was possible, such data-specific preliminary filtering methods were turned off in the programs. Unfortunately, in some compressors there is no way to abandon filters and we have to resign from including them in the experiments.

4.4.2 Examined algorithms

Before we go to the comparison of the compression algorithms, we expand the abbreviations of the examined methods:

- A02—the BWT-based algorithm with the inversion frequencies transform as the second stage; the results are from the work by Arnavut [9];
- acb—the Associative Coder by Buyanovsky [40]; the compression results are from experiments with the *acb 2.00c* program [41];
- B97—the best version of the PPM algorithms proposed by Bunton [37];
- boa—the *boa 0.58b* compression program by Sutton [160], which is an implementation of the PPM algorithm;
- BS99—the BWT-based algorithm invented by Balkenhol and Shtarkov [18];
- BW94—the original Burrows–Wheeler compression algorithm [39];
- bzip—the *bzip 0.21* compression program by Seward [148]; this program achieves the same compression ratios as Fenwick's method [68];
- CTW—context tree weighting method proposed by Willems *et al.* [189]; the results are from Reference [17];

- DM—the improved BWT-based algorithm proposed in this dissertation with the move-to-front transform as the second stage;
- DW—the improved BWT-based algorithm proposed in this dissertation with the weighted frequency count transform as the second stage;
- F96—the BWT-based algorithm proposed by Fenwick [68]; the same compression ratios are achieved by the *bzip 0.21* program [148], which is used for experiments on the Silesia corpus;
- gzip—standard *gzip* program[72]; this is an implementation of the well known LZ77 algorithm [202];
- LDMC—currently the best dynamic Markov coding algorithm, originally introduced by Cormack and Horspool [52]; this is an improved version, LazyDMC, by Bunton [33];
- lgha—the speed-optimised *ha* compression program; this implementation is provided by Lyapko [105];
- LZMA—the Ziv–Lempel algorithm presented by Pavlov [127]; the results are from experiments with the *7-zip* program;
- LZW—standard UNIX *compress* program; this is an implementation of the LZW algorithm [187];
- MH—the cPPMII algorithm proposed by Shkarin [154, 155]; the results are from experiments with the *PPMonstr var. H* program;
- MI4—the cPPMII algorithm by Shkarin [156]; the result are from experiments with the *PPMonstr var. I* program with order 4;
- MI64—the cPPMII algorithm by Shkarin [156]; the result are from experiments with the *PPMonstr var. I* program with order 64;
- PPMdH—the PPMII algorithm proposed by Shkarin [154, 155]; the results are from experiments with the *PPMd var. H* program;
- PPMN—the PPMN algorithm by Smirnov [158]; the results are from experiments with the *ppmnb1+* program;
- rar—the *rar 2.90* compression program [137];
- szip—the BWT-based algorithm presented by Schindler [144]; the results are from experiments with the *szip* program [145];
- T98—the PPMD+ algorithm proposed by Teahan [163]; the results are from experiments with the *ppmd+* program [162];

- ufa—the binary PPM algorithm by Pavlov [126]; the results are from experiments with the *ufa 0.04 Beta 1* program;

- VW98—the switching method, algorithm presented by Volf and Willems [182, 183];

- WM01—the best BWT-based algorithm with no second stage; the results are from the work by Wirth and Moffat [195];

- ybs—the BWT-based compression program with the distance coder (DC) transform as the second stage; this is an implementation by Yoockin [200].

4.4.3 Comparison procedure

All the mentioned compression algorithms were compared on two corpora—the Calgary corpus and the Silesia corpus. For each file of the corpora, all the compression programs were run and the size of the compressed file, the compression time, and the decompression time were measured. The experiments were performed on a computer equipped with the AMD Athlon 1.4 GHz processor, 512 MB DDRAM, and Microsoft Windows 2000 Professional operating system. The times of program execution were measured by using the program utility *ntimer*. The presented times are user times, as we want to know only the time of compression not the I/O operations times.

For comparison, we calculated two compression ratios. The first one is the standard compression ratio expressed in output bits per input character (bpc). The second one is a normalised compression ratio. We calculated it by dividing the standard compression ratio by the best standard compression ratio of all the examined compression methods. The goal of using also normalised compression ratio is that each file can be compressed to the smallest possible number of bits, namely its entropy. We do not know, however, the entropy. The best what we can do is to approximate it by the length of the shortest compressed sequence. Therefore, the normalised compression ratio says how far each compression algorithm is from the best one. To compare the algorithms in terms of multi criteria optimisation, we calculated also the compression and decompression speed. For each file these speeds were found and then the average were calculated.

We mentioned about large differences between the three main families of algorithms in modern compression: the Ziv–Lempel methods, the prediction by partial matching methods, and the Burrows–Wheeler transform-based methods. For a clear distinction we denote these algorithms by different marks at the figures. The LZ methods are denoted by $\circ$, the PPM methods are denoted by $\star$, and the BWT-based methods by $\bullet$.

The compression programs were examined with many combinations of parameters. For some of them, however, the number of available sets of parameters is vast, and it is impossible to examine all possibilities. The compression results presented in the literature are in most cases optimised for the compression ratio. Therefore, for possibility of comparison to the published methods, for which there are no executables available, we decided to choose such a set of parameters that leads us to the best compression ratio. If, however, there can be found an interesting set of parameters, for example such a set for which a given algorithm dominates some other algorithms that it does not dominate for the chosen set, we also included these combination of parameters in comparison. This situation took place for Shkarin's PPM algorithms, for which we examined four set of parameters.

The data-specific compression options, like English text filters, were turned off. The maximum size of memory available for the compression was chosen to provide an honest comparison of algorithms. The memory limit equal to 10 times the size of the file to compress (but not less than 16 MB) was decided to be a good choice. This is a natural memory requirement for the BWT-based compression algorithms and the PPM methods significantly differ in the memory consumption from each other (with the growth of the context length the memory needed to store the model grows fast and is not naturally bounded). A more detailed description of the options of the programs used in the experiments is presented in Appendix C.

4.4.4 Experiments on the Calgary corpus

The Calgary corpus is a standard corpus for examining the compression algorithms. There are a number of compression methods for which the compression ratios can be found in the literature. Therefore, we can compare the best compression methods proposed so far.

Table 4.11 contains the standard compression ratios for files from the Calgary corpus, and Table 4.12 contains the normalised compression ratios. The best PPM algorithms achieve on this corpus significantly better compression ratios than the LZ and the BWT-based methods. If, however, we look at Tables 4.13 and 4.14, we see that this high efficiency is occupied by the low speed of these algorithms.

Better comparison can be done if we look at Figures 4.15, 4.16, 4.17, and 4.18. (The figures do not contain data for some algorithms presented in Table 4.11 as there are no executables for them, and we could not measure their compression and decompression speed.)

We can see that in general, the PPM algorithms achieve the best compression ratio, but they are slow (Figures 4.15 and 4.16). The fastest compression algorithms are the LZ methods, but their compression ratio is poor. The BWT-

File	Size [B]	gzip	LZW	LZMA	BW94	F96	szip	bwc	BS99	ybs	WM01	A02	CTW	VW98
bib	111,261	2.509	3.346	2.202	2.07	1.95	1.969	1.968	1.91	1.930	1.951	1.96	1.79	1.714
book1	768,771	3.250	3.300	2.717	2.49	2.39	2.348	2.401	2.27	2.224	2.363	2.22	2.19	2.150
book2	610,856	2.700	3.291	2.224	2.13	2.04	2.020	2.045	1.96	1.927	2.013	1.95	1.87	1.820
geo	102,400	5.345	6.076	4.183	4.45	4.50	4.308	4.297	4.16	4.497	4.354	4.18	4.46	4.526
news	377,109	3.063	3.896	2.522	2.59	2.50	2.480	2.506	2.42	2.392	2.465	2.45	2.29	2.210
obj1	21,504	3.839	5.226	3.526	3.98	3.87	3.779	3.823	3.73	3.948	3.800	3.88	3.68	3.607
obj2	246,814	2.628	4.170	1.997	2.64	2.46	2.464	2.487	2.45	2.448	2.462	2.54	2.31	2.245
paper1	53,161	2.791	3.774	2.607	2.55	2.46	2.495	2.474	2.41	2.398	2.453	2.45	2.25	2.152
paper2	82,199	2.887	3.519	2.658	2.51	2.41	2.432	2.439	2.36	2.334	2.416	2.36	2.21	2.136
pic	513,216	0.817	0.970	**0.652**	0.83	0.77	0.767	0.797	0.72	0.713	0.768	0.70	0.79	0.764
progc	39,611	2.678	3.866	2.545	2.58	2.49	2.506	2.494	2.45	2.469	2.469	2.50	2.29	2.195
progl	71,646	1.805	3.031	1.678	1.80	1.72	1.706	1.700	1.68	1.689	1.678	1.74	1.56	1.482
progp	43,379	1.812	3.112	1.685	1.79	1.70	1.735	1.709	1.68	1.700	1.692	1.74	1.60	1.460
trans	93,695	1.611	3.265	1.432	1.57	1.50	1.512	1.498	1.46	1.473	1.484	1.55	1.34	1.256
Average		2.695	3.632	2.331	2.43	2.34	2.323	2.331	2.26	2.296	2.312	2.30	2.19	2.123
Std. dev.		1.079	1.148	0.872	0.917	0.932	0.886	0.888	0.868	0.954	0.901	0.885	0.924	0.949

File	LDMC	T98	B97	PPMN	PPMdH	MH	MI4	MI64	lgha	acb	rar	boa	ufa	**DM**	**DW**
bib	2.018	1.862	1.786	1.739	1.732	1.680	1.806	**1.661**	1.938	1.936	2.388	1.738	1.937	1.906	1.892
book1	2.298	2.303	2.184	2.200	2.198	2.135	2.222	**2.120**	2.453	2.317	3.102	2.205	2.360	2.306	2.266
book2	2.030	1.963	1.862	1.848	1.838	1.783	1.891	**1.745**	2.142	1.936	2.538	1.860	2.048	1.977	1.957
geo	4.484	4.733	4.458	4.376	4.346	4.160	3.888	**3.868**	4.639	4.556	5.235	4.426	4.385	4.221	4.139
news	2.534	2.355	2.285	2.223	2.195	2.142	2.232	**2.088**	2.617	2.318	2.887	2.222	2.508	2.451	2.409
obj1	3.830	3.728	3.678	3.559	3.541	3.504	3.364	**3.346**	3.657	3.504	3.672	3.619	3.845	3.743	3.701
obj2	2.560	2.378	2.283	2.185	2.174	2.118	2.030	**1.891**	2.606	2.201	2.432	2.232	2.571	2.431	2.417
paper1	2.525	2.330	2.250	2.225	2.196	2.147	2.206	**2.121**	2.364	2.344	2.729	2.224	2.481	2.413	2.400
paper2	2.429	2.315	2.213	2.201	2.182	2.126	2.185	**2.112**	2.335	2.338	2.783	2.203	2.407	2.367	2.345
pic	0.758	0.795	0.781	0.728	0.756	0.715	0.721	0.668	0.805	0.745	0.759	0.747	0.812	0.738	0.714
progc	2.575	2.363	2.291	2.253	2.208	2.165	2.197	**2.104**	2.388	2.333	2.661	2.261	2.509	2.453	2.428
progl	1.822	1.677	1.545	1.474	1.443	1.401	1.547	**1.343**	1.712	1.505	1.764	1.484	1.787	1.683	1.671
progp	1.840	1.696	1.531	1.503	1.456	1.417	1.541	**1.336**	1.706	1.503	1.751	1.464	1.807	1.671	1.670
trans	1.708	1.467	1.325	1.259	1.226	1.188	1.377	**1.136**	1.533	1.294	1.547	1.244	1.547	1.453	1.454
Average	2.387	2.283	2.177	2.127	2.107	2.049	2.086	**1.967**	2.350	2.202	2.589	2.138	2.357	2.272	2.247
Std. dev.	0.903	0.960	0.930	0.916	0.911	0.884	0.786	0.829	0.928	0.936	1.055	0.934	0.894	0.879	0.862

Table 4.11: Compression ratios (in bpc) of the algorithms for the Calgary corpus

File	Size [B]	gzip	LZW	LZMA	BW94	F96	szip	bwc	BS99	ybs	WM01	A02	CTW	VW98
bib	111,261	1.511	2.014	1.326	1.246	1.173	1.185	1.185	1.150	1.162	1.175	1.180	1.078	1.032
book1	768,771	1.533	1.644	1.282	1.175	1.130	1.108	1.133	1.071	1.049	1.115	1.047	1.033	1.014
book2	610,856	1.547	1.924	1.274	1.221	1.170	1.158	1.172	1.123	1.104	1.154	1.117	1.072	1.043
geo	102,400	1.382	1.571	1.081	1.150	1.158	1.114	1.111	1.075	1.163	1.126	1.081	1.153	1.170
news	377,109	1.467	1.882	1.208	1.240	1.200	1.188	1.200	1.159	1.146	1.181	1.173	1.097	1.058
obj1	21,504	1.147	1.562	1.054	1.189	1.157	1.129	1.143	1.115	1.180	1.136	1.160	1.100	1.078
obj2	246,814	1.390	2.238	1.056	1.396	1.303	1.303	1.315	1.296	1.295	1.302	1.343	1.222	1.187
paper1	53,161	1.315	1.779	1.229	1.202	1.161	1.176	1.166	1.136	1.131	1.157	1.155	1.061	1.015
paper2	82,199	1.367	1.666	1.259	1.188	1.144	1.152	1.155	1.117	1.105	1.144	1.117	1.046	1.011
pic	513,216	1.253	1.488	**1.000**	1.273	1.181	1.176	1.222	1.104	1.094	1.178	1.074	1.212	1.172
progc	39,611	1.273	1.837	1.210	1.226	1.188	1.191	1.185	1.164	1.173	1.173	1.188	1.088	1.043
progl	71,646	1.344	2.257	1.249	1.340	1.279	1.270	1.266	1.251	1.258	1.249	1.296	1.162	1.103
progp	43,379	1.356	2.329	1.261	1.340	1.277	1.299	1.279	1.257	1.272	1.266	1.302	1.198	1.093
trans	93,695	1.418	2.874	1.261	1.382	1.320	1.331	1.319	1.285	1.297	1.306	1.364	1.180	1.106
Average		1.379	1.933	1.196	1.255	1.203	1.199	1.204	1.165	1.174	1.190	1.186	1.122	1.080
Std. dev.		0.113	0.383	0.103	0.080	0.063	0.073	0.067	0.076	0.079	0.064	0.102	0.064	0.061

File	LDMC	T98	B97	PPMN	PPMdH	MH	MI4	MI64	lgha	acb	rar	boa	ufa	**DM**	**DW**
bib	1.215	1.121	1.075	1.047	1.043	1.011	1.087	**1.000**	1.167	1.166	1.438	1.046	1.166	1.148	1.139
book1	1.084	1.086	1.030	1.038	1.037	1.007	1.048	**1.000**	1.157	1.093	1.463	1.040	1.113	1.088	1.069
book2	1.163	1.125	1.067	1.059	1.053	1.022	1.084	**1.000**	1.228	1.109	1.454	1.066	1.174	1.133	1.121
geo	1.159	1.224	1.153	1.131	1.124	1.075	1.005	**1.000**	1.199	1.178	1.353	1.144	1.134	1.091	1.070
news	1.214	1.128	1.094	1.065	1.051	1.026	1.069	**1.000**	1.253	1.110	1.383	1.064	1.201	1.174	1.154
obj1	1.145	1.114	1.099	1.064	1.058	1.047	1.005	**1.000**	1.093	1.047	1.097	1.082	1.149	1.119	1.106
obj2	1.354	1.258	1.207	1.155	1.150	1.120	1.074	**1.000**	1.378	1.164	1.286	1.180	1.360	1.286	1.278
paper1	1.190	1.099	1.061	1.049	1.035	1.012	1.040	**1.000**	1.115	1.105	1.287	1.049	1.170	1.138	1.132
paper2	1.150	1.096	1.048	1.042	1.033	1.007	1.035	**1.000**	1.106	1.107	1.318	1.043	1.140	1.121	1.110
pic	1.163	1.219	1.198	1.117	1.160	1.097	1.106	1.025	1.235	1.143	1.164	1.146	1.245	1.132	1.095
progc	1.224	1.123	1.089	1.071	1.049	1.029	1.044	**1.000**	1.135	1.109	1.265	1.075	1.192	1.166	1.154
progl	1.357	1.249	1.150	1.098	1.074	1.043	1.152	**1.000**	1.275	1.121	1.313	1.105	1.331	1.253	1.244
progp	1.377	1.269	1.146	1.125	1.090	1.061	1.153	**1.000**	1.277	1.125	1.311	1.096	1.353	1.251	1.250
trans	1.504	1.291	1.166	1.108	1.079	1.046	1.212	**1.000**	1.349	1.139	1.362	1.095	1.362	1.279	1.280
Average	1.236	1.172	1.113	1.084	1.074	1.043	1.080	**1.002**	1.212	1.123	1.321	1.088	1.221	1.170	1.157
Std. dev.	0.117	0.075	0.056	0.038	0.042	0.035	0.060	**0.007**	0.089	0.034	0.103	0.043	0.092	0.069	0.075

Table 4.12: Normalised compression ratios of the algorithms for the Calgary corpus

File	Size [B]	gzip	LZW	LZMA	F96	szip	bwc	ybs	PPMN	PPMdH
bib	111,261	0.03	0.06	0.34	0.09	0.08	0.15	0.08	0.21	0.09
book1	768,771	0.23	0.16	2.29	0.97	0.92	0.97	0.84	2.56	0.95
book2	610,856	0.13	0.14	1.80	0.66	0.65	0.73	0.60	1.50	0.58
geo	102,400	0.06	0.05	0.18	0.11	0.07	0.16	0.08	0.47	0.13
news	377,109	0.08	0.11	0.94	0.36	0.36	0.46	0.30	1.07	0.37
obj1	21,504	0.01	0.03	0.05	0.03	0.02	0.07	0.02	0.08	0.03
obj2	246,814	0.08	0.07	0.66	0.22	0.18	0.28	0.19	0.50	0.18
paper1	53,161	0.02	0.04	0.13	0.04	0.04	0.08	0.03	0.14	0.04
paper2	82,199	0.03	0.04	0.19	0.07	0.06	0.13	0.05	0.19	0.06
pic	513,216	0.30	0.06	1.01	0.12	8.82	0.29	0.20	0.30	0.11
progc	39,611	0.02	0.04	0.10	0.02	0.03	0.11	0.03	0.11	0.03
progl	71,646	0.03	0.06	0.22	0.06	0.06	0.11	0.02	0.11	0.04
progp	43,379	0.03	0.03	0.16	0.04	0.06	0.08	0.02	0.09	0.04
trans	93,695	0.02	0.04	0.27	0.07	0.08	0.13	0.06	0.14	0.06
Total	3,141,622	1.07	0.93	8.34	2.86	11.43	3.75	2.52	7.47	2.71
Avg. comp. speed		2875	2710	384	1346	1074	728	1615	511	1414
Std. dev.		1108	2034	73	867	357	330	723	359	951

File	Size [B]	MH	MI4	MI64	lgha	acb	rar	boa	ufa	**DM**	**DW**
bib	111,261	0.21	0.38	0.53	0.39	1.07	0.11	0.69	0.56	0.08	0.13
book1	768,771	1.81	2.27	4.10	1.54	22.60	0.73	8.00	3.60	0.67	0.89
book2	610,856	1.31	1.73	3.44	1.16	12.04	0.51	4.80	2.72	0.49	0.64
geo	102,400	0.29	1.01	1.01	0.60	1.55	0.14	1.44	0.78	0.14	0.18
news	377,109	0.76	1.45	2.48	0.96	6.01	0.34	3.14	1.92	0.33	0.43
obj1	21,504	0.06	0.20	0.21	0.11	0.19	0.04	0.19	0.14	0.04	0.04
obj2	246,814	0.44	1.10	1.41	0.62	2.16	0.17	1.54	1.25	0.23	0.30
paper1	53,161	0.11	0.21	0.28	0.20	0.45	0.05	0.29	0.29	0.06	0.08
paper2	82,199	0.16	0.30	0.41	0.20	0.76	0.08	0.46	0.43	0.07	0.11
pic	513,216	0.31	0.63	1.06	0.32	2.20	0.15	0.93	1.81	0.19	0.29
progc	39,611	0.08	0.18	0.21	0.12	0.27	0.05	0.20	0.22	0.04	0.06
progl	71,646	0.11	0.21	0.32	0.14	0.30	0.06	0.27	0.35	0.06	0.08
progp	43,379	0.08	0.16	0.25	0.14	0.24	0.03	0.18	0.24	0.04	0.07
trans	93,695	0.16	0.28	0.45	0.15	0.33	0.06	0.30	0.45	0.06	0.11
Total	3,141,622	5.89	10.11	16.16	6.65	50.17	2.52	22.43	14.76	2.50	3.41
Avg. comp. speed		572	295	195	458	131	1241	196	193	1187	821
Std. dev.		313	164	88	343	75	674	120	34	488	292

Table 4.13: Compression times (in seconds) of the algorithms for the Calgary corpus

File	Size [B]	gzip	LZW	LZMA	F96	szip	bwc	ybs	PPMN	PPMdH
bib	111,261	0.01	0.04	0.03	0.04	0.04	0.10	0.04	0.22	0.09
book1	768,771	0.03	0.10	0.08	0.45	0.42	0.44	0.36	2.58	0.97
book2	610,856	0.02	0.06	0.06	0.31	0.27	0.30	0.24	1.56	0.64
geo	102,400	0.01	0.03	0.03	0.07	0.06	0.10	0.06	0.45	0.15
news	377,109	0.02	0.10	0.05	0.18	0.17	0.25	0.15	1.11	0.36
obj1	21,504	0.01	0.03	0.02	0.02	0.02	0.07	0.02	0.08	0.03
obj2	246,814	0.02	0.05	0.03	0.07	0.07	0.13	0.08	0.53	0.21
paper1	53,161	0.01	0.03	0.03	0.02	0.03	0.09	0.03	0.12	0.05
paper2	82,199	0.01	0.03	0.03	0.03	0.03	0.08	0.03	0.20	0.08
pic	513,216	0.02	0.04	0.04	0.06	0.10	0.19	0.10	0.34	0.13
progc	39,611	0.02	0.02	0.02	0.02	0.01	0.07	0.02	0.11	0.04
progl	71,646	0.01	0.04	0.03	0.03	0.02	0.10	0.02	0.12	0.04
progp	43,379	0.01	0.02	0.02	0.01	0.02	0.07	0.01	0.10	0.04
trans	93,695	0.01	0.02	0.02	0.03	0.02	0.08	0.03	0.14	0.06
Total	3,141,622	0.21	0.61	0.49	1.34	1.28	2.07	1.19	7.66	2.89
Avg. comp. speed		12105	4308	5066	2860	2772	1194	2728	487	1271
Std. dev.		8989	3418	3667	1839	1161	660	1126	308	800

File	Size [B]	MH	MI4	MI64	lgha	acb	rar	boa	ufa	**DM**	**DW**
bib	111,261	0.21	0.39	0.55	0.22	0.91	0.03	0.59	0.58	0.06	0.10
book1	768,771	1.86	2.38	4.19	1.58	22.04	0.05	8.06	3.98	0.48	0.73
book2	610,856	1.30	1.82	3.53	1.16	12.07	0.04	4.88	3.08	0.33	0.51
geo	102,400	0.30	1.15	1.16	0.53	1.56	0.01	1.55	0.84	0.10	0.15
news	377,109	0.80	1.51	2.56	0.96	6.04	0.03	3.18	2.09	0.23	0.34
obj1	21,504	0.05	0.22	0.21	0.12	0.19	0.01	0.18	0.17	0.02	0.04
obj2	246,814	0.45	1.21	1.45	0.61	2.14	0.02	1.56	1.40	0.13	0.22
paper1	53,161	0.10	0.23	0.28	0.12	0.37	0.02	0.27	0.29	0.03	0.06
paper2	82,199	0.16	0.31	0.40	0.18	0.74	0.02	0.47	0.46	0.04	0.08
pic	513,216	0.34	0.70	1.13	0.05	2.19	0.02	1.06	2.05	0.13	0.25
progc	39,611	0.08	0.19	0.23	0.11	0.26	0.02	0.20	0.25	0.02	0.04
progl	71,646	0.13	0.23	0.34	0.12	0.35	0.02	0.26	0.39	0.03	0.05
progp	43,379	0.09	0.17	0.23	0.09	0.21	0.02	0.17	0.26	0.02	0.05
trans	93,695	0.15	0.30	0.46	0.14	0.34	0.02	0.27	0.48	0.04	0.08
Total	3,141,622	6.02	10.81	16.72	5.99	49.41	0.33	22.70	16.32	1.66	2.70
Avg. comp. speed		557	275	190	1118	134	8147	198	176	1946	1072
Std. dev.		273	148	82	2567	74	6937	110	30	690	300

Table 4.14: Decompression times (in seconds) of the algorithms for the Calgary corpus

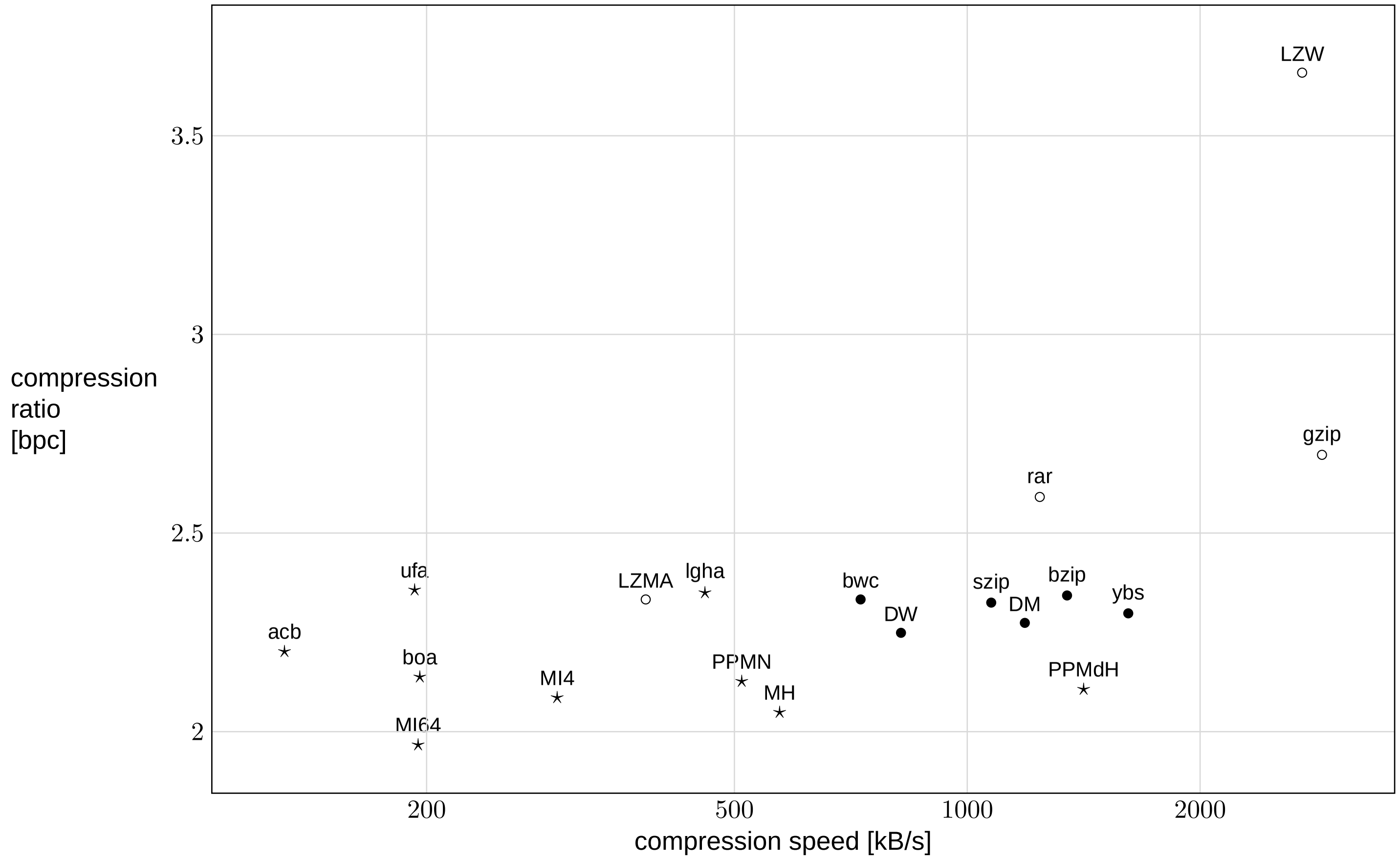

Figure 4.15: Compression ratio versus compression speed of the examined algorithms for the Calgary corpus. The LZ methods are denoted by ◦, the PPM methods are denoted by ⋆, and the BWT-based methods by •.

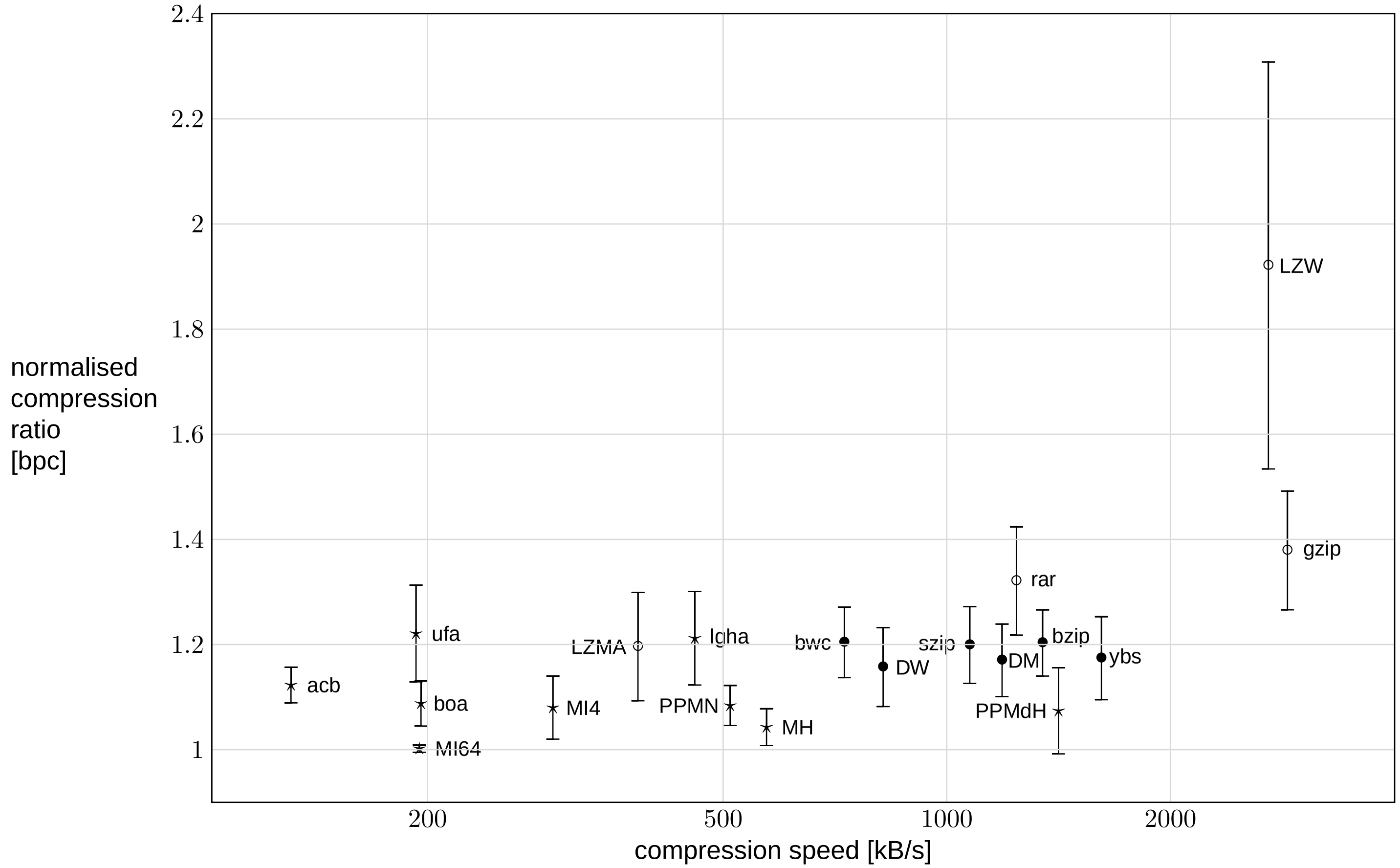

Figure 4.16: Normalised compression ratio versus compression speed of the examined algorithms for the Calgary corpus. The LZ methods are denoted by $\circ$, the PPM methods are denoted by $\star$, and the BWT-based methods by $\bullet$.

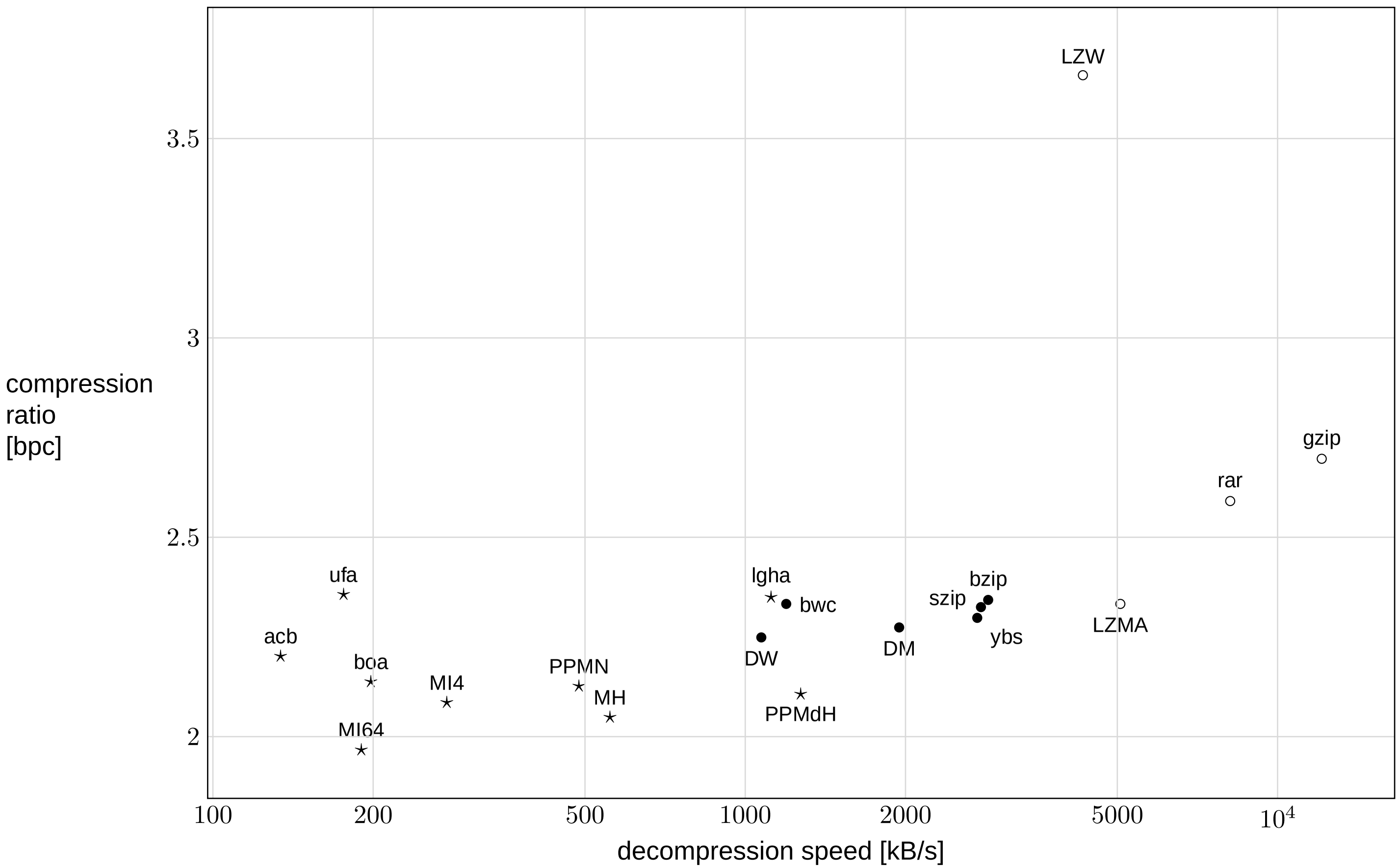

Figure 4.17: Compression ratio versus decompression speed of the examined algorithms for the Calgary corpus. The LZ methods are denoted by ◦, the PPM methods are denoted by ⋆, and the BWT-based methods by •.

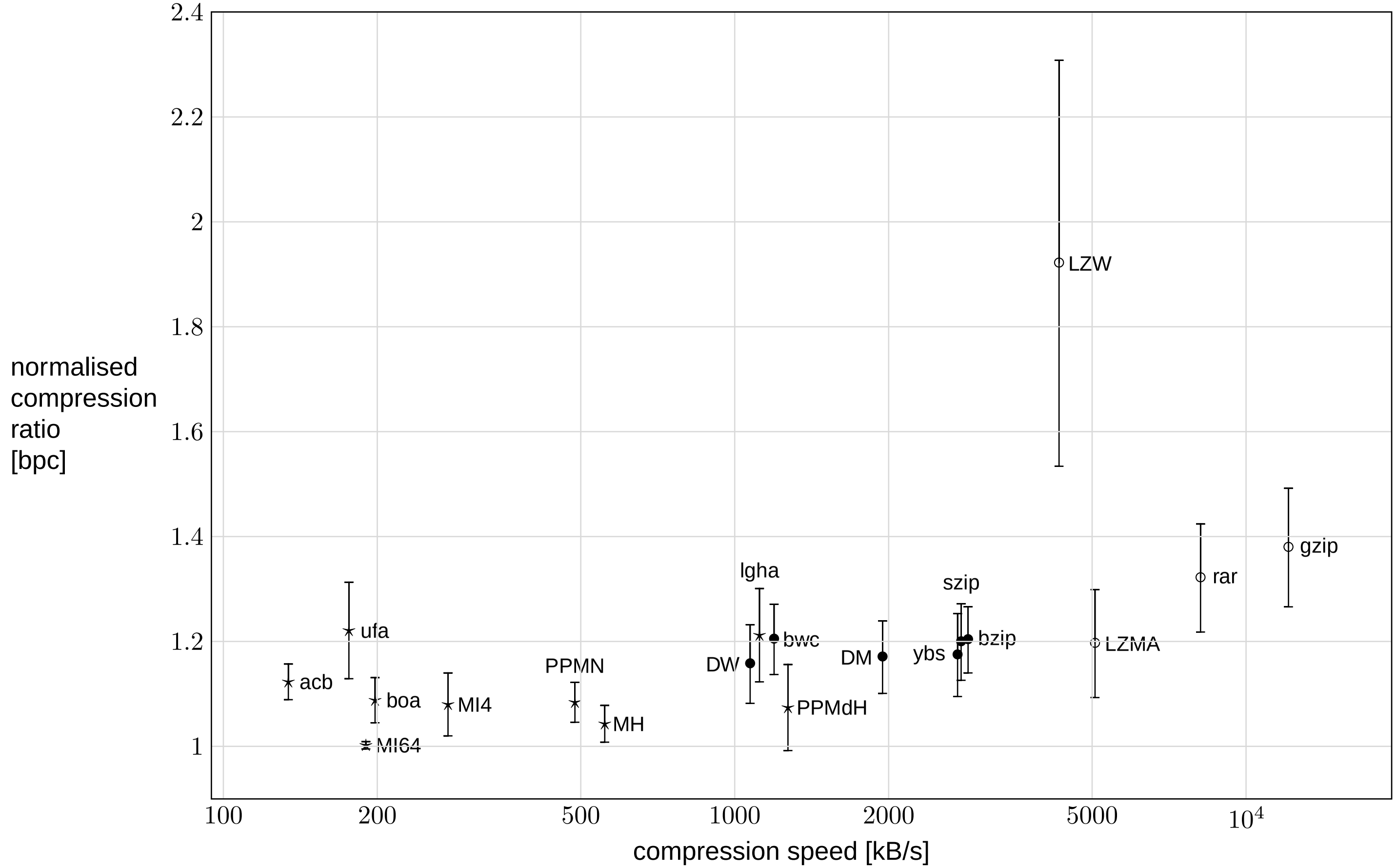

Figure 4.18: Normalised compression ratio versus decompression speed of the examined algorithms for the Calgary corpus. The LZ methods are denoted by ○, the PPM methods are denoted by ⋆, and the BWT-based methods by •.

based compression methods obtain compression ratios comparable to the PPM methods, but work significantly faster. The only exception is the recent PPMdH algorithm by Shkarin [155] published in 2002. It obtains very good compression ratios and compresses relatively fast. We see that this algorithm dominates the most of the BWT-based algorithms, however standard deviation of compression ratio and especially compression speed is much higher than for the BWT-based methods. We can see that also both the BWT-based algorithms proposed in this dissertation, the DW (BWT-based algorithm with the WFC as the second stage) and the DM (BWT-based algorithm with the MTF-2 as the second stage), obtain better compression ratios than other BWT-based algorithms and all the LZ methods.

Let us take a look at Figures 4.17 and 4.18, in which the speeds of decompression are compared to the compression ratios. We can observe that, the decompression speed of the PPM algorithms is almost identical to their compression speed. The LZ algorithms decompress much faster than compress. The difference between decompression and compression speed for the BWT-based algorithms is significant but lower than for the LZ algorithms. Now, the PPMdH algorithm dominates only two BWT-based methods, however standard deviations of their average decompression speed is much higher than for those algorithms. The DM algorithm is Pareto-optimal, while the DW algorithm leads to the best compression ratios in the family of the BWT-based methods. An interesting algorithm is the LZMA method, which achieves the best compression ratios among the LZ methods. The compression speed of this algorithm is low, it is over two times slower than the slowest BWT-based method, but with regards to the decompression speed it outperforms all the BWT-based algorithms, and also the LZW method.

As we can see, the results are almost identical if we analyse the standard compression ratio and the normalised compression ratio. The advantage of the normalised ratios is that the standard deviation of compression ratios are much lower. It is caused by the definition of this ratio, which entails that the best compression ratio can be 1.000. If we calculate the standard deviation of standard compression ratios, we should remember, that we compress files of different types, so the possible compression ratio can differ by a large factor. For example, no algorithm achieves better compression ratio than 3.800 bpc for geo file, while all the examined compression methods have no problem to break the 1.000 bpc ratio for pic file. The advantage of the standard compression ratio is that it does not change when new compression methods are introduced. The normalised compression ratio changes every time an algorithm that gives the best compression ratio for any file from the corpus is introduced. Many papers also contain compression results calculated as the standard compression ratio, so it is easy to use them for a brief comparison.

File	Size [B]	gzip	LZW	LZMA	bzip	szip	bwc	ybs	PPMN	PPMdH
dickens	10,192,446	3.023	3.145	2.222	2.178	1.968	1.952	1.781	1.800	1.853
mozilla	51,220,480	2.967	5.305	2.093	2.770	2.649	2.684	2.641	2.428	2.476
mr	9,970,564	2.948	3.016	2.204	1.932	1.869	1.841	1.806	1.849	1.883
nci	33,553,445	0.712	0.912	0.412	0.427	0.371	0.328	0.316	0.539	0.439
ooffice	6,152,192	4.019	5.471	3.157	3.693	3.473	3.547	3.465	3.216	3.288
osdb	10,085,684	2.948	3.459	2.262	2.189	1.971	1.908	2.002	1.941	1.894
reymont	6,627,202	2.198	2.265	1.591	1.483	1.330	1.292	1.222	1.293	1.279
samba	21,606,400	2.002	3.262	1.395	1.670	1.600	1.579	1.574	1.429	1.389
sao	7,251,944	5.877	7.429	4.889	5.449	5.314	5.431	5.306	4.959	5.281
webster	41,458,703	2.327	2.680	1.618	1.656	1.491	1.429	1.325	1.310	1.293
xml	5,345,280	0.991	1.697	0.680	0.652	0.599	0.616	0.606	0.643	0.569
x-ray	8,474,240	5.700	6.616	4.224	3.801	3.609	3.584	3.606	3.946	3.682
Average		2.976	3.771	2.229	2.325	2.187	2.183	2.138	2.113	2.111
Std. dev.		1.590	2.000	1.320	1.420	1.390	1.430	1.410	1.330	1.390

File	Size [B]	MH	MI4	MI64	lgha	acb	rar	boa	ufa	**DM**	**DW**
dickens	10,192,446	1.797	1.928	**1.704**	2.214	1.931	2.498	1.837	1.986	1.868	1.816
mozilla	51,220,480	2.417	2.237	**2.077**	2.949	2.436	2.602	2.638	2.777	2.600	2.543
mr	9,970,564	1.857	1.835	1.832	1.977	2.007	2.707	1.874	1.879	1.822	**1.770**
nci	33,553,445	0.418	0.574	**0.293**	0.686	0.368	0.534	0.463	0.569	0.299	0.297
ooffice	6,152,192	3.225	2.942	**2.834**	3.750	3.264	3.573	3.421	3.634	3.459	3.380
osdb	10,085,684	1.844	1.848	1.839	3.085	2.075	2.627	1.957	2.115	**1.828**	1.835
reymont	6,627,202	1.140	1.413	**1.117**	1.547	1.313	1.893	1.386	1.471	1.237	1.224
samba	21,606,400	1.342	1.504	**1.262**	1.917	1.453	1.641	1.516	1.697	1.518	1.518
sao	7,251,944	5.186	**4.763**	4.775	5.370	5.199	5.555	5.214	5.332	5.294	5.195
webster	41,458,703	1.257	1.402	**1.159**	1.766	1.441	1.938	1.381	1.488	1.309	1.276
xml	5,345,280	0.556	0.803	**0.538**	1.075	0.577	0.741	0.641	0.857	0.595	0.602
x-ray	8,474,240	3.637	3.578	3.584	4.411	4.043	5.275	3.772	3.919	3.575	**3.518**
Average		2.065	2.069	**1.918**	2.562	2.176	2.632	2.175	2.310	2.117	2.081
Std. dev.		1.370	1.190	1.280	1.400	1.400	1.550	1.370	1.380	1.410	1.380

Table 4.15: Compression ratios (in bpc) of the algorithms for the Silesia corpus

4.4.5 Experiments on the Silesia corpus

In Section 5, we discussed why the Calgary corpus is not a good candidate for a dataset containing typical files that are used nowadays. In that section, we introduced the Silesia corpus consisting of files of large sizes. In the next experiment, the compression methods were compared on this corpus. Tables 4.15 and 4.16 contain compression ratios for files from it. The compression and decompression speeds are shown in Tables 4.17 and 4.18.

Similarly to our discussion of experiments on the Calgary corpus, we may have a better view of the results if we look at Figures 4.19, 4.20, 4.21, and 4.22. First, let us look at figures related to compression (4.19 and 4.20). We can make a similar observation to the one for the Calgary corpus—the PPM algorithms obtain the best compression ratios, but they are in general much slower than other

File	Size [B]	gzip	LZW	LZMA	bzip	szip	bwc	ybs	PPMN	PPMdH
dickens	10,192,446	1.774	1.846	1.304	1.278	1.155	1.146	1.045	1.056	1.087
mozilla	51,220,480	1.429	2.554	1.008	1.334	1.275	1.292	1.272	1.169	1.192
mr	9,970,564	1.666	1.704	1.245	1.092	1.056	1.040	1.020	1.045	1.064
nci	33,553,445	2.430	3.113	1.406	1.457	1.266	1.119	1.078	1.840	1.498
ooffice	6,152,192	1.418	1.930	1.114	1.303	1.225	1.252	1.223	1.135	1.160
osdb	10,085,684	1.613	1.892	1.237	1.197	1.078	1.044	1.095	1.062	1.036
reymont	6,627,202	1.968	2.028	1.424	1.328	1.191	1.157	1.094	1.158	1.145
samba	21,606,400	1.586	2.585	1.105	1.323	1.268	1.251	1.247	1.132	1.101
sao	7,251,944	1.234	1.560	1.026	1.144	1.116	1.140	1.114	1.041	1.109
webster	41,458,703	2.008	2.312	1.396	1.429	1.286	1.233	1.143	1.130	1.116
xml	5,345,280	1.842	3.154	1.264	1.212	1.113	1.145	1.126	1.195	1.058
x-ray	8,474,240	1.620	1.881	1.201	1.080	1.026	1.019	1.025	1.122	1.047
Average		1.716	2.213	1.228	1.265	1.171	1.153	1.124	1.174	1.134
Std. dev.		0.319	0.532	0.143	0.122	0.093	0.090	0.084	0.216	0.124

File	Size [B]	MH	MI4	MI64	lgha	acb	rar	boa	ufa	**DM**	**DW**
dickens	10,192,446	1.055	1.131	**1.000**	1.299	1.133	1.466	1.078	1.165	1.096	1.066
mozilla	51,220,480	1.164	1.077	**1.000**	1.420	1.173	1.253	1.270	1.337	1.252	1.224
mr	9,970,564	1.049	1.037	1.035	1.117	1.134	1.529	1.059	1.062	1.029	**1.000**
nci	33,553,445	1.427	1.959	**1.000**	2.341	1.256	1.823	1.580	1.942	1.020	1.014
ooffice	6,152,192	1.138	1.038	**1.000**	1.323	1.152	1.261	1.207	1.282	1.221	1.193
osdb	10,085,684	1.009	1.011	1.006	1.688	1.135	1.437	1.071	1.157	**1.000**	1.004
reymont	6,627,202	1.110	1.265	**1.000**	1.385	1.175	1.695	1.241	1.317	1.107	1.096
samba	21,606,400	1.063	1.192	**1.000**	1.519	1.151	1.300	1.201	1.345	1.203	1.203
sao	7,251,944	1.089	**1.000**	1.003	1.127	1.092	1.166	1.095	1.119	1.111	1.091
webster	41,458,703	1.085	1.210	**1.000**	1.524	1.243	1.672	1.192	1.284	1.129	1.101
xml	5,345,280	1.033	1.493	**1.000**	1.998	1.072	1.377	1.191	1.593	1.106	1.119
x-ray	8,474,240	1.034	1017	1.019	1.254	1.149	1.499	1.072	1.114	1.016	**1.000**
Average		1.105	1.203	**1.005**	1.500	1.155	1.457	1.188	1.310	1.108	1.093
Std. dev.		0.111	0.278	0.011	0.360	0.053	0.200	0.144	0.246	0.084	0.081

Table 4.16: Normalised compression ratios of the algorithms for the Silesia corpus

methods. For the Silesia corpus, the PPMdH algorithm is also the fastest one from the PPM family, but now the best BWT-based algorithm, the DW, achieves better compression ratio and is Pareto-optimal. If we consider normalised compression ratio, then also the DM algorithm is non-dominated. The LZ algorithms for the Silesia corpus are typically (excluding LZMA method) the fastest ones, but also obtain the poorest compression ratios. If we consider the standard compression ratio, then the DW method obtains only a slightly poorer ratio than the MH and the MI4 algorithms, but if we look at the normalised compression ratio, then the DW algorithm outperforms the mentioned ones. The only algorithm that is significantly better than the DW is the MI64. The MI64 algorithm compresses, however, several times slower than the DW method.

In Figures 4.21 and 4.22, we can see the behaviour of the compression al-

File	Size [B]	gzip	LZW	LZMA	bzip	szip	bwc	ybs	PPMN	PPMdH
dickens	10,192,446	2.96	1.78	149.54	13.02	22.06	16.70	17.08	40.03	13.38
mozilla	51,220,480	28.14	9.53	207.73	52.59	96.13	67.95	68.18	318.65	87.05
mr	9,970,564	4.80	1.57	131.31	8.52	23.54	13.45	13.05	52.63	11.63
nci	33,553,445	7.83	3.98	116.25	63.71	122.17	59.62	55.25	16.10	7.62
ooffice	6,152,192	1.84	1.26	19.03	6.65	10.06	7.97	8.26	47.20	10.62
osdb	10,085,684	1.68	1.99	49.28	11.09	16.75	15.64	16.11	44.87	15.25
reymont	6,627,202	3.60	1.01	35.83	8.43	12.43	10.94	10.21	10.92	5.99
samba	21,606,400	4.20	3.54	89.43	20.95	54.56	27.55	28.44	64.92	18.32
sao	7,251,944	1.93	1.70	23.29	10.61	10.26	12.85	13.16	64.17	25.29
webster	41,458,703	8.91	6.46	267.95	50.27	82.40	78.45	71.63	120.22	40.32
xml	5,345,280	0.67	0.86	13.68	6.17	15.72	6.80	6.67	4.05	1.86
x-ray	8,474,240	1.40	1.76	23.84	8.28	10.13	11.45	11.56	94.85	20.13
Total	211,938,580	67.96	35.44	927.16	260.29	476.21	329.37	319.60	878.61	257.46
Avg. comp. speed		4102	5714	263	855	506	659	695	476	1200
Std. dev.		1846	1074	72	168	153	95	85	592	1174

File	Size [B]	MH	MI4	MI64	lgha	acb	rar	boa	ufa	**DM**	**DW**
dickens	10,192,446	25.07	20.95	59.97	17.95	417.69	9.75	111.80	45.27	11.38	13.32
mozilla	51,220,480	159.54	242.26	415.90	191.90	984.12	36.74	1172.40	270.15	73.45	83.09
mr	9,970,564	18.30	28.52	43.37	17.46	223.39	7.21	158.96	44.42	12.14	13.48
nci	33,553,445	22.39	26.06	329.00	22.49	307.90	11.00	44.30	114.20	66.66	66.43
ooffice	6,152,192	19.15	42.46	59.53	25.87	205.67	5.18	161.13	37.26	8.19	9.57
osdb	10,085,684	30.60	53.62	85.50	39.84	261.24	9.80	177.14	53.39	14.02	15.51
reymont	6,627,202	12.82	10.45	37.25	7.34	205.63	5.56	39.35	26.05	6.40	7.58
samba	21,606,400	41.28	67.43	230.48	46.32	207.45	12.08	207.30	93.45	28.07	30.78
sao	7,251,944	41.46	109.77	118.58	50.78	402.05	7.77	361.81	51.17	15.09	17.54
webster	41,458,703	82.01	68.33	260.41	64.99	1245.21	31.87	338.50	178.18	57.67	61.66
xml	5,345,280	5.58	6.88	63.91	5.83	42.20	2.17	12.98	19.69	7.13	7.66
x-ray	8,474,240	29.58	72.22	77.90	44.87	288.18	8.64	267.76	58.81	12.69	14.64
Total	211,938,580	487.78	748.95	1781.80	535.64	4790.73	147.77	3053.43	992.04	312.89	341.26
Avg. comp. speed		520	422	125	541	52	1444	155	208	717	640
Std. dev.		354	344	47	387	37	637	211	47	147	117

Table 4.17: Compression times (in seconds) of the algorithms for the Silesia corpus

File	Size [B]	gzip	LZW	LZMA	bzip	szip	bwc	ybs	PPMN	PPMdH
dickens	10,192,446	0.15	0.75	0.75	5.76	6.15	4.66	4.72	40.31	13.98
mozilla	51,220,480	0.87	3.63	3.69	20.82	21.24	21.35	24.11	319.43	91.56
mr	9,970,564	0.21	0.68	0.84	4.06	5.56	4.15	4.59	53.09	12.07
nci	33,553,445	0.28	1.83	0.99	14.25	15.28	11.93	11.93	17.30	8.68
ooffice	6,152,192	0.80	0.52	0.62	3.24	3.14	2.51	3.15	47.51	11.29
osdb	10,085,684	0.19	0.77	0.78	5.80	6.21	4.45	4.88	45.19	15.97
reymont	6,627,202	0.10	0.47	0.41	3.34	3.64	2.57	2.71	11.21	6.34
samba	21,606,400	0.29	1.42	1.11	7.34	8.89	7.19	7.45	65.35	19.45
sao	7,251,944	0.20	0.69	0.99	5.60	5.59	4.42	6.01	64.18	26.49
webster	41,458,703	0.65	2.79	2.44	19.72	22.34	17.55	16.76	121.68	42.29
xml	5,345,280	0.08	0.32	0.19	1.99	2.17	1.47	1.55	4.28	2.04
x-ray	8,474,240	0.28	0.71	1.21	5.39	5.65	4.28	5.31	96.47	20.97
Total	211,938,580	4.10	14.58	14.02	97.31	105.86	86.53	93.17	886.00	271.13
Avg. decomp. speed		56381	13749	15557	2056	1872	2419	2226	456	1106
Std. dev.		26796	2099	7847	474	375	497	583	550	1027

File	Size [B]	MH	MI4	MI64	lgha	acb	rar	boa	ufa	**DM**	**DW**
dickens	10,192,446	25.80	22.55	61.20	18.68	417.77	0.38	112.48	50.34	6.43	8.65
mozilla	51,220,480	162.14	273.81	445.49	215.70	982.64	1.80	1193.79	297.32	39.78	52.53
mr	9,970,564	18.82	32.14	46.66	18.50	219.30	0.36	160.79	49.46	6.41	8.17
nci	33,553,445	24.08	28.70	330.35	23.82	305.34	0.65	45.12	130.95	15.74	20.79
ooffice	6,152,192	19.52	46.59	63.76	27.46	203.68	0.29	163.90	40.49	5.16	6.86
osdb	10,085,684	31.31	59.00	90.56	41.52	259.71	0.36	179.82	58.63	7.00	9.30
reymont	6,627,202	13.27	11.51	38.27	7.66	204.62	0.20	39.78	29.33	3.54	4.88
samba	21,606,400	42.21	75.39	238.06	48.85	311.51	0.58	211.45	104.70	12.07	16.44
sao	7,251,944	41.42	125.61	134.28	54.53	401.03	0.43	369.13	55.29	9.72	13.06
webster	41,458,703	84.77	73.45	273.32	67.75	1240.99	1.33	342.61	198.50	25.58	35.17
xml	5,345,280	5.82	7.43	64.46	6.00	42.03	0.14	13.20	22.37	1.80	2.76
x-ray	8,474,240	29.90	82.23	87.00	48.51	287.79	0.44	272.90	63.14	8.59	11.04
Total	211,938,580	499.06	838.41	1873.41	578.98	4876.41	6.96	3104.97	1100.52	141.82	189.65
Avg. decomp. speed		501	385	118	514	50	29272	153	187	1561	1146
Std. dev.		327	316	45	370	34	9241	207	39	564	525

Table 4.18: Decompression times (in seconds) of the algorithms for the Silesia corpus

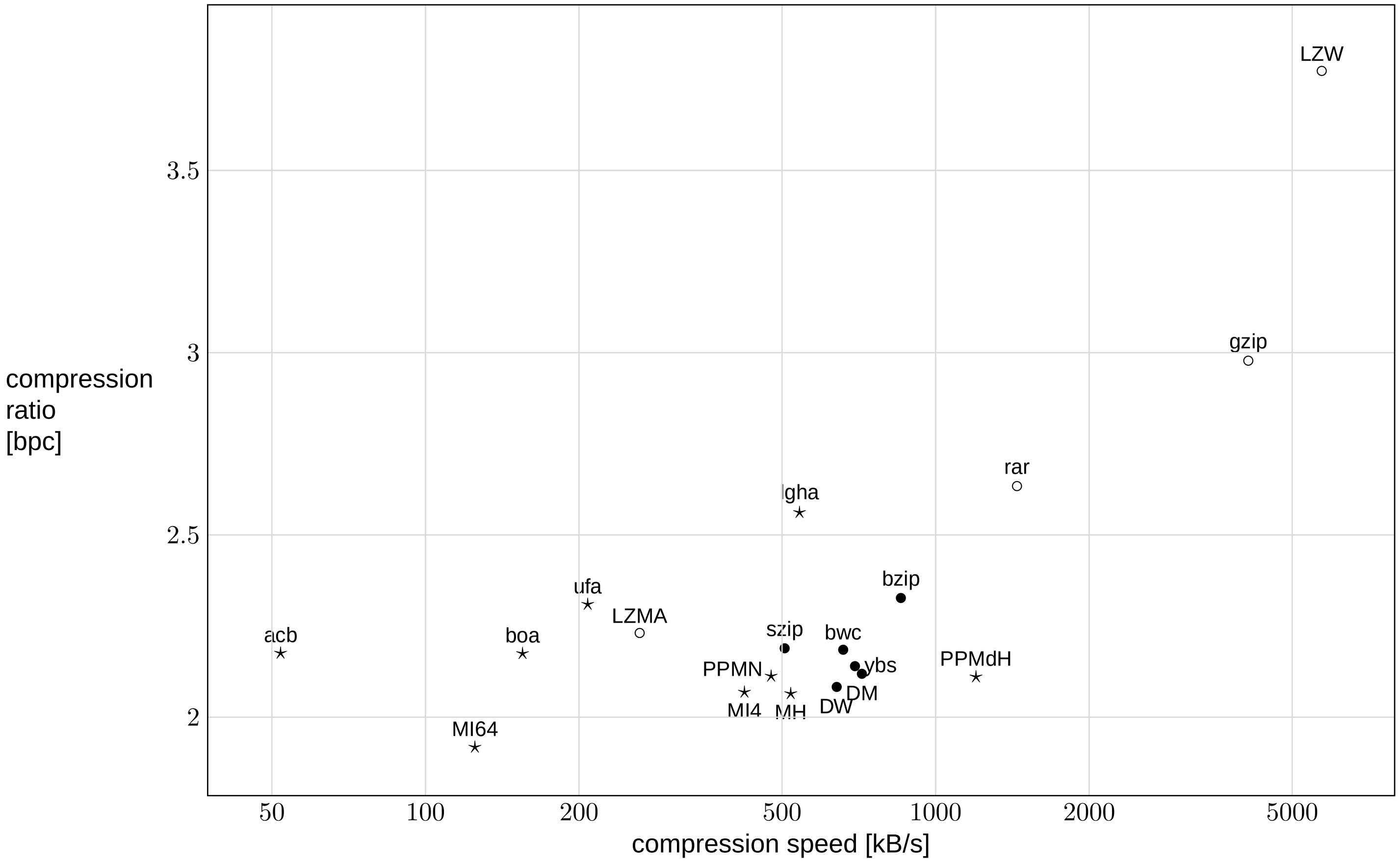

Figure 4.19: Compression ratio versus compression speed of the examined algorithms for the Silesia corpus. The LZ methods are denoted by ○, the PPM methods are denoted by ⋆, and the BWT-based methods by •.

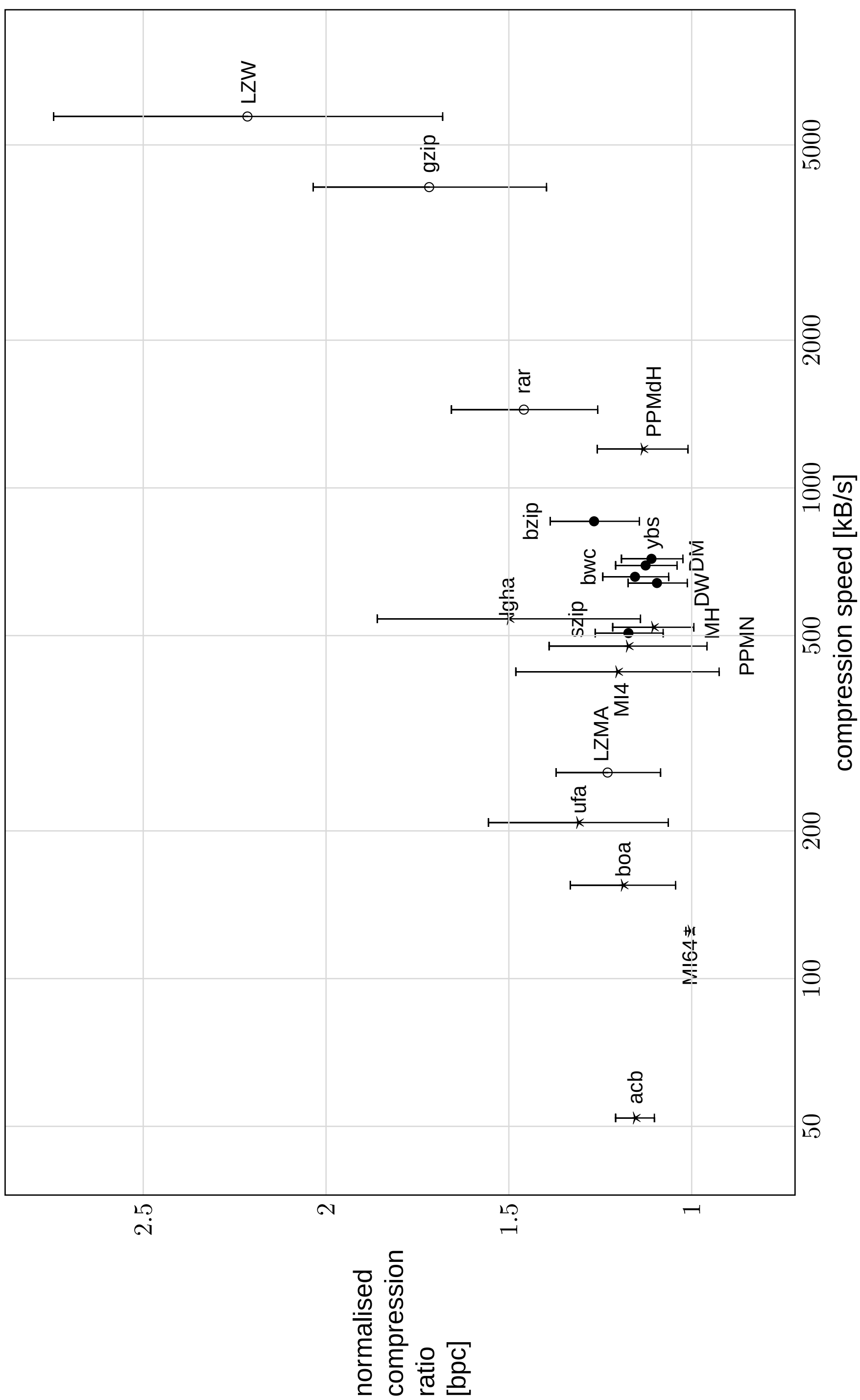

Figure 4.20: Normalised compression ratio versus compression speed of the examined algorithms for the Silesia corpus. The LZ methods are denoted by ◦, the PPM methods are denoted by ⋆, and the BWT-based methods by •.

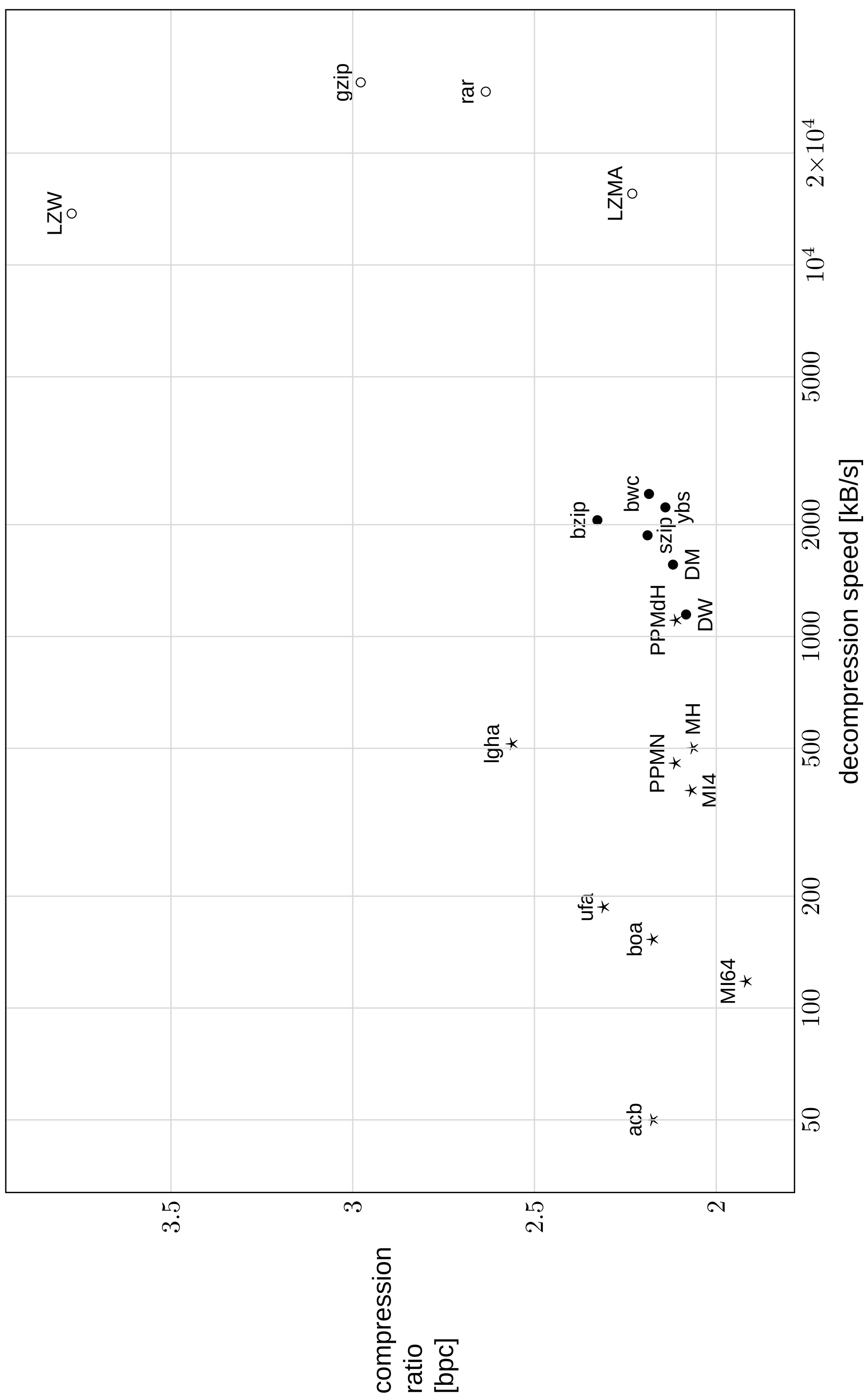

Figure 4.21: Compression ratio versus decompression speed of the examined algorithms for the Silesia corpus. The LZ methods are denoted by $\circ$, the PPM methods are denoted by $\star$, and the BWT-based methods by $\bullet$.

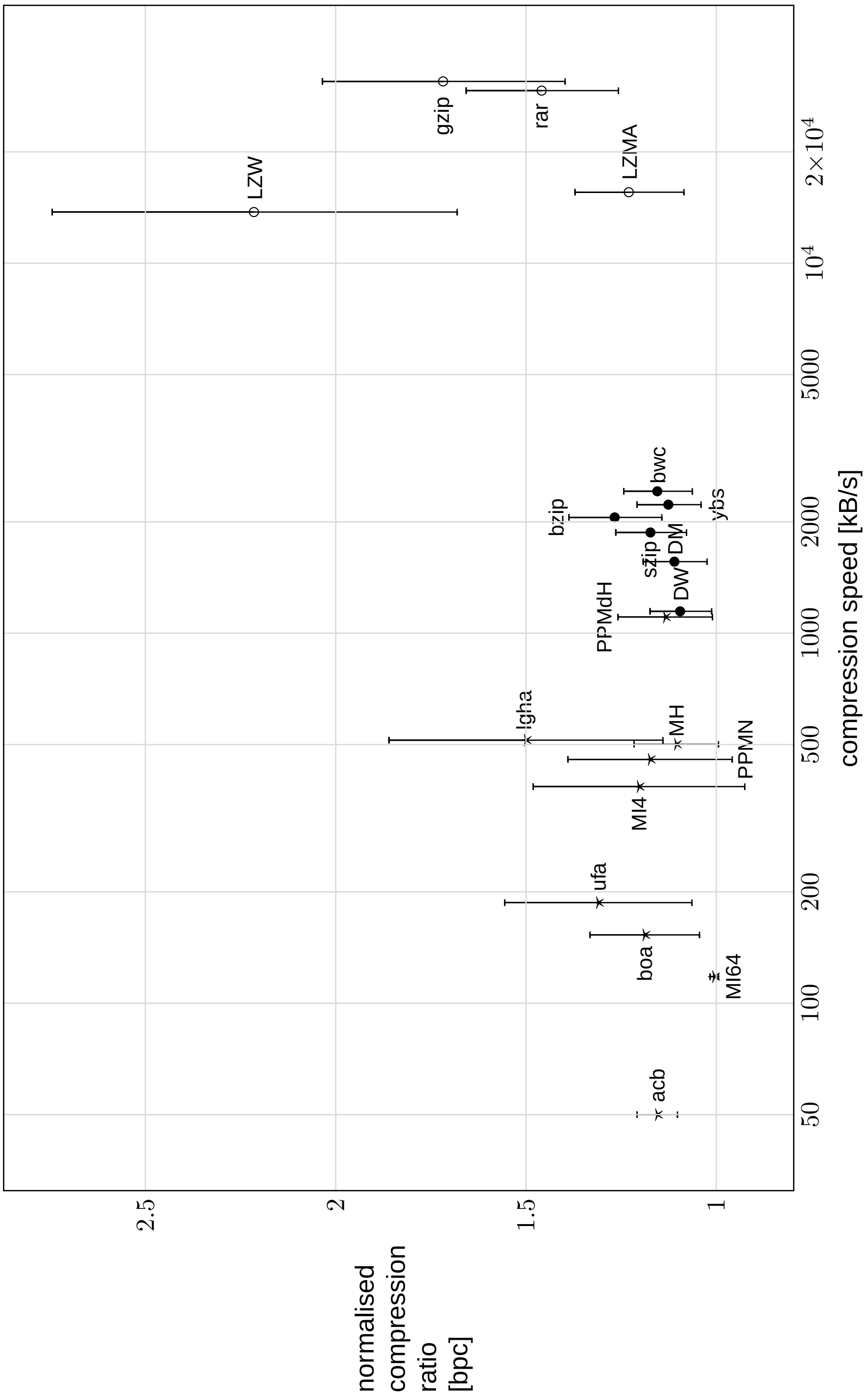

Figure 4.22: Normalised compression ratio versus decompression speed of the examined algorithms for the Silesia corpus. The LZ methods are denoted by $\circ$, the PPM methods are denoted by $\star$, and the BWT-based methods by $\bullet$.

gorithms in the decompress process for the Silesia corpus. The observation is also similar to that made for the Calgary corpus. The decompression speed of the LZ methods is significantly higher than their compression speed. The PPM algorithms compress and decompress with almost identical speed. All the examined BWT-based methods decompress faster than PPM algorithms, but the difference between the PPMdH and DW is small.

The standard deviation of the compression and decompression speed is significantly lower for the BWT-based algorithms than for the PPM methods. Also the standard deviation of the normalised compression ratio for the majority of the BWT-based algorithms is smaller than for the PPM methods (except for MI64 and acb).

Until now, we have analysed the average compression ratio. Let us now take a look at the specific files from the Silesia corpus. The dominance, in terms of the compression ratio, of the MI64 algorithm is almost total for the Calgary corpus. In the experiments for the Silesia corpus, we see that this method achieves the best compression ratios only for 8 out of 12 files. For one file, sao, the best compression ratio obtains the MI4 algorithm, while for the other three, the best are the methods introduced in this dissertation—DW and DM. For two medical files, mr and x-ray, the DW method achieves significantly better compression ratio than the MI64 algorithm. For the last file, osdb, the DW algorithm also outperforms the MI64 method, but the DM algorithm is even better.

4.4.6 Experiments on files of different sizes and similar contents

In the last experiment, we compare the compression algorithms on files of similar contents and different sizes. For this purpose, four files from the Silesia corpus were chosen. Each file is of different type: dickens is a plain text file, osdb is a binary database file, samba consists of concatenated source code files of a programming project, and x-ray is a representation of a medical image.

From each file the pieces of sizes 8 kB, 16 kB, 32 kB, 64 kB, ..., 8192 kB, and in one case also 16384 kB, were taken starting always from the beginning of the file. Several compression methods were run on the obtained files. Tables E.1–E.8 contain the results of these experiments. The results are also plotted in Figures 4.23–4.30 to show how the properties of the compression methods depend on the file size.

Figures 4.23 and 4.24 illustrate the experiments with parts of the dickens file. The compression ratio decreases almost monotonically while the file size grows. It is what one would expect, as the contents of the dickens file is an English text and on larger files the probability of symbol occurrence can be estimated better. The only compression algorithm, for which there is no progress in the compression ratio for large files is bzip. It happens because if the size of the file to be compressed is bigger than 900 kB, the *bzip* program splits the file into pieces of size 900 kB, and compresses them separately. More interesting observations can be made for this file if we look at the dependence of the compression speed on the file size. For small files, the compression speed grows for all the algorithms if the file size grows. If the file to be compressed breaks the limit of several tens–several hundred kB, the compression speed decreases. For the PPM algorithms it is caused by the need of storing and searching a model of vast size. For the BWT-based algorithms, the decrease of speed is caused by the sorting procedure in the BWT, which is super-linear. The speed of the bzip method is steady for large files, as this program compresses separately pieces of size of 900 kB. The speed-improved BWT computation algorithm used in the DW method allows the compressor decrease the speed of compression slowly.

The results of the experiments for a binary database file, osdb, are presented in Figures 4.25 and 4.26. Looking at the first figure, we see also a monotonic improvement of the compression ratio for all the compression methods with growing file size. We notice also how the compression ratio of the BWT-based methods, ybs and DW, improves, closing to the one obtained by the MI64 algorithm. For the largest file, the compression ratios of the DW and the MI64 algorithms are almost equal. The situation shown in Figure 4.26 is similar to the one discussed for the dickens file. Here the decrease of speed for the DW algorithm is also slow.

The samba file is a tarred source code of a programming project. The project contains files which are: C and Perl sources, documentation in HTML, PDF and text files, text configuration files, and others. What is important, the files are rather small and of very different contents. Therefore, the contents of the samba file is changing from part to part of this file. Figure 4.27 shows how hard are different parts of the file to compress. All the compression algorithms behave similarly and as we can see, the PPM, and the LZ algorithms achieve the best compression ratios. Analysing Figure 4.28 we notice how the compression speed varies. The general tendency of decreasing the speed with file size growing cannot be observed. We can, however, notice that the speeds of the bzip, PPMdH, and DW algorithms are similar for files of different sizes (larger than 50 kB).

The last file in this experiment, x-ray, stores an X-ray image. The compression ratios, shown in Figure 4.29, decrease with growing file size, however for the largest files, the ratio slightly deteriorates. For pieces of small size the best results are obtained by the MI64 and the LZMA algorithms. With the growth of the file size, the compression ratio improves for them much slower than for the BWT-based algorithms. When the file size exceeds 100 kB, the DW algorithm yields the best compression ratio. The DW method outperforms significantly, with regard to the compression ratio, other methods for this sequence. We should notice also a very good result of the BWT-based compression method, ybs, which obtains ratios almost identical to the best PPM algorithm. Similar

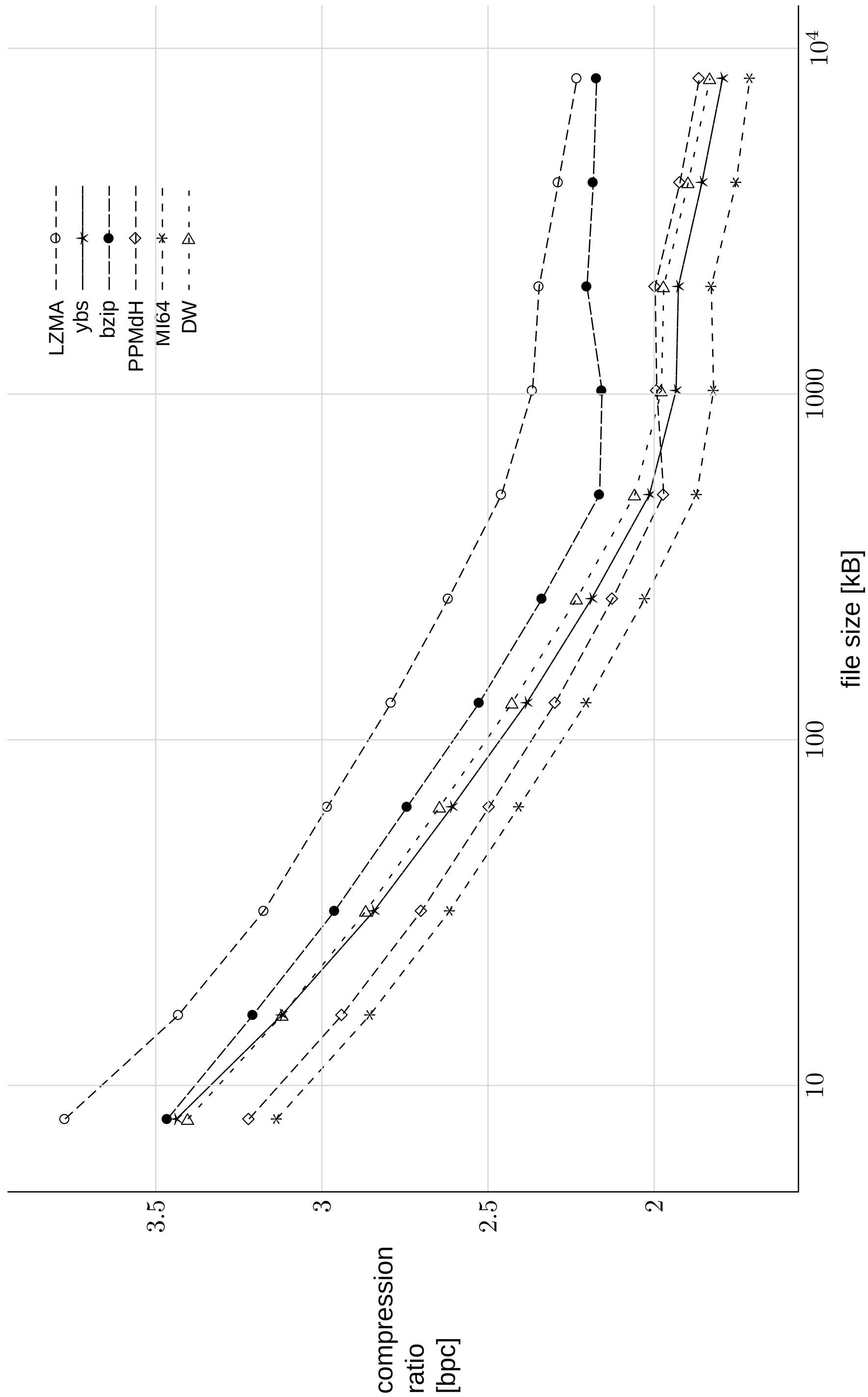

Figure 4.23: Compression ratios for parts of the dickens file of different sizes

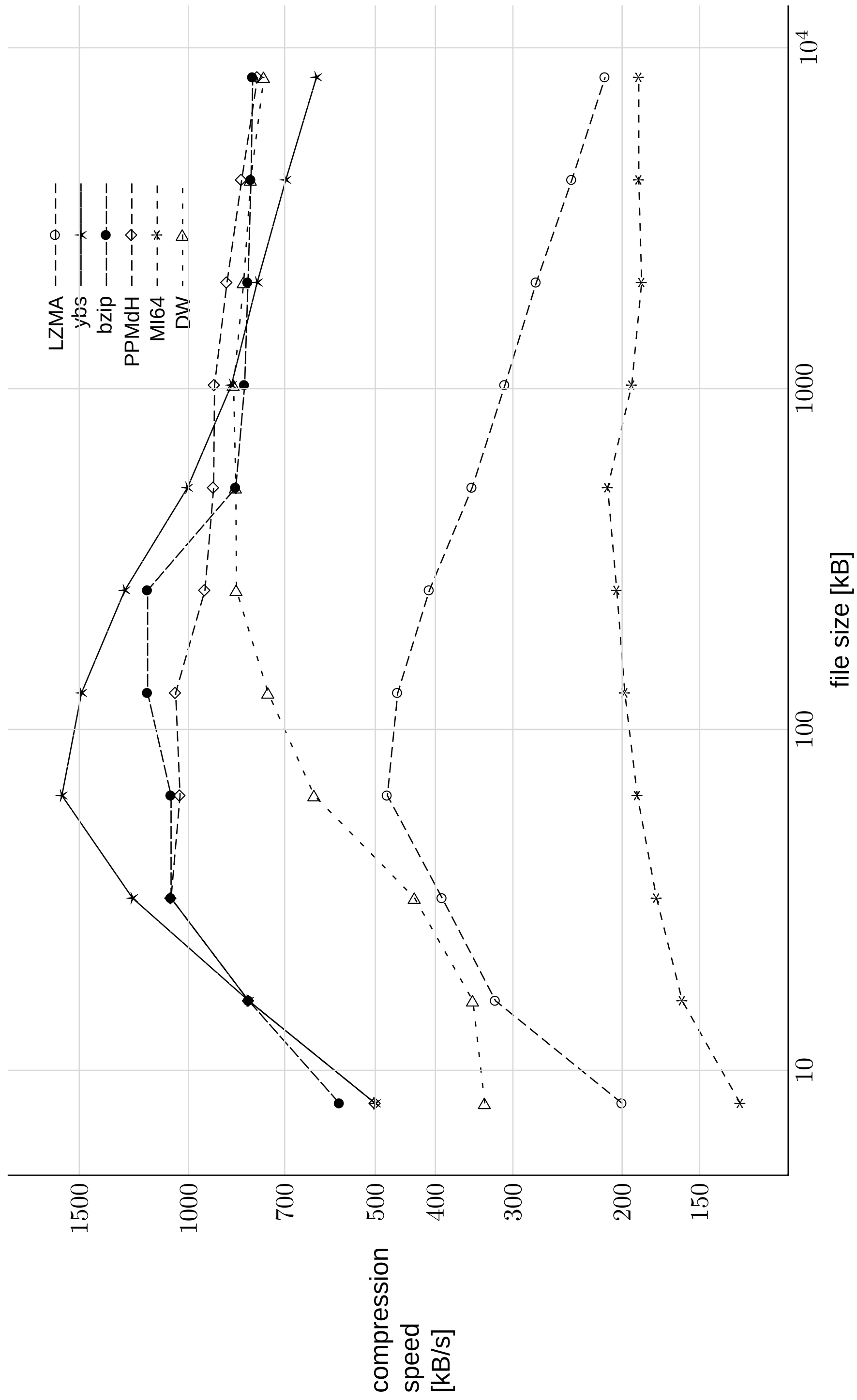

Figure 4.24: Compression speeds for parts of the dickens file of different sizes

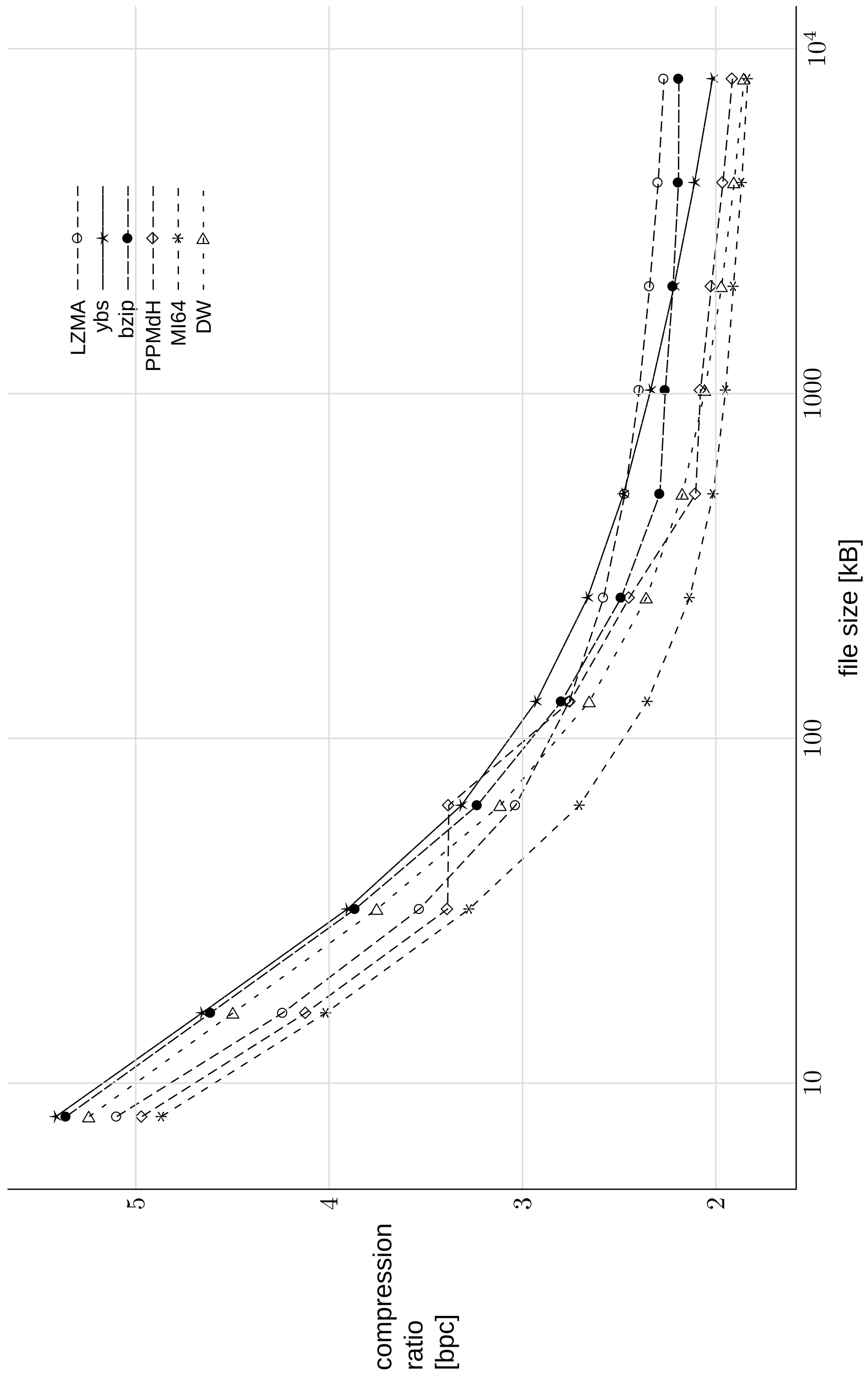

Figure 4.25: Compression ratios for parts of the osdb file of different sizes

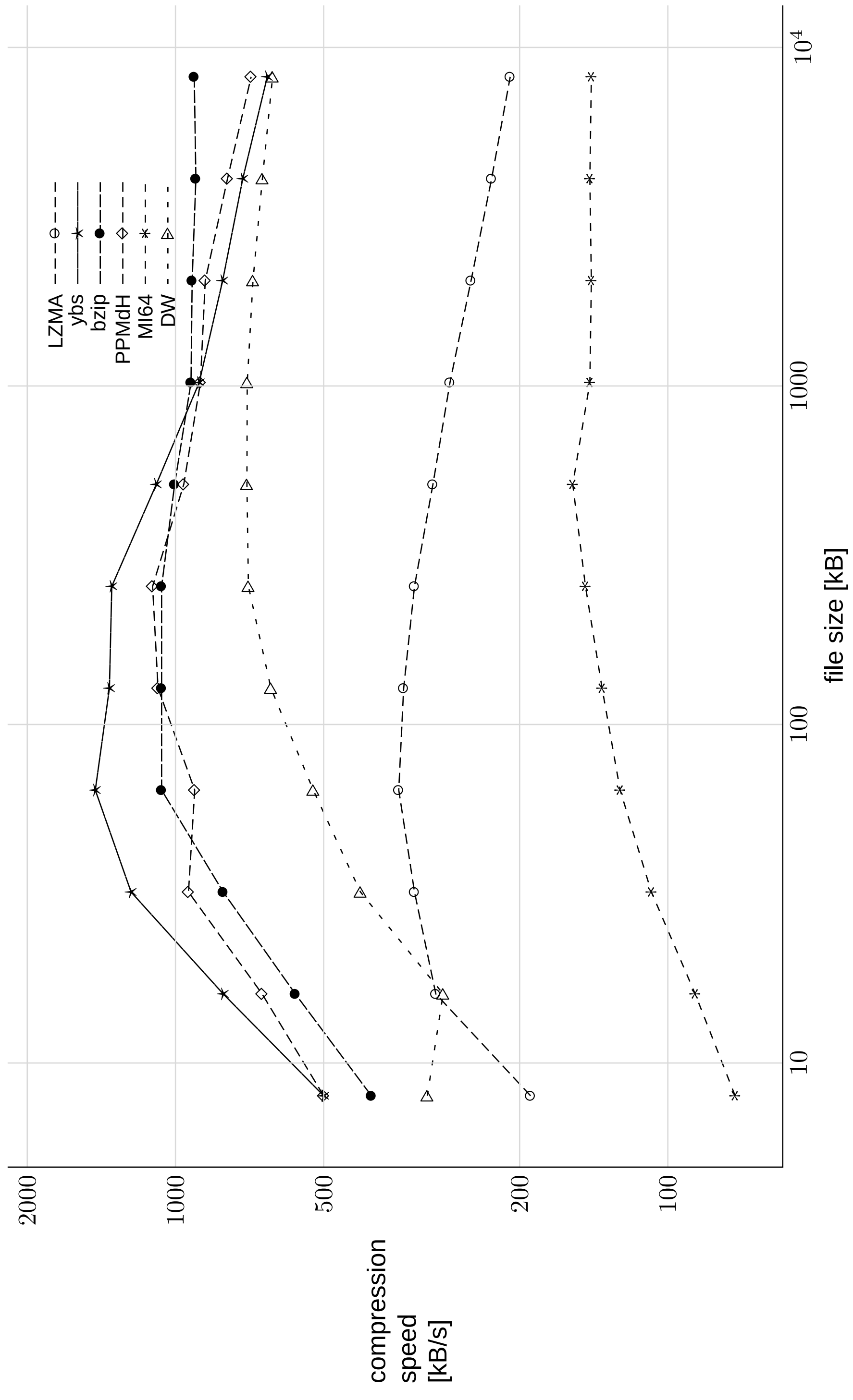

Figure 4.26: Compression speeds for parts of the osdb file of different sizes

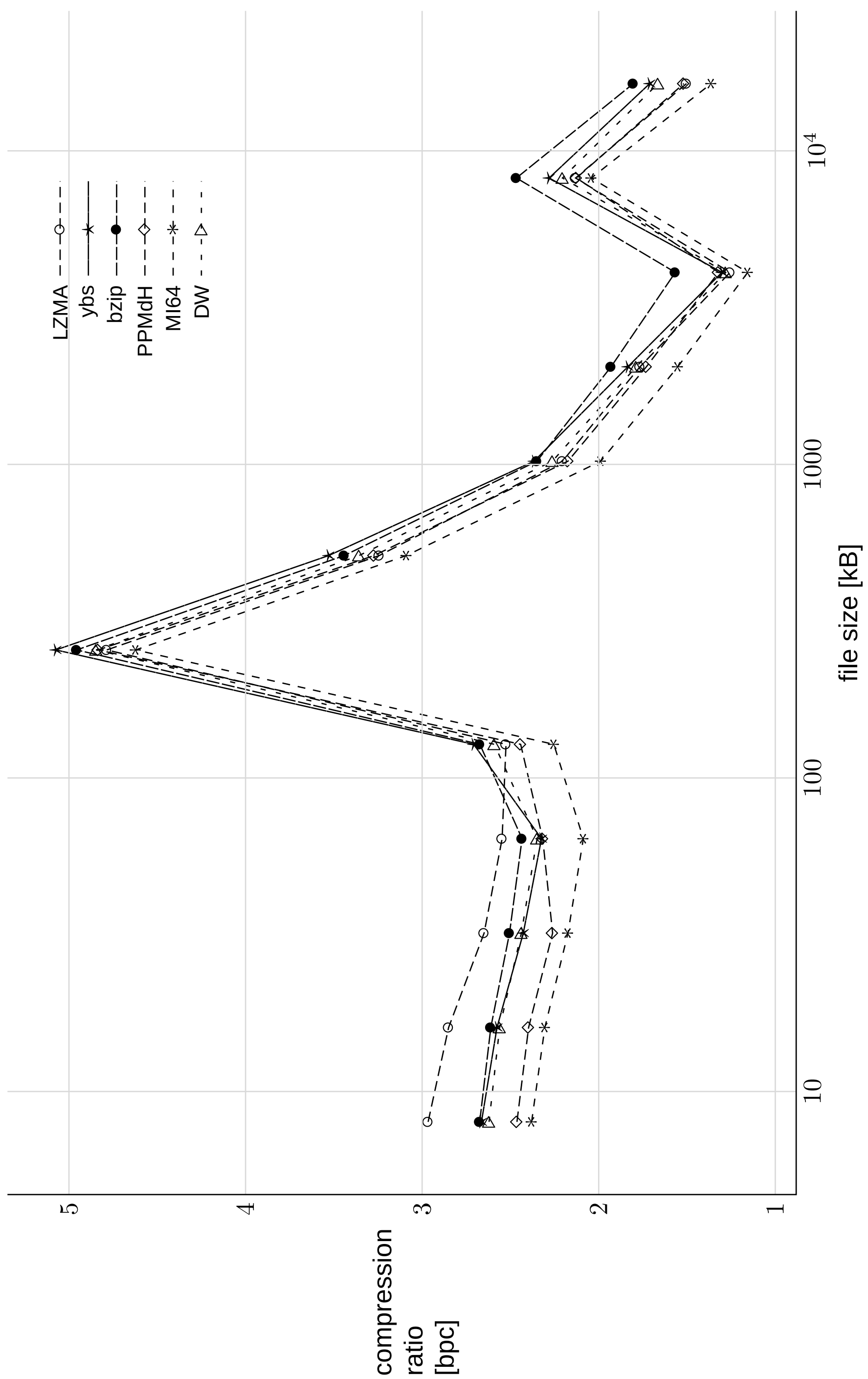

Figure 4.27: Compression ratios for parts of the samba file of different sizes

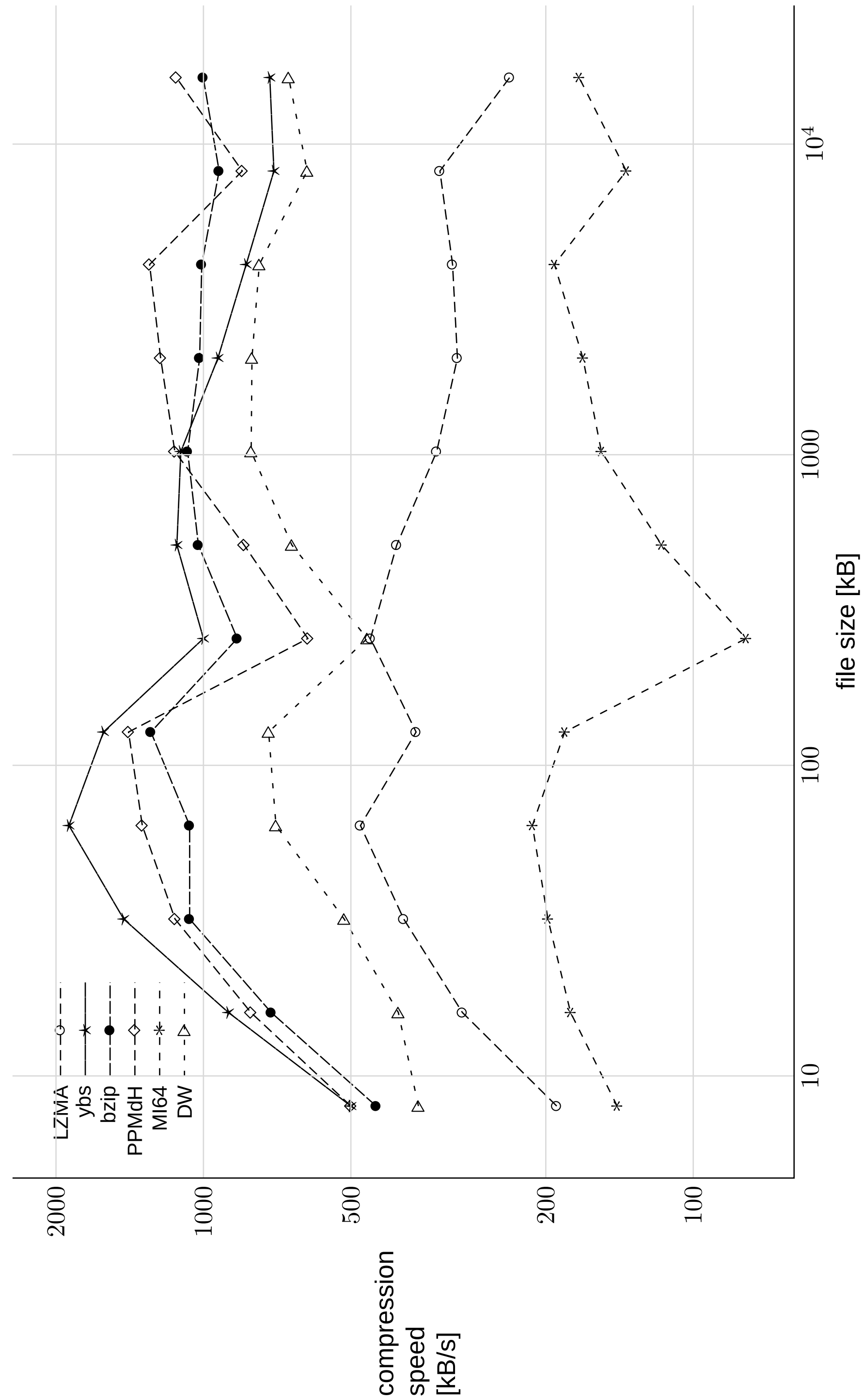

Figure 4.28: Compression speeds for parts of the samba file of different sizes

results for file sizes from 128 kB to 512 kB gives the bzip method. The improvement of the compression ratio for it is stopped by the maximal size of a piece which can be compressed by this algorithm. The results of the compression speed for this file (Figure 4.30) are similar to the ones for the other discussed files. The ybs and bzip are the fastest algorithms, but the speed of the DW method deteriorates only slightly and is also high.

4.4.7 Summary of comparison results

The results show that for both corpora the best compression ratios are obtained by Shkarin's cPPMII compression algorithm [155, 156]. The advantage in terms of the compression ratio of this algorithm over other compression methods is significant. Its main disadvantages are low speed of running and high memory consumption.

From the LZ algorithms the most interesting one is the LZMA. The compression ratios obtained by this method are significantly better than those achieved by other algorithms from this family. Unfortunately, the compression speed is low, comparable to the PPM methods. This disadvantage is partially compensated by fast decompression. In the situations, in which the compression will be made rarely, the LZMA is an interesting candidate to employ.

The PPM methods significantly outperform other compression algorithms in terms of the compression ratio for the files from the Calgary corpus. The main disadvantage of all the PPM algorithms, as also of the DMC and the CTW (which unfortunately we could not examine because of the unavailability of their implementations), is their low speed of compression and decompression.

The BWT-based algorithms yield worse compression ratios for small files (from the Calgary corpus) than the PPM algorithms. For large files, however, the distance between these two families of algorithms becomes smaller. The compression speed of the BWT-based algorithms is also higher. In these methods, the decompression speed is about two or three times higher than the compression speed, what can be important in some cases.

Analysing the experimental results for different algorithms from the point of view of multi criteria optimisation, we can roughly split the methods into three groups. The first group contains the PPM algorithms (also the CTW and the DMC ones), which achieve the best compression ratios, but work slow. The second group contains the LZ algorithms, which work fast, but provide poor compression ratios. The third group contains the BWT-based algorithms, which run faster than the PPM algorithms, and offer the compression ratios much better than those obtained with the LZ methods. Many of these algorithms, ybs, szip, bzip, DW, DM, are non-dominated by other algorithms in some experiments.

The algorithm that leads to the best compression ratios among the BWT-based family is the improved algorithm introduced in this dissertation—DW.

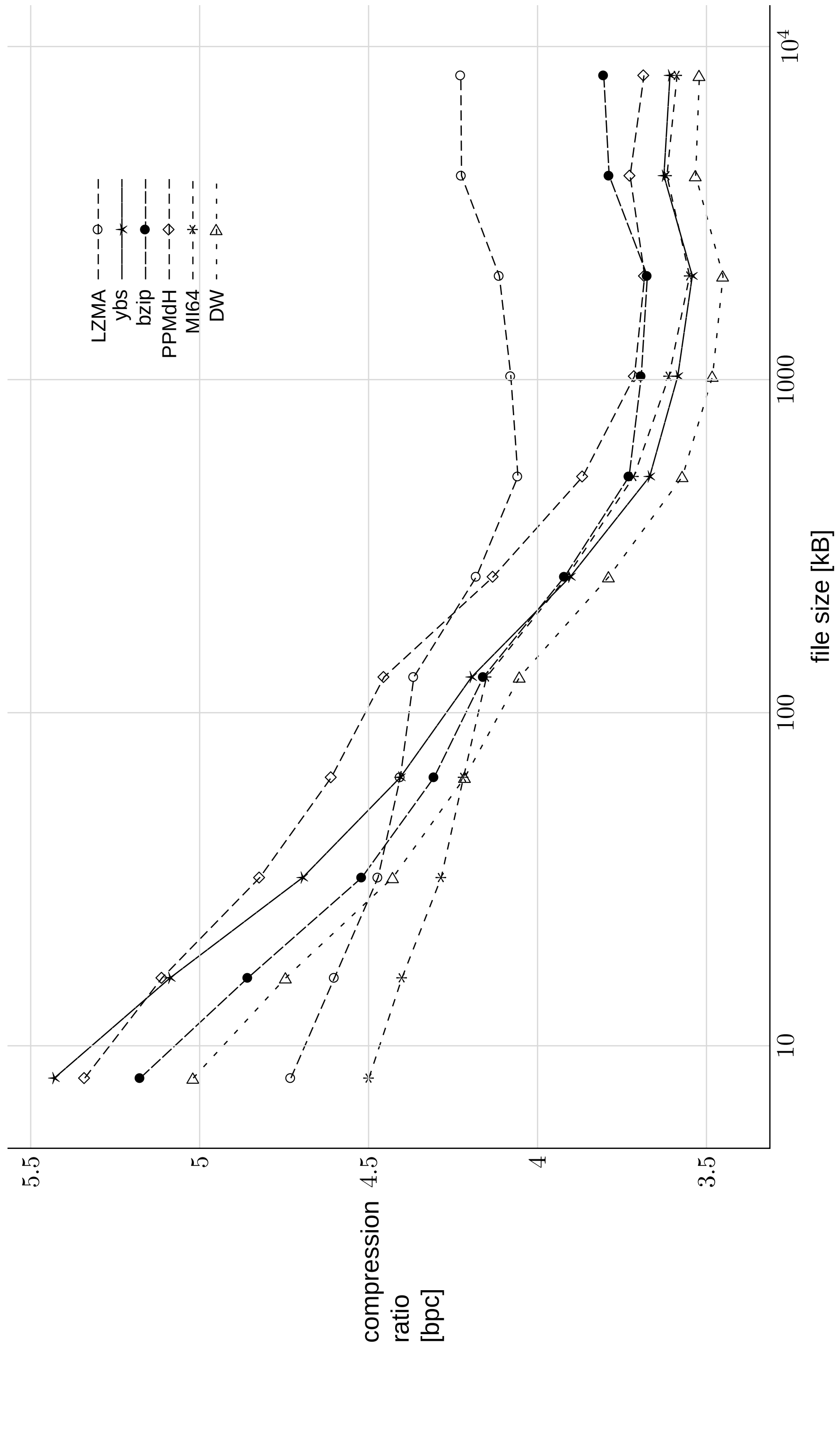

Figure 4.29: Compression ratios for parts of the x-ray file of different sizes

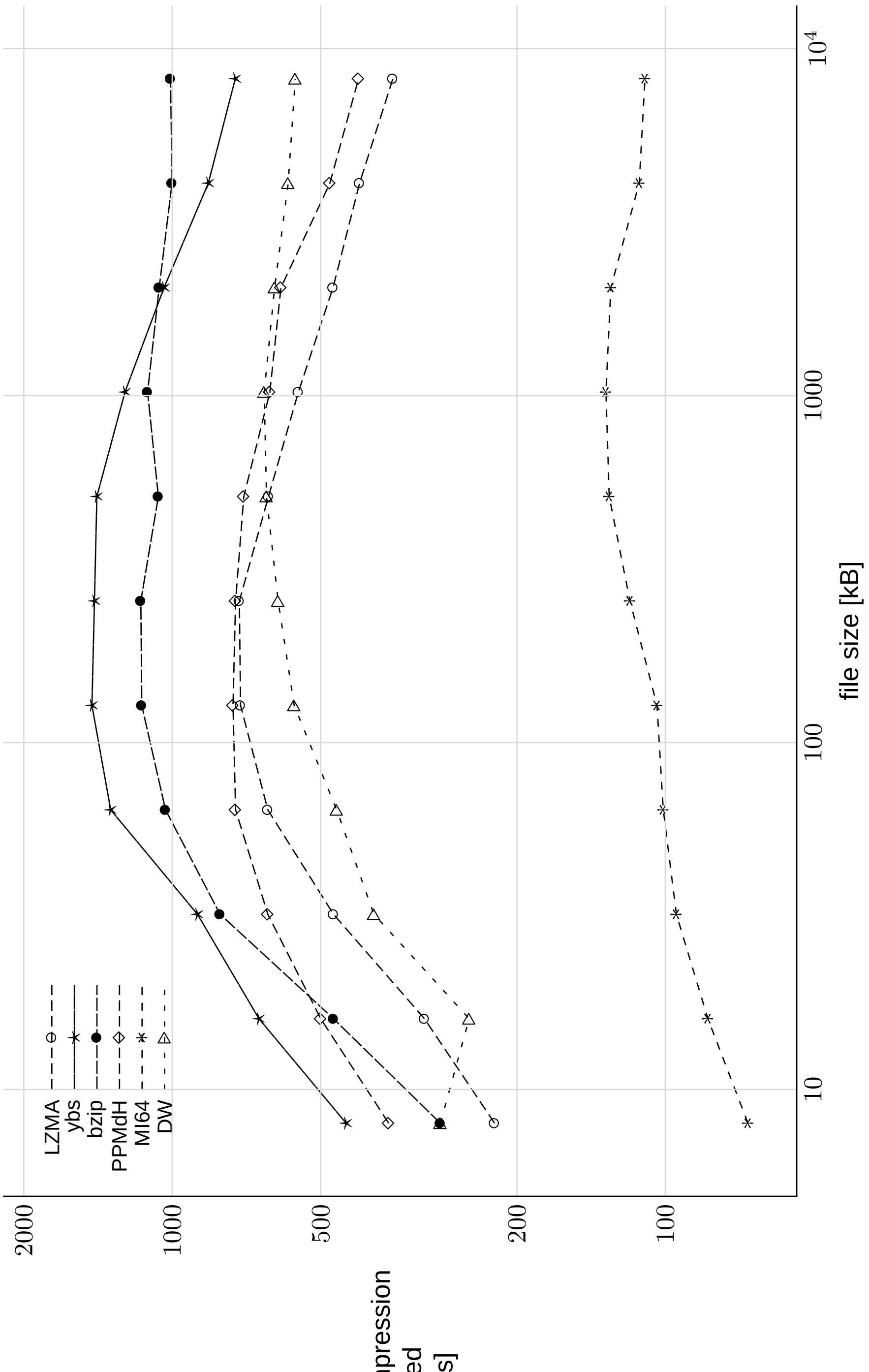

Figure 4.30: Compression speeds for parts of the x-ray file of different sizes

The speed of the DW method is comparable to the other BWT-based algorithms. The tests on files of different sizes show, however, that the speed of compression and decompression is much more steady for the DW method than for other BWT-based algorithm which use blocks of large sizes.

Chapter 5

Conclusions

What will be the conclusion of all this?

— Robert Bolton
Instructions for a Right Comforting
Afflicted Consciences (1635)

The subject of the dissertation are universal lossless data compression algorithms. The background of this field of research was presented in Chapter 2. There we talked about a need of compression, and situations in which universal lossless methods are useful. The algorithms used contemporarily were described in detail. A special, in-depth description of the Burrows–Wheeler transform-based algorithms, was provided (Chapter 3) as the thesis of the dissertation concerns this family of compression methods.

The stages of the BWCA [39] were examined in detail. The first stage is the transform, introduced by Burrows and Wheeler. The transform is well established and the research concentrates on the efficient methods for its computation. Several approaches to this task were discussed. We proposed a significant improvement to the Itoh–Tanaka's method [90], and some small improvements to the Seward's method [149] which combined together offer a highly efficient BWT computation method (Section 4.1.2). A practical performance of the proposed solution was compared, in Section 4.3.1, with other approaches, for which the implementation, or detailed description that allows us to implement it efficiently, was available. The results showed that the combined method of the improved Itoh–Tanaka's methods of orders 1 and 2 is significantly faster than the other examined approaches. Its memory requirement is $8n$, where n is the length of the input sequence. The main disadvantage of this approach is its unknown average-case time complexity and poor worst-case complexity, $O(n^2 \log n)$. The poor time complexity is caused by the sorting procedure used—the Seward's method. It is possible to use different sorting procedures, but in practice they are less efficient. Since the worst-case time complexity of the improved Itoh–Tanaka's method is its significant disadvantage, we proposed, similarly to Seward [150], a fallback procedure, which is used if the execution of the main method takes too long. To this end, the maximal time of execution of the method of time complexity $O(n \log n)$ is estimated, and if the improved Itoh–Tanaka's method exceeds this limit it is stopped to run the fallback procedure. The fallback can be for example the Manber–Myers's method [106], as Seward proposed [150], of the worst-case time complexity $O(n(\log n)^2)$, or the Larsson–Sadakane's method [102] of the worst-case time complexity $O(n \log n)$. The combined solution guarantees the worst-case time complexity $O(n \log n)$ and offers a fast work due to the proposed improved method, which is highly efficient for typical sequences.

The research on the second stage transform was started from the analysis of the structure of the Burrows–Wheeler transform output. In Section 4.1.3, we proved some results regarding the structure of this sequence, concluding that it can be approximated with high precision as an output of a piecewise stationary memoryless source. Then, the methods of probability estimation of symbol occurrence in such sequences were investigated (Section 4.1.4). We postulated to weight the symbol importance in the estimation process. We also derived the bounds on the expected redundancy introduced by this method. The equations obtained are very hard to solve, so we decided to analyse them numerically for some weight functions. As the result of these investigations, a transform, weighted frequency count (WFC), as the second stage of the BWCA was introduced. We showed that the WFC is a generalisation of the well known methods: move-to-front, frequency count, and sort-by-time. Some of these methods were formerly used in the versions of the BWCA. We also discussed the time complexity of the WFC, showing a way in which it can be implemented with the worst-case time complexity $O(nkl)$, where k is the alphabet size, and l is a small number, lower than 32 on typical computers. The experiments performed in Sections 4.3.2 and 4.3.3 showed that the WFC transform leads to the best compression ratios among the second stage methods.

The last stage of the BWCA is an entropy coding. The arithmetic coding is an optimal solution for this task, in terms of the compression ratio. The most important part of this stage is a probability estimation of symbol occurrence, which is then used to assign codes to symbols from the alphabet. For this task we introduced a weighted probability estimation (Section 4.1.6). The experimental results (Section 4.3.4) confirmed the validity of usage of this method. Because the sequences are only assumed to be an output of the context tree source of unknown parameters and structure, and the understanding of the BWCA working is incomplete, we were unable to provide a thorough justification for the usage of this method of probability estimation in the BWCA.

The improved algorithm was compared to the state of the art methods published in the literature. The first task in this comparison was to choose a set of test data. There are three widely used standard corpora: the Calgary corpus, the Canterbury corpus, and the large Canterbury corpus. The first corpus is the most popular and many compression methods were examined on it. As we discussed in Section 4.2.1, the two Canterbury corpora are not good candidates to be contemporarily the standard data sets. The main disadvantage of the three corpora is their lack of files of sizes and contents that are used nowadays. Therefore, to provide a comparison of the compression methods on modern files, we introduced the Silesia corpus.

As we discussed in Section 4.2.2, compression methods should be compared in terms of multi criteria optimisation, as three criteria are important: compression ratio, compression speed, and decompression speed. Therefore we considered two processes separately: compression, in which the compression ratio and the compression speed are important, and decompression, in which the compression ratio and the decompression speed matter.

The comparison showed that the proposed algorithm, denoted by DW in the tables and figures, yields the best compression ratios for the both corpora in the family of the BWT-based algorithms. It gives also the best compression ratios for most component files of the corpora. Its compression speed is comparable to other such algorithms. For the Silesia corpus, the DW algorithm compresses about 26% faster than the slowest examined BWT-based method, szip, and about 34% slower than the fastest such an algorithm, bzip. Typically, the compression speeds of the BWT-based algorithms differ by about 10%. The decompression speed of the DW algorithm is about two times higher than its compression speed. In the decompression, the fastest BWT-based algorithm, bwc, is however about two times faster than the DW method. From figures shown in Chapter 4 we can determine also the set of the BWT-based algorithms, which are non-dominated by other methods from this family. One of the non-dominated algorithms is the DW method.

We proposed also a variant of this method, the DM algorithm, which obtains the second best compression ratios from the examined BWT-based algorithms for the Silesia corpus. Its compression speed is lower only from the bzip algorithm. In the decompression, the DM method is about 35% slower than the fastest, bwc, algorithm. The DM algorithm is also non-dominated for both the compression and decompression processes for the both corpora.

The above-described observations of the behaviour of the improved algorithm confirm the thesis of this dissertation.

A comparison of the DW algorithm to the other universal lossless data compression algorithms showed that some PPM algorithms lead to better compression ratios. They are typically much slower than the BWT-based algorithms, but one of the recent methods by Shkarin [155], PPMdH, dominates the DW in the compression of the Calgary corpus. In the decompression, a variant of the DW algorithm, DM, is non-dominated, though. We should also notice that the standard deviation of the speeds of the Shkarin's algorithm is over 3 times larger than the standard deviation of the speeds of the proposed method. The advantage of the PPMdH algorithm comes from the compression of small files. As we can see in the experiments described in Section 4.4.6, the BWT-based algorithms work relatively slow for small files, because of the properties of the BWT computation methods, and they speed up when the file size grows. The experiments on the Silesia corpus showed that the proposed algorithm is non-dominated in both processes, and significantly outperforms the best PPM algorithms in the compression and decompression speed. It also yields the best compression ratios for 3 out of 12 files from the Silesia corpus.

The results in this work can surely be a point of departure for further research. The proposed method for the BWT computation is highly efficient in practice, but the analysis of its average-case time complexity is missed. The experiments suggest that it is low, but calculating it is a difficult, open problem. The theoretical research on the probability estimation for the output of the piecewise stationary memoryless source were at some point discontinued, because of the high complexity of the equations. It could be interesting to provide some more in-depth theoretical analysis. There is probably also a possibility to propose weight functions that yield better compression ratios. Finally, it would be nice if more convincing theoretical grounds for the weighted probability have been found.

Acknowledgements

If we meet someone who owes us thanks,
we right away remember that.
But how often do we meet someone to whom
we owe thanks without remembering that?

— JOHANN WOLFGANG VON GOETHE
Ottilie's Diary (1809)

I cannot imagine that this dissertation could be completed without help from many people. Here, I want to thank all of them.

First, I would like to thank my supervisor, Zbigniew J. Czech, who encouraged me to work in the field of theoretical computer science, and motivated me all the time during the last four years. I am most grateful for his support, stimulating discussions, and constructive remarks on my work. I also thank him for a vast number of improvements, that he suggested to incorporate into the dissertation.

Marcin Ciura, Roman Starosolski, and Szymon Grabowski proofread the whole dissertation or its major parts. Thank them all for their patience in reading the early versions of the dissertation and for suggesting many ways in which the text can be improved. Thank you Marcin for your enormous persistence in showing me how I can be more clear and precise in writing. Thank you also for collaboration on a paper we wrote together. I would like to thank you, Roman, for discussions on the compression algorithms. As Szymon shares my interests in universal lossless data compression, and especially Burrows–Wheeler transform-based compression, he was a very good discussion partner. Thank you Szymon for our stimulating discussions, for your relevant remarks, and for pointing me the papers that I did not know.

Some parts of the compression program accompanying the dissertation are based on existing source codes. An implementation of one of the BWT computation methods is based on the routines by N. Jesper Larsson and Kunihiko Sadakane. An implementation of some other methods for computing the BWT is based on source code by Julian Seward. The arithmetic coding routines, used by me, are modified source codes by John Carpinelli, Alistair Moffat, Radford Neal, Wayne Salamonsen, Lang Stuiver, Andrew Turpin, and Ian Witten. Thank you all for publishing the source codes. The contents of the Silesia corpus introduced in this dissertation was changing, and I want to thank Szymon Grabowski, Uwe Herklotz, Sami J. Mäkinen, Maxim Smirnov, and Vadim Yoockin, who shared with me their opinion and suggested improvements. I would like also to thank Ewa Piętka for her suggestion of including medical data in the experiments, and for providing me such files. I am grateful to my friend, Adam Skórczyński for the possibility of developing together the LaTeX Editor, which was used to write the whole dissertation.

During my Ph.D. study I was employed in the Software Department of the Silesian University of Technology. I am thankful for the possibility to work in this team. Many thanks go to the manager Przemysław Szmal. I am not able to enumerate all the staff, so I would like to give special thanks to my closest friends and colleagues: Marcin Ciura, Agnieszka Debudaj–Grabysz, Aleksandra Łazarczyk, Adam Skórczyński, Roman Starosolski, Bożena Wieczorek, and Mirosław Wieczorek.

Finally, an exceptional 'Thank you!' I would like to say to my mother and father who are present in my life for the longest time, and who gave me the best possible upbringing.

Tarnowskie Góry
February, 2003

Bibliography

[1] J. Åberg and Yu. M. Shtarkov. Text compression by context tree weighting. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'97)*, pages 377–386, March 25–27, 1997. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 28]

[2] J. Åberg, Yu. M. Shtarkov, and B. J. M. Smeets. Towards understanding and improving escape probabilities in PPM. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'97)*, pages 22–31, March 25–27, 1997. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 25]

[3] S. Albers. Improved randomized on-line algorithms for the list update problem. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 412–419, 1995. [cited at p. 47, 49]

[4] S. Albers and M. Mitzenmacher. Average case analyses of list update algorithms, with applications to data compression. *Algorithmica*, 21(3):312–329, July 1998. [cited at p. 49, 50, 168, 170]

[5] A. Apostolico and S. Lonardi. Compression of biological sequences by greedy off-line textual substitution. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'2000)*, pages 143–152, March 28–30, 2000. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 29]

[6] M. Arimura and H. Yamamoto. Almost sure convergence coding theorem for block sorting data compression algorithm. In *Proceedings of International Symposium on Information Theory and Its Applications (ISITA98), Mexico City, Mexico*, October 14–16, 1998. [cited at p. 45]

[7] M. Arimura and H. Yamamoto. Asymptotic optimality of the block sorting data compression algorithm. *IEICE Transactions on Fundamentals of Electronics Communications & Computer Sciences*, E81-A(10):2117–2122, October 1998. [cited at p. 45]

[8] M. Arimura, H. Yamamoto, and S. Arimoto. A bitplane tree weighting method for lossless compression of gray scale images. *IEICE Transactions on Fundamentals of Electronics Communications & Computer Sciences*, E80-A(11):2268–2271, November 1997. [cited at p. 28]

[9] Z. Arnavut. Generalization of the BWT transformation and inversion ranks. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'2002)*, page 447, April 2–4, 2002. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 57, 105, 107]

[10] Z. Arnavut and S. S. Magliveras. Block sorting and compression. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'97)*, pages 181–190, March 25–27, 1997. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 45, 50, 87]

[11] Z. Arnavut and S. S. Magliveras. Lexical permutation sorting algorithm. *The Computer Journal*, 40(5):292–295, 1997. [cited at p. 45, 87]

[12] R. Arnold and T. C. Bell. A corpus for the evaluation of lossless compression algorithms. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'97)*, pages 201–210, March 25–27, 1997. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 5, 92]

[13] F. D. Awan, N. Zhang, N. Motgi, R. T. Iqbal, and A. Mukherjee. LIPT: A reversible lossless text transform to improve compression performance. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'2001)*, page 481, March 27–29, 2001. [cited at p. 59]

[14] F. S. Awan and A. Mukherjee. LIPT: A lossless text transform to improve compression. In *Proceedings of International Conference on Information and Theory: Coding and Computing, IEEE Computer Society, Las Vegas, Nevada*, pages 452–460, April 2–4 2001. [cited at p. 59]

[15] R. Bachrach and R. El-Yaniv. Online list accessing algorithms and their applications: Recent empirical evidence. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 53–62, 1997. [cited at p. 47]

[16] B. Balkenhol and S. Kurtz. Universal data compression based on the Burrows–Wheeler transformation: Theory and practice. *IEEE Transactions on Computers*, 49(10):1043–1053, October 2000. [cited at p. 35, 43, 46]

[17] B. Balkenhol, S. Kurtz, and Yu. M. Shtarkov. Modifications of the Burrows and Wheeler data compression algorithm. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'99)*, pages 188–197, March 29–31, 1999. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 48, 55, 58, 107]

[18] B. Balkenhol and Yu. M. Shtarkov. One attempt of a compression algorithm using the BWT. Preprint 99-133, SFB343, Discrete Structures in Mathematics, Faculty of Mathematics, University of Bielefeld, Postfach 100131 33501, Bielefeld, Germany, 1999. http://www.mathematik.uni-bielefeld.de/sfb343/preprints/pr99133.ps.gz. [cited at p. 46, 48, 51, 55, 58, 82, 88, 105, 107]

[19] T. Bell and A. Moffat. A note on the DMC data compression scheme. *The Computer Journal*, 32(1):16–20, February 1989. [cited at p. 26]

[20] T. Bell, I. H. Witten, and J. G. Cleary. Modelling for text compression. *ACM Computing Surveys*, 21(4):557–591, 1989. [cited at p. 5, 34, 50, 92]

[21] T. C. Bell. Better OPM/L test compression. *IEEE Transactions on Communications*, COM-34(12):1176–1182, December 1986. [cited at p. 22]

[22] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice Hall, Englewood Cliffs, NJ, 1990. [cited at p. 9, 92]

[23] T. C. Bell and D. Kulp. Longest-match string searching for Ziv–Lempel compression. *Software–Practice and Experience*, 23(7):757–772, July 1993. [cited at p. 22]

[24] T. C. Bell and I. H. Witten. The relationship between greedy parsing and symbolwise text compression. *Journal of the ACM*, 41(4):708–724, 1994. [cited at p. 22]

[25] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 360–369, 1997. [cited at p. 40, 169]

[26] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Communications of ACM*, 29(4):320–330, April 1986. [cited at p. 4, 33, 48, 168, 170]

[27] E. Binder. Distance coder. Usenet group: comp.compression, 2000. [cited at p. 51]

[28] E. Binder. The *dc* program. ftp://ftp.elf.stuba.sk/pub/pc/pack/dc124.zip, 2000. [cited at p. 57]

[29] E. Binder. Private communication, 2001. [cited at p. 51]

[30] Ch. Bloom. Solving the problem of context modeling. http://www.cbloom.com/papers/ppmz.zip, March 1998. [cited at p. 25]

[31] P. F. Brown, S. A. Della Pietra, V. J. Della Pietra, J. C. Lai, and R. L. Mercer. An estimate of an upper bound for the entropy of English. *Computational Linguistics*, 18(1):31–40, 1992. [cited at p. 29]

[32] S. Bunton. The structure of DMC. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'95)*, pages 72–81, March 28–30, 1995. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 26]

[33] S. Bunton. *On-line stochastic processes in data compression*. Ph.D. Thesis, University of Washington, 1996. [cited at p. 25, 26, 27, 108]

[34] S. Bunton. An executable taxonomy of on-line modeling algorithms. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'97)*, pages 42–51, March 25–27, 1997. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 25]

[35] S. Bunton. A generalization and improvement to PPM's blending. UW-CSE Technical Report UW-CSE-97-01-10, The University of Washington, January 1997. ftp://ftp.cs.washington.edu/tr/1997/01/UW-CSE-97-01-10.PS.Z (also Proceedings of DCC'97 p. 426). [cited at p. 25]

[36] S. Bunton. A percolating state selector for suffix-tree context models. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'97)*, pages 32–41, March 25–27, 1997. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 25]

[37] S. Bunton. Semantically motivated improvements for PPM variants. *The Computer Journal*, 40(2-3):76–93, 1997. [cited at p. 25, 107]

[38] S. Bunton. Bayesian state combining for context models (extended abstract). In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'98)*, pages 329–338, March 30–April 1, 1998. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 25]

[39] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. SRC Research Report 124, Digital Equipment Corporation, Palo Alto, California, May 10, 1994. ftp://ftp.digital.com/pub/DEC/SRC/research-reports/SRC-124.ps.zip. [cited at p. 4, 31, 34, 38, 45, 48, 53, 107, 141]

[40] G. Buyanovsky. Associative coding. *The Monitor*, 8:10–22, 1994. In Russian. [cited at p. 107]

[41] G. Buyanovsky. The *acb 2.00c* program. ftp://ftp.elf.stuba.sk/pub/pc/pack/acb_200c.zip, 1997. [cited at p. 107]

[42] J. Carpinelli, A. Moffat, R. Neal, W. Salamonsen, L. Stuiver, A. Turpin, and I. H. Witten. The source codes for word, character, integer, and bit based compression using arithmetic coding. http://www.cs.mu.oz.au/~alistair/arith_coder/arith_coder-3.tar.gz, February 1999. [cited at p. 170]

[43] B. Chapin. Switching between two on-line list update algorithms for higher compression of Burrows–Wheeler transformed data. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'2000)*, pages 183–192, March 28–30, 2000. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 49]

[44] B. Chapin and S. R. Tate. Higher compression from the Burrows–Wheeler transform by modified sorting. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'98)*, page 532, March 30–April 1, 1998. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 57]

[45] X. Chen, S. Kwong, and M. Li. A compression algorithm for DNA sequences and its applications in genome comparison. *Genome Informatics*, 10:51–61, 1999. [cited at p. 29]

[46] J. Cheney. Compressing XML with multiplexed hierarchical PPM models. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'2001)*, pages 163–172, March 27–29, 2001. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 165]

[47] M. G. Ciura and S. Deorowicz. How to squeeze a lexicon. *Software–Practice and Experience*, 31(11):1077–1090, 2001. [cited at p. 6, 29]

[48] J. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, COM-32:396–402, 1984. [cited at p. 3, 23, 24]

[49] J. G. Cleary and W. J. Teahan. Unbounded length context for PPM. *The Computer Journal*, 40(2/3):67–75, 1997. [cited at p. 23, 25, 42]

[50] J. G. Cleary, W. J. Teahan, and I. H. Witten. Unbounded length context for PPM. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'95)*, pages 52–61, March 28–30, 1995. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 42]

[51] G. V. Cormack and R. N. Horspool. Algorithms for adaptive Huffman codes. *Information Processing Letters*, 18(3):159–165, 1984. [cited at p. 14]

[52] G. V. Cormack and R. N. Horspool. Data compression using dynamic Markov modeling. *The Computer Journal*, 30(6):541–550, December 1987. [cited at p. 4, 26, 108]

[53] J. Daciuk, S. Mihov, B. W. Watson, and R. E. Watson. Incremental construction of minimal acyclic finite-state automata. *Computational Linguistics*, 26(1):3–16, 2000. [cited at p. 29]

[54] S. Deorowicz. Improvements to Burrows–Wheeler compression algorithm. *Software–Practice and Experience*, 30(13):1465–1483, 10 November 2000. [cited at p. 5, 84, 102]

[55] S. Deorowicz. Second step algorithms in the Burrows–Wheeler compression algorithm. *Software–Practice and Experience*, 32(2):99–111, 2002. [cited at p. 5, 57, 81, 84, 102]

[56] M. Effros. Universal lossless source coding with the Burrows Wheeler transform. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'99)*, pages 178–187, March 29–31, 1999. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 45]

[57] N. Ekstrand. Lossless compression of grayscale images via context tree weighting. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'96)*, pages 132–139, April 1–3, 1996. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 28]

[58] N. Ekstrand. *Universal Lossless Source Coding Techniques for Images and Short Data Sequences*. Ph.D. Thesis, Lund Institute of Technology, Lund University, 2001. [cited at p. 28]

[59] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21:194–203, 1975. [cited at p. 57, 87]

[60] P. Elias. Interval and recency rank source coding: Two on-line adaptive variable length schemes. *IEEE Transactions on Information Theory*, IT-33:3–10, January 1987. [cited at p. 51]

[61] Erlangen/bethesda data and online services. http://cactus.nci.nih.gov/, 2002. [cited at p. 164]

[62] N. Faller. An adaptive system for data compression. In *Record of the 7th Asilomar Conference on Circuits, Systems, and Computers*, pages 593–597, 1973. [cited at p. 14]

[63] M. Farach. Optimal suffix construction with large alphabets. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science*, pages 137–143, October 1997. [cited at p. 39]

[64] P. Fenwick. A new data structure for cumulative frequency tables. *Software–Practice and Experience*, 24(3):327–336, March 1994. (Errata published in 24(7):677, July 1994.). [cited at p. 15]

[65] P. Fenwick. Experiments with a block sorting text compression algorithm. Technical Report 111, The University of Auckland, Department of Computer Science, May 1995. ftp://ftp.cs.auckland.ac.nz/pub/peter-f/TechRep111.ps. [cited at p. 48]

[66] P. Fenwick. Improvements to the block sorting text compression algorithm. Technical Report 120, The University of Auckland, Department of Computer Science, August 1995. ftp://ftp.cs.auckland.ac.nz/pub/peter-f/TechRep120.ps. [cited at p. 45]

[67] P. Fenwick. Block sorting text compression—final report. Technical Report 130, The University of Auckland, Department of Computer Science, April 1996. ftp://ftp.cs.auckland.ac.nz/pub/peter-f/TechRep130.ps. [cited at p. 34, 35, 48, 82, 88]

[68] P. Fenwick. The Burrows–Wheeler transform for block sorting text compression: Principles and improvements. *The Computer Journal*, 39(9):731–740, 1996. [cited at p. 45, 48, 53, 56, 107, 108]

[69] P. Fenwick. Burrows–Wheeler compression with variable length integer codes. *Software–Practice and Experience*, 32(13):1307–1316, 2002. [cited at p. 57]

[70] E. R. Fiala and D. H. Greene. Data compression with finite windows. *Communications of ACM*, 32(4):490–505, 1989. [cited at p. 22]

[71] A. S. Fraenkel and S. T. Klein. Robust universal complete codes for transmission and compression. *Discrete Applied Mathematics*, 64:31–55, 1996. [cited at p. 57]

[72] J. L. Gailly. The *gzip* program. http://www.gzip.org/, 1993. [cited at p. 108]

[73] R. G. Gallager. Variations on a theme by Huffman. *IEEE Transactions on Information Theory*, 24(6):668–674, 1978. [cited at p. 14]

[74] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19:331–353, 1997. [cited at p. 39]

[75] S. Golomb. Runlength encodings. *IEEE Transactions on Information Theory*, IT-12(3):399–401, July 1966. [cited at p. 45]

[76] Sz. Grabowski. Text preprocessing for Burrows–Wheeler block sorting compression. In *VII Konferencja Sieci i Systemy Informatyczne Łódź'1999, Materiały konferencyjne*, pages 229–239, 1999. [cited at p. 58]

[77] Sz. Grabowski. Private communication, November 2000. [cited at p. 51]

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

[79] P. C. Gutmann and T. C. Bell. A hybrid approach to text compression. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'94)*, pages 225–233, March 29–31, 1994. [cited at p. 22]

[80] D. T. Hoang, P. M. Long, and J. S. Vitter. Dictionary selection using partial matching. *Information Sciences*, 119(1-2):57–72, 1999. [cited at p. 23]

[81] C. A. R. Hoare. Quicksort. *The Computer Journal*, 4:10–15, 1962. [cited at p. 41]

[82] R. N. Horspool. The effect of non-greedy parsing in Ziv-Lempel compression methods. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'95)*, pages 302–311, March 28–30, 1995. [cited at p. 22]

[83] P. G. Howard. The design and analysis of efficient lossless data compression systems. Technical Report CS-93-28, Department of Computer Science, Brown University, Providence, Rhode Island, 1993. http://www.cs.brown.edu/publications/techreports/reports/CS-93-28.html. [cited at p. 25]

[84] P. G. Howard and J. S. Vitter. Analysis of arithmetic coding for data compression. *Information Processing and Management*, 28(6):749–763, 1992. [cited at p. 15]

[85] P. G. Howard and J. S. Vitter. Practical implementations of arithmetic coding. *Image and text compression. Kluwer Academic Publishers*, pages 85–112, 1992. [cited at p. 15]

[86] P. G. Howard and J. S. Vitter. Arithmetic coding for data compression. *Proceedings of the IEEE*, 82(6):857–865, June 1994. [cited at p. 15]

[87] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proceedings of the Institute of Radio Engineers*, pages 1098–1101, September 1952. [cited at p. 13]

[88] S. J. Inglis. *Lossless Document Image Compression*. Ph.D. Thesis, The University of Waikato, March 1999. [cited at p. 29]

[89] S. Irani. Two results on the list update problem. *Information Processing Letters*, 38(6):301–306, 1991. [cited at p. 47]

[90] H. Itoh and H. Tanaka. An efficient method for in memory construction of suffix arrays. In *Proceedings of the IEEE String Processing and Information Retrieval Symposium & International Workshop on Groupware*, pages 81–88, 1999. [cited at p. 5, 41, 62, 101, 141]

[91] F. Jelinek. *Probabilistic Information Theory*. New York: McGraw-Hill, 1968. [cited at p. 14]

[92] A. Kadach. *Effective algorithms for lossless compression of textual data*. Ph.D. Thesis, Russian Science Society, Novosibirsk, Russia, 1997. In Russian, http://www.compression.graphicon.ru/download/articles/lz/kadach_cndd_1997_ps.rar. [cited at p. 50]

[93] D. E. Knuth. Dynamic Huffman coding. *Journal of Algorithms*, 6:163–180, 1985. [cited at p. 14]

[94] D. E. Knuth. *The Art of Computer Programming. Sorting and Searching*, volume 3. Addison-Wesley, Reading, MA, second edition, 1998. [cited at p. 169]

[95] R. E. Krichevsky and V. K. Trofimov. The performance of universal encoding. *IEEE Transactions on Information Theory*, 27:199–207, 1981. [cited at p. 28, 74, 88]

[96] S. Kurtz and B. Balkenhol. Space efficient linear time computation of the Burrows and Wheeler transformation. *Numbers, Information and Complexity, Kluwer Academic Publishers*, pages 375–383, 2000. [cited at p. 39]

[97] G. G. Langdon. An introduction to arithmetic coding. *IBM Journal of Research and Development*, 28(2):135–149, March 1984. [cited at p. 15]

[98] Large Canterbury corpus. http://corpus.canterbury.ac.nz/descriptions/, 1997. [cited at p. 5, 93]

[99] N. J. Larsson. The context trees of block sorting compression. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'98)*, pages 189–198, March 30–April 1, 1998. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 46]

[100] N. J. Larsson. The source code for the Larsson–Sadakane suffix array computation algorithm. http://www.dna.lth.se/~jesper/research.html, 1999. [cited at p. 169]

[101] N. J. Larsson. *Structures of String Matching and Data Compression*. Ph.D. Thesis, Department of Computer Science, Lund University, Box 118, S-22100 Lund, Sweden, September 1999. [cited at p. 46]

[102] N. J. Larsson and K. Sadakane. Faster suffix sorting. Technical Report LU-CS-TR:99-214, Department of Computer Science, Lund University, Box 118, S-22100 Lund, Sweden, May 1999. http://www.cs.lth.se/~jesper/ssrev-tr.pdf. [cited at p. 40, 101, 142, 167, 169]

[103] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy-Kan, and D. B. Shmoys, editors. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. John Wiley and Sons, Chichester, West Sussex, New York, 1985. [cited at p. 58]

[104] D. Loewenstern and P. N. Yianilos. Significantly lower entropy estimates for natural DNA sequences. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'97)*, pages 151–160, March 25–27, 1997. [cited at p. 29]

[105] G. Lyapko. The *lgha* program. http://www.geocities.com/SiliconValley/Lab/6606/lgha.htm, 1999. [cited at p. 108]

[106] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. [cited at p. 40, 101, 142, 167, 169]

[107] G. Manzini. An analysis of the Burrows–Wheeler transform (extended abstract). In *Proceedings of the tenth annual ACM-SIAM symposium on discrete algorithms (SODA'99)*, pages 669–677, 1999. [cited at p. 45]

[108] G. Manzini. The Burrows–Wheeler transform: Theory and practice. *Lecture Notes in Computer Science, Springer Verlag*, 1672:34–47, 1999. [cited at p. 45]

[109] G. Manzini. An analysis of the Burrows–Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001. [cited at p. 45]

[110] J. McCabe. On serial files with relocatable records. *Operations Research*, 13:609–618, 1965. [cited at p. 46]

[111] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976. [cited at p. 38]

[112] V. Miller and M. Wegman. Variations on a theme be Ziv and Lempel. In *A. Apostolico and Z. Galil, editors, Combinatorial Algorithms on Words, Volume 12*, pages 131–140, 1985. NATO ASI Series F. [cited at p. 23]

[113] A. Moffat. Implementing the PPM data compression scheme. *IEEE Transactions on Communications*, 28(11):1917–1921, November 1990. [cited at p. 25]

[114] A. Moffat. An improved data structure for cumulative probability tables. *Software–Practice and Experience*, 29(7):647–659, 1999. [cited at p. 15]

[115] A. Moffat, R. M. Neal, and I. H. Witten. Arithmetic coding revisited. *ACM Transactions on Information Systems*, 16(3):256–294, 1998. [cited at p. 15, 170]

[116] A. Moffat and A. Turpin. *Compression and Coding Algorithms*. Kluwer Academic Publishers, Norwell, Massachusetts, 2002. [cited at p. 9]

[117] Mozilla Project. http://www.mozilla.org/, 1998–2002. [cited at p. 163]

[118] M. Nelson and J. L. Gailly. *The Data Compression Book*. M&T Books, New York, NY, second edition, 1995. [cited at p. 9]

[119] C. G. Nevill-Manning and I. H. Witten. Protein is incompressible. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'99)*, pages 257–266, March 29–31, 1999. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 29]

[120] Open source database benchmark. http://sourceforge.net/projects/osdb, 2002. [cited at p. 164]

[121] Openoffice.org source project. http://www.openoffice.org, 2002. [cited at p. 164]

[122] V. Pareto. *Cours d'économie politique, professé à l'Université de Lausanne, tome I*. Lausanne: Rouge, 1896. [cited at p. 97]

[123] V. Pareto. *Cours d'économie politique, professé à l'Université de Lausanne, tome II*. Lausanne: Rouge, 1897. [cited at p. 97]

[124] V. Pareto. *Manuale di economia politica, conuna introduzione alla scienca Sociale*, volume in-8. Milano, 1906. [cited at p. 97]

[125] R. Pasco. *Source coding algorithms for fast data compression*. Ph.D. Thesis, Department of Electrical Engineering, Stanford University, Stanford, 1976. [cited at p. 15]

[126] I. Pavlov. The *ufa 0.04 beta 1* program. ftp://ftp.elf.stuba.sk/pub/pc/pack/ufa004b1.zip, 1998. [cited at p. 109]

[127] I. Pavlov. The *7-zip 2.30 beta 17* program. http://www.7-zip.org, 2002. [cited at p. 108]

[128] G. Perec. *La Disparition*. Denoël, 1969. In French. [cited at p. 12]

[129] G. Perec. *A Void*. Harvill, 1994. (English translation by A. Gilbert). [cited at p. 12]

[130] N. Porter, editor. *Webster's Revised Unabridged Dictionary*. C. & G. Merriam Co., Springfield, Massachusetts, 1913. [cited at p. 96, 165]

[131] Project Gutenberg. http://www.promo.net/pg/, 1971–2002. [cited at p. 96, 163, 165]

[132] N. Reingold and J. Westbrook. Off-line algorithms for the list update problem. *Information Processing Letters*, 60:75–80, 1996. [cited at p. 47]

[133] W. Reymont. *Chłopi*. 1904–09. [cited at p. 96, 164]

[134] J. Rissanen. Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20:198–203, 1976. [cited at p. 15]

[135] J. Rissanen. Complexity of strings in the class of Markov sources. *IEEE Transactions on Information Theory*, IT-32:526–532, July 1986. [cited at p. 18]

[136] J. Rissanen and G. G. Langdon. Arithmetic coding. *IBM Journal of Research and Development*, 23:149–162, 1979. [cited at p. 15]

[137] E. Roshal. The *rar 2.90* program. http://www.rarlab.com, 2001. [cited at p. 108]

[138] F. Rubin. Arithmetic stream coding using fixed precision registers. *IEEE Transactions on Information Theory*, IT-25(6):672–675, November 1979. [cited at p. 15]

[139] K. Sadakane. A fast algorithm for making suffix arrays and for Burrows–Wheeler transformation. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'98)*, pages 129–138, March 30–April 1, 1998. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 40]

[140] K. Sadakane, T. Okazaki, and H. Imai. Implementing the context tree weighting method for text compression. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'2000)*, pages 123–132, April 2–4, 2000. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 28]

[141] The Samba Project. http://www.samba.org, 2002. [cited at p. 164]

[142] SAO star catalogue. http://tdc-www.harvard.edu/software/catalogs/sao.html, 2002. [cited at p. 164]

[143] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann Publishers, second edition, 2000. [cited at p. 9]

[144] M. Schindler. A fast block-sorting algorithm for lossless data compression. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'97)*, page 469, March 25–27, 1997. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 48, 108]

[145] M. Schindler. The *szip 1.11 beta* program, 1998. http://www.compressconsult.com/szip/. [cited at p. 108]

[146] F. Schulz. *Adaptive Suchverfahren*. Ph.D. Thesis, der Universitat des Saarlandes, Saarbrücken, Germany, 1999. In German. [cited at p. 82]

[147] R. Sedgewick. A new upper bound for Shellsort. *Journal of Algorithms*, 7:159–173, 1986. [cited at p. 169]

[148] J. Seward. The *bzip 0.21* program, 1996. http://www.muraroa.demon.co.uk. [cited at p. 107, 108]

[149] J. Seward. On the performance of BWT sorting algorithms. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'2000)*, pages 173–182, March 28–30, 2000. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 41, 101, 141, 168, 169]

[150] J. Seward. The *bzip2* program, 2002. http://www.muraroa.demon.co.uk. [cited at p. 53, 101, 142, 168, 169]

[151] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423,623–656, 1948. [cited at p. 12]

[152] C. E. Shannon. Prediction and entropy of printed English. *Bell System Technical Journal*, 30:50–64, 1951. [cited at p. 29]

[153] D. L. Shell. A high-speed sorting procedure. *Communications of ACM*, 2(7):30–32, 1959. [cited at p. 169]

[154] D. Shkarin. Improving the efficiency of PPM algorithm. *Problems of Information Transmission*, 34(3):44–54, 2001. In Russian, http://sochi.net.ru/~maxime/doc/PracticalPPM.ps.gz. [cited at p. 25, 108]

[155] D. Shkarin. PPM: One step to practicality. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'2002)*, pages 202–211, April 2–4, 2002. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 25, 108, 119, 136, 144]

[156] D. Shkarin. The *PPMII/cPPMII* program. ftp://ftp.elf.stuba.sk/pub/pc/pack/ppmdi.rar, 2002. [cited at p. 25, 108, 136]

[157] W. Skarbek. *Metody reprezentacji obrazów cyfrowych*. Akademicka Oficyna Wydawnicza PLJ, Warszawa, 1993. In Polish. [cited at p. 9]

[158] M. Smirnov. The *PPMNB1+* program. ftp://ftp.elf.stuba.sk/pub/pc/pack/ppmnb1.rar, 2002. [cited at p. 108]

[159] J. Storer and T. G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29:928–951, 1982. [cited at p. 22]

[160] I. Sutton. The *boa 0.58b* program. ftp://ftp.elf.stuba.sk/pub/pc/pack/boa058.zip, 1998. [cited at p. 107]

[161] J. Suzuki. A relationship between context tree weighting and general model weighting techniques for tree sources. *IEICE Transactions on Fundamentals of Electronics Communications & Computer Sciences*, E81-A(11):2412–2417, November 1998. [cited at p. 28]

[162] W. J. Teahan. The *ppmd+* program. ftp://ftp.cs.waikato.ac.nz/pub/compression/ppm/ppm.tar.gz, 1997. [cited at p. 108]

[163] W. J. Teahan. *Modelling English text*. Ph.D. Thesis, University of Waikato, Hamilton, New Zealand, May 1998. [cited at p. 29, 59, 108]

[164] W. J. Teahan and J. G. Cleary. The entropy of English using PPM-based models. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'96)*, pages 53–62, April 1–3, 1996. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 29]

[165] W. J. Teahan and J. G. Cleary. Models of English text. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'97)*, pages 12–21, March 25–27, 1997. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 29]

[166] W. J. Teahan and J. G. Cleary. Tag based models of English text. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'98)*, pages 43–52, March 30–April 1, 1998. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 29]

[167] W. J. Teahan, S. Inglis, J. G. Cleary, and G. Holmes. Correcting English text using PPM models. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'98)*, pages 289–298, March 30–April 1, 1998. [cited at p. 29]

[168] B. Teia. A lower bound for randomised list update algorithms. *Information Processing Letters*, 47:5–9, 1993. [cited at p. 47]

[169] J. Teuhola and T. Raita. Application of a finite-state model to text compression. *The Computer Journal*, 36(7):607–614, 1993. [cited at p. 26]

[170] T. J. Tjalkens, P. A. J. Volf, and F. M. J. Willems. A context-tree weighting method for text generating sources. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'97)*, page 472, March 25–27, 1997. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 28]

[171] T. J. Tjalkens and F. M. J. Willems. Implementing the context-tree weighting method: Arithmetic coding. In *International Conference on Combinatorics, Information Theory & Statistics*, page 83, Portland, July 18–20 1997. [cited at p. 28]

[172] K. Tsai-Hsing. Improving suffix-array construction algorithms with applications. Master's thesis, Gunma University, Kiryu, 376-8515, Japan, 2001. http://www.isl.cs.gunma-u.ac.jp/Yokoo/dataroom/paper/00/thkao/thesis.ps. [cited at p. 63]

[173] E. Ukkonen. On-line construction of suffix tress. *Algorithmica*, 14(3):249–260, 1995. [cited at p. 39]

[174] V. N. Vapnik. *Estimation of Dependencies based on Empirical Data*. Springer Verlag, NJ, 1982. [cited at p. 74]

[175] S. Verdu. Private communication, August 2001. [cited at p. 70]

[176] J. S. Vitter. Design and analysis of dynamic Huffman codes. *Journal of the ACM*, 34(4):825–845, 1987. [cited at p. 14]

[177] J. S. Vitter. Algorithm 673: dynamic Huffman coding. *ACM Transactions on Mathematical Software*, 15(2):158–167, 1989. [cited at p. 14]

[178] Vitrual library of Polish literature. http://monika.univ.gda.pl/~literat/books.htm, 2002. [cited at p. 164]

[179] P. A. J. Volf. Context-tree weighting for text-sources. In *Proceedings of the IEEE International Symposium on Information Theory, Ulm, Germany*, page 64, June 29–July 4, 1997. [cited at p. 28]

[180] P. A. J. Volf and F. M. J. Willems. Context-tree weighting for extended tree sources. In *Proceedings of the 17th Symposium on Information Theory in the Benelux, Enschede, The Netherlands*, pages 95–101, May 30–31, 1996. [cited at p. 28]

[181] P. A. J. Volf and F. M. J. Willems. A context-tree branch-weighting algorithm. In *Proceedings of the 18th Symposium on Information Theory in the Benelux, Veldhoven, The Netherlands*, pages 115–122, May 15–16, 1997. [cited at p. 28]

[182] P. A. J. Volf and F. M. J. Willems. Switching between two universal source coding algorithms. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'98)*, pages 491–500, March 30–April 1, 1998. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 28, 49, 109]

[183] P. A. J. Volf and F. M. J. Willems. The switching method: Elaborations. In *Proceedings of the 19th Symposium on Information Theory in the Benelux, Veldhoven, The Netherlands*, pages 13–20, May 28–29, 1998. [cited at p. 28, 49, 109]

[184] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behaviour*. Princeton University Press, Princeton, 1953. [cited at p. 98]

[185] M. J. Weinberger, J. Rissanen, and M. Feder. A universal finite memory source. *IEEE Transactions on Information Theory*, IT-41:643–652, May 1995. [cited at p. 19]

[186] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the* 14*th IEEE Annual Symposium on Switching and Automata Theory, The University of Iowa*, pages 1–11, 1973. [cited at p. 38]

[187] T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984. [cited at p. 23, 108]

[188] F. M. J. Willems. The context-tree weighting method: Extentensions. *IEEE Transactions on Information Theory*, pages 792–798, March 1998. [cited at p. 28]

[189] F. M. J. Willems, Yu. M. Shtarkov, and T. J. Tjalkens. The context tree weighting method: Basic properties. *IEEE Transactions on Information Theory*, 41:653–664, May 1995. [cited at p. 4, 27, 107]

[190] F. M. J. Willems, Yu. M. Shtarkov, and T. J. Tjalkens. Context weighting for general finite-context sources. *IEEE Transactions on Information Theory*, 42(5):1514–1520, September 1996. [cited at p. 28]

[191] F. M. J. Willems, Yu. M. Shtarkov, and T. J. Tjalkens. Reflections on: The context-tree weighting method: Basic properties. *IEEE Information Theory Society Newsletter*, 47:1, 20–27, March 1997. [cited at p. 28]

[192] F. M. J. Willems and T. J. Tjalkens. Complexity reduction of the context-tree weighting method. In *Proceedings of the 18th Symposium Information Theory in Benelux, Veldhoven*, pages 123–130, May 15–16, 1997. [cited at p. 28]

[193] R. N. Williams. An extremely fast Ziv–Lempel data compression algorithm. In *J. A. Storer and J. H. Reif, editors, Proceedings of the IEEE Data Compression Conference (DCC'91)*, pages 362–371, April 8–11, 1991. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 22]

[194] R. N. Williams. The LZRW algorithms family. http://www.ross.net/compression/introduction.html, 1991. [cited at p. 22]

[195] A. I. Wirth and A. Moffat. Can we do without ranks in Burrows Wheeler transform compression. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'2001)*, pages 419–428, 2001. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 56, 82, 88, 105, 109]

[196] I. H. Witten and T. C. Bell. The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, 37(3):1085–1094, 1991. [cited at p. 25]

[197] I. H. Witten, Z. Bray, M. Mahoui, and B. Teahan. Text mining: A new frontier for lossless compression. In *J. A. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference (DCC'99)*, March 29–31, 1999. IEEE Computer Society Press, Los Alamitos, California. [cited at p. 29]

[198] 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. [cited at p. 9]

[199] I. H. Witten, R. M. Neal, and J. F. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987. [cited at p. 15]

[200] V. Yoockin. The *ybs 0.03* program. ftp://ftp.elf.stuba.sk/pub/pc/pack/ybs003ew.zip, 2002. [cited at p. 105, 109]

[201] T. L. Yu. Dynamic Markov compression. *Dr. Dobb's Journal*, (243):30–33, 96–100, January 1996. [cited at p. 26]

[202] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23:337–343, 1977. [cited at p. 3, 21, 108]

[203] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, IT-24:530–536, 1978. [cited at p. 3, 21, 22]

Appendices

Appendix A

Silesia corpus

In this appendix, a more detailed description of the files in the Silesia corpus is presented.

dickens

Charles Dickens wrote many novels. The file is a concatenation of fourteen of his works that can be found in the Project Gutenberg [131]. This is a simple text file. The included novels are: *A Child's History Of England*, *All The Year Round: Contributions*, *American Notes*, *The Battle Of Life*, *Bleak House*, *A Christmas Carol*, *David Copperfield*, *Dombey And Son*, *Doctor Marigold*, *Going Into Society*, *George Silverman's Explanation*, *Barnaby Rudge: a tale of the Riots of 'eighty*, *The Chimes*, *The Cricket On The Hearth*.

mozilla

A Mozilla 1.0 [117] open source web browser was installed on the Tru64 UNIX operating system and then the contents of the Mozilla.org directory were tarred. There are 525 files of types such as: executables, jar archives, HTML, XML, text, and others.

mr

A magnetic resonance medical picture of a head. This file is stored in a DICOM format and contains 19 planes.

nci

The chemical databases of structures contain information of structures, their components, 2D or 3D coordinates, properties, etc. The file is a part of the August 2000 2D file stored in an SDF format which is a common file format developed to handle a list of molecular structures associated with properties [61]. The original database is of size 982 MB so it is necessary to truncate it to be suitable for a part of the corpus. The 32 MB piece (rounded down to the nearest end of the record) is taken from about the middle of the original file (starting at the first record after leaving 400 MB of data).

ooffice

An OpenOffice.org [121] is an open source project, which is composed of the word processor, spreadsheet program, presentation maker, and graphical program. The file is a dynamic linked library from version 1.01 for Windows operating system.

osdb

An Open Source Database Benchmark [120] is a project created to provide a free test for database systems. Some of the parts of the project are sample databases. The 40 MB benchmark was run on a MySQL 3.23 server. The file is one of the MySQL database files, hundred.med.

reymont

A book *Chłopi* [133] by Władysław Reymont was honoured the Nobel Price in 1924. The text of the book was taken from the Virtual Library of Polish Literature [178]. Then it was converted to the LaTeX files from which the uncompressed PDF file was produced. The file is uncompressed, because the build-in compression in the PDF format is rather poor, and much better results can be obtained when we compress the uncompressed PDF files.

samba

Samba [141] is an open source project that is intended to be a free alternative to the SMB/CIFS clients. The file contains tarred source code (also documentation and graphics) of the Samba 2.2-3 version.

sao

There are many star catalogues containing the data of sky objects. The chosen one, SAO catalogue [142], is suitable especially for the amateur astronomers. It contains the description of 258,996 stars, and is composed of binary records.

webster

The 1913 Webster Unabridged Dictionary [130] is an English dictionary stored in a rather simple HTML. The file is a concatenation of files that can be obtained from Project Gutenberg [131].

xml

A concatenation of 21 XML files used by Cheney [46]. The chosen files are highly, mixed, and little structured and tagged.

x-ray

An X-ray medical picture of child's hand. This is a 12-bit grayscale image.

Appendix B

Implementation details

The compression program introduced in the thesis is implemented in C++. Here we provide a brief description of its usage and the contents of source code files.

B.1 Program usage

The compression program *sdc* can be executed with several command line parameters to choose the compression method. The syntax of the execution is:

sdc ⟨e|d|i⟩ [switches] ⟨input_file_name⟩ ⟨output_file_name⟩

The options are:

- e—encoding,
- d—decoding; if this option is chosen, then switches are ignored,
- i—show info of the compressed file; if this option is chosen, then switches are ignored.

The available switches are:

- –b⟨size⟩—sets the block size. The proper values are expressed in MB and must lie in the range $[1, 128]$. It should be remembered that the memory used by program is about 9 times greater than the block size. The special value 0 sets the block size equal to the size of the file to be compressed but not greater than 128 MB. The default value is 0.
- –f{L|S|I|C}—chooses the BWT computation method. The available options are:
  - M—the Manber–Myers's method [106],
  - L—the Larsson–Sadakane's method [102],

  - S—the improved Seward's *copy* method [149] implemented in the *bzip2* program [150], with the Manber–Myers's method as a fallback,
  - IT1—the improved Itoh–Tanaka-based method (Section 4.1.2) of order 1, with the Manber–Myers's method as a fallback,
  - IT2—the improved Itoh–Tanaka-based method of order 2, with the Manber–Myers's method as a fallback,
  - IT—the improved Itoh–Tanaka-based method of order chosen relating on the contents of the file, with the Manber–Myers's method as a fallback,
  - C—the combined method; depending on the contents of the file, the program chooses between all the available methods.

  The choice of the method affects the compression speed only. Usually the fastest method is C, but the best worst-case time complexity gives the L method. The default value is C.

- –s{W|M|T}—chooses the second stage transform. The available options are:

  - M—the move-to-front transform [26],
  - T—the time-stamp(0) transform [4],
  - W—the weighted frequency count transform (Section 4.1.5).

  The default value is W.

- –w⟨wf⟩—chooses the weight function if the WFC transform is used. (Always the quantised versions of the weight functions are used.) The proper values for wf are from the range $[1, 9]$. The weight functions are shown in Figure 4.8. The default value is 9.

- –t⟨dist⟩—chooses the maximum distance in the WFC weight functions. The proper values are from the range $[1, 15]$. The distance is set according to the rule $t_{max} = 2^{\text{dist}}$. The default value is 11.

- –m{0|1|2}—chooses the version of the move-to-front transform. The available options are:

  - 0—the original MTF transform,
  - 1—the MTF-1 transform,
  - 2—the MTF-2 transform.

  The default value is 2.

B.2 Source code

Burrows–Wheeler transform

Several BWT computation methods are implemented in the compression program. The Larsson–Sadakane's method [102], implemented in a class CLSBWT, is located in the files lsbwt.cpp and lsbwt.h. This source code is based on the original implementation by Larsson and Sadakane [100].

The other BWT computation methods are implemented in a class CSITBWT, located in the source code files sitbwt.cpp and sitbwt.h. The main sorting procedure is based on the Seward's method [149]. It executes as follows:

1. Sort the suffixes with the bucket sort procedure according to one (large buckets) and two (small buckets) initial symbols.

2. Sort the large buckets with the Shell sort [153] procedure according to their size.

3. Sort the suffixes in the buckets with a ternary quick sort procedure [25]. As the pivot in the quick sort procedure a median of three is used. If the number of suffixes to be sorted is less than some threshold then the Shell sort procedure is used.

4. If a level of recursive calls of the quick sort procedure is higher than an assumed number, the Shell sort procedure for sorting the remaining part is executed.

5. If the sorting takes too long (longer than the assumed complexity) the quick sort procedure is stopped and the fallback procedure is used, which is the Manber–Myers's method [106].

We introduced several improvements to the implementation based on the *bzip2* program [150]. The most important are:

- The increments in the Shell sort procedure [153] are replaced from Knuth proposal [94] to Sedgewick proposal [147]. The experiments show that this change accelerates the sorting process.

- If the number of suffixes to be sorted with the quick sort procedure is larger than some threshold, a pseudo-median of nine is used to choose the pivot.

- If the number of suffixes is small (lest than 8) an insertion sort procedure is employed.

- A number of the recursive levels of the quick sort procedure is increased from 12 to 256.

- A speed up for sorting strings with a long common prefix is used. Such strings are compared in a special way without the recursive calls of the quick sort procedure.

The class CSITBWT includes also an implementation of the improved Itoh–Tanaka's method (Section 4.1.2). The working of the Itoh–Tanaka's method is as follows:

1. Sort the suffixes with the bucket sort procedure according to one (large buckets) and two (small buckets) initial symbols, splitting meanwhile the buckets into types D, E, and I.

2. Calculate what is the number of buckets of type E. If it is higher than $0.07n$, then use the improved Itoh–Tanaka's method of order 1. In the other case, use the improved Itoh–Tanaka's method of order 2.

3. Calculate the total number of suffixes of types D and I. Choose the smaller number and use according to it a forward or a reverse lexicographic order to sort with the string-sorting procedure the smaller number of suffixes.

4. Execute the improved Seward's method for sorting strings. If the improved Itoh–Tanaka's method of order 1 is used, the special processing of the suffixes of type E is cared during sorting.

5. Sort the remaining unsorted suffixes.

Other stages

A weighted frequency count transform (Section 4.1.5) is implemented in a class CWFC, stored in the files wfc.cpp and wfc.h. The implementation is highly optimised for speed to minimise the cost of maintaining the list L. The worst-case time complexity of this implementation is $O(nlk)$.

A move-to-front transform [26] is implemented in a class CMTF located in the files mtf.cpp and mtf.h. This is an optimised implementation of the worst-case time complexity $O(nk)$.

A time-stamp(0) transform [4] is implemented in a class CTS located in the files ts.cpp and ts.h. This is an implementation of the worst-case time complexity $O(nk)$.

A zero run length encoding is implemented in a class CRLE0 located in the files rle0.cpp and rle0.h. The worst-case time complexity of this transform is $O(n)$.

The arithmetic coding class CBAC is an implementation based on the source code by Moffat *et al.* [42] presented, in 1998, with the paper revisiting the arithmetic coding [115]. This class implements the weighted probability estimation method introduced in this thesis, as well as the coding of symbols being the output of the RLE-0 stage into a binary code. This class is located in the files aac.cpp and aac.h.

Auxiliary program components

A class CCompress is a class gathering all the stages of the proposed compression algorithm. It is located in the files compress.cpp and compress.h.

All the classes implementing the stages of the compression algorithm inherits from the class CStage. An implementation of this abstract class is located in the files stage.cpp and stage.h. As all the stages of the compression algorithm are implemented in the separate classes, the memory buffers are necessary to store the intermediate results. To this end, a class CMemBuf is implemented. It is located in the files membuf.cpp and membuf.h. For easy processing of parameters of the compression algorithm, e.g., the version of the move-to-front transform, the chosen weight function, etc. a class CParameters is designed. It is located in the files pars.cpp and pars.h.

The main class of the compression program, CSDC, implements the full functionality of the utility *sdc*. It is located in the files sdc.cpp and sdc.h.

Appendix C

Detailed options of examined compression programs

This appendix contains a more detailed description of the options chosen for the examined programs.

7-zip

The method LZMA is chosen with the option –m0=LZMA. The memory for the dictionary is set to be 10 times larger (but not less than 16 MB) than the size of the file to compress. The option –mx is used to get the maximum compression.

acb 2.00c

The maximum compression method, u, is used.

boa 0.58b

The program is executed with a compression method –m15.

bzip 0.21

The block size is set, with the option –9, to the maximal possible value, 900 kB.

compress

The maximum size of the dictionary is used.

DW, DM

The size of the block is chosen to be the size of the file to compress.

gzip

The option –9 is used for maximum compression.

lgha

The default options are used.

PPMd var. H

The PPM order is set to 16 with the option –o16. The memory limit is set, with the option –m, to be 10 times the size of the file to compress, but not less than 16 MB.

ppmnb1+

The order of the PPM is set to a pseudo order 9 with the option –O9. The memory limit is set, with the option –M:50, to maximum possible value, which is 50 MB. Also options –DA, –ICd, –E8d are used to disable data-specific filters.

PPMonstr var. H

The PPM order is set to 16 with the option –o16. The memory limit is set, with the option –m, to be 10 times the size of the file to compress, but not less than 16 MB.

PPMonstr var. I

The PPM order is set to 4 and 64, with the options –o4 and –o64 respectively, as we examine two sets of parameters for this program. The memory limit is set, with the option –m, to be 10 times the size of the file to compress, but not less than 16 MB. The special memory limit of 256 MB was set for file mozilla, because with larger memory limit for this file the program crushed.

rar 2.90

The program is used with a compression method –m3. The maximum size of the dictionary is chosen with the option –md4096. The multimedia filters are turned off using –mm–.

szip

The order of the BWT is set to unlimited (as in the classical BWT) with the option –o0. The size of the block is set to 4.1 MB (maximal possible value) with the option –b41.

ufa 0.04 Beta 1

The program is executed with the option –m5, which chooses the binary PPM algorithm.

ybs

The block size is set to 16 MB, which is the maximum possible value, with the option –m16m.

Appendix D

Illustration of the properties of the weight functions

This appendix contains additional figures presenting the results of the experiments with probability estimation methods for the output of the piecewise stationary memoryless source.

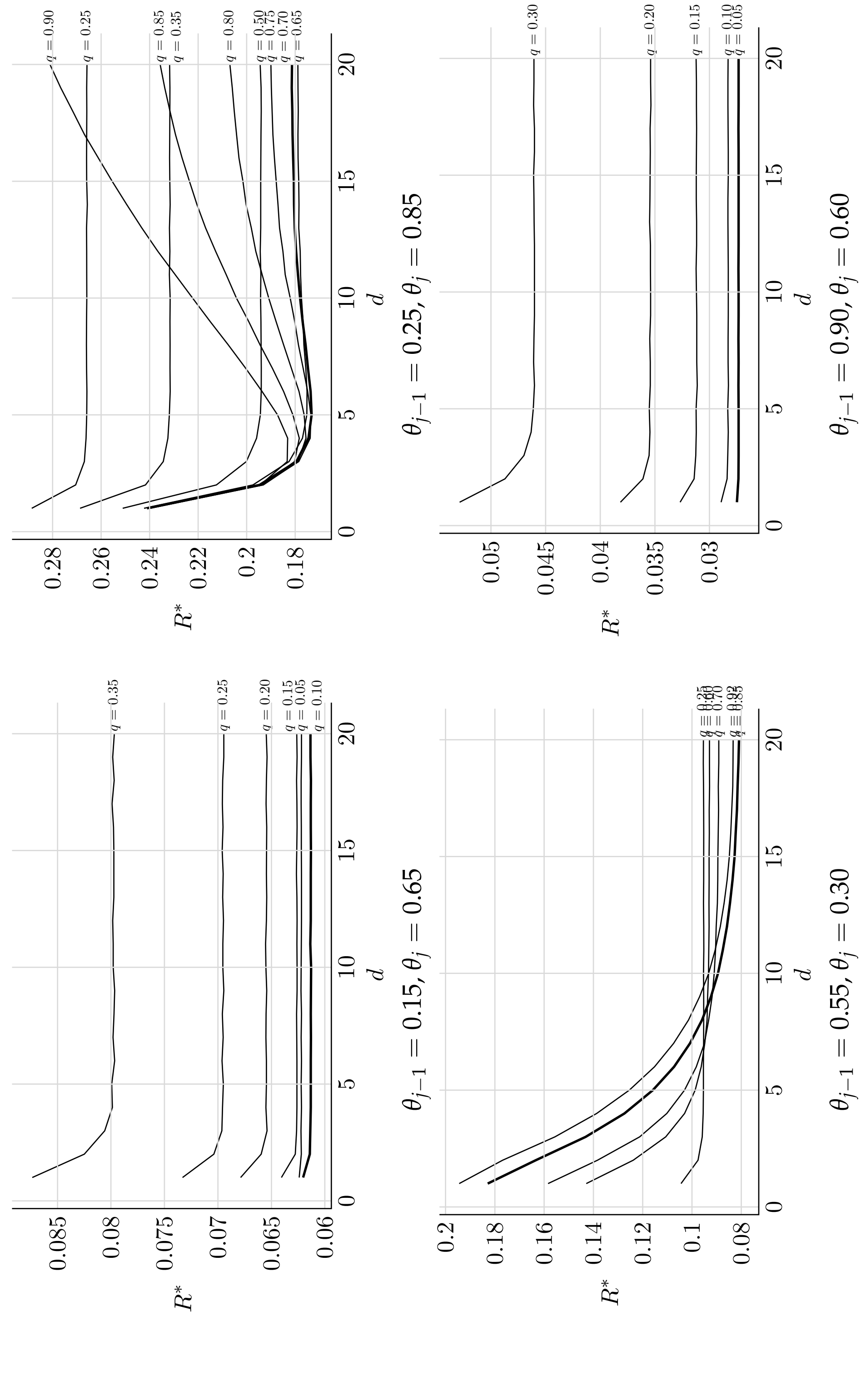

Figure D.1: Illustration of the properties of the weight function w_2 for $m = 20$

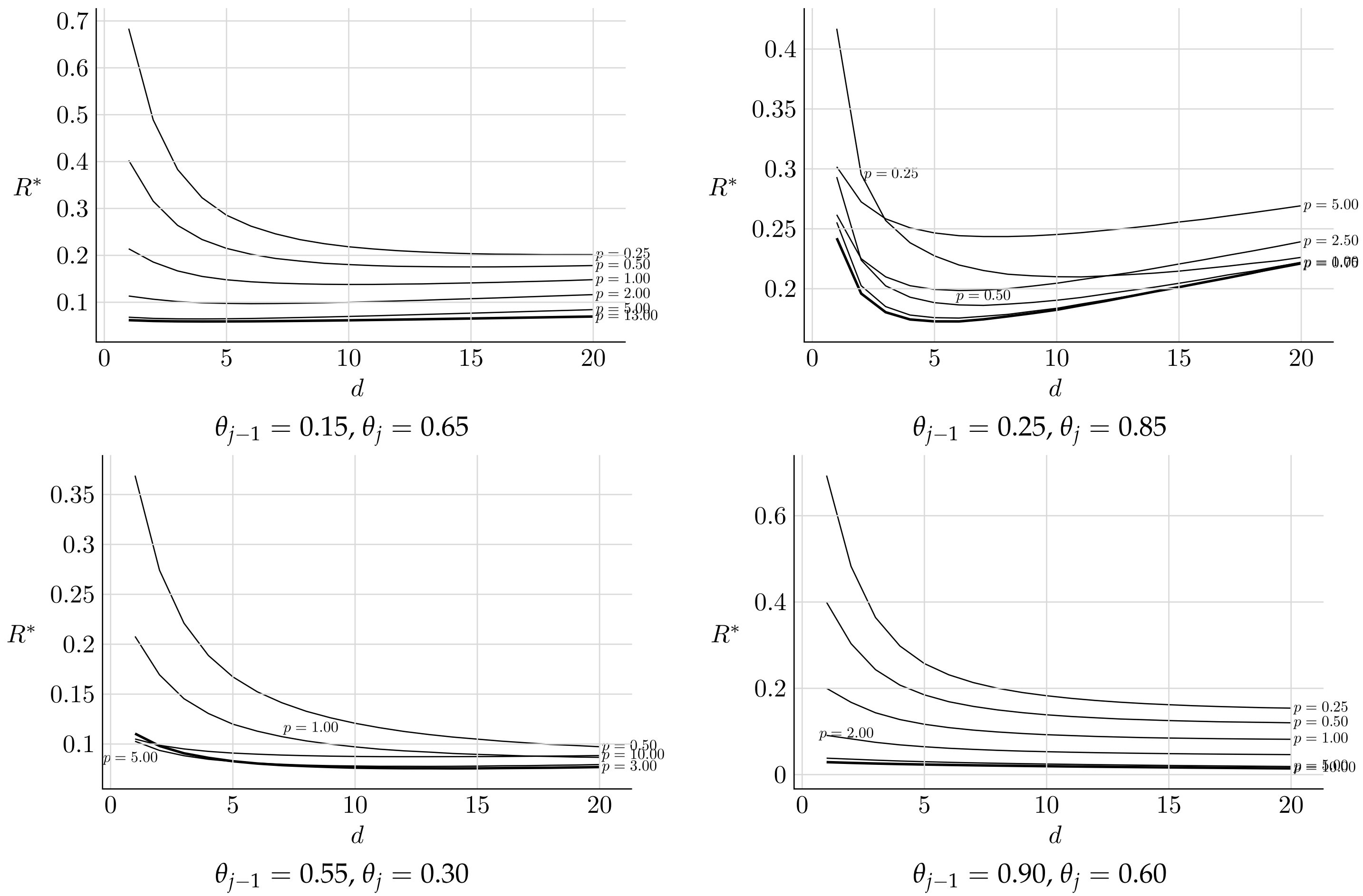

Figure D.2: Illustration of the properties of the weight function w_3 for $m = 20$

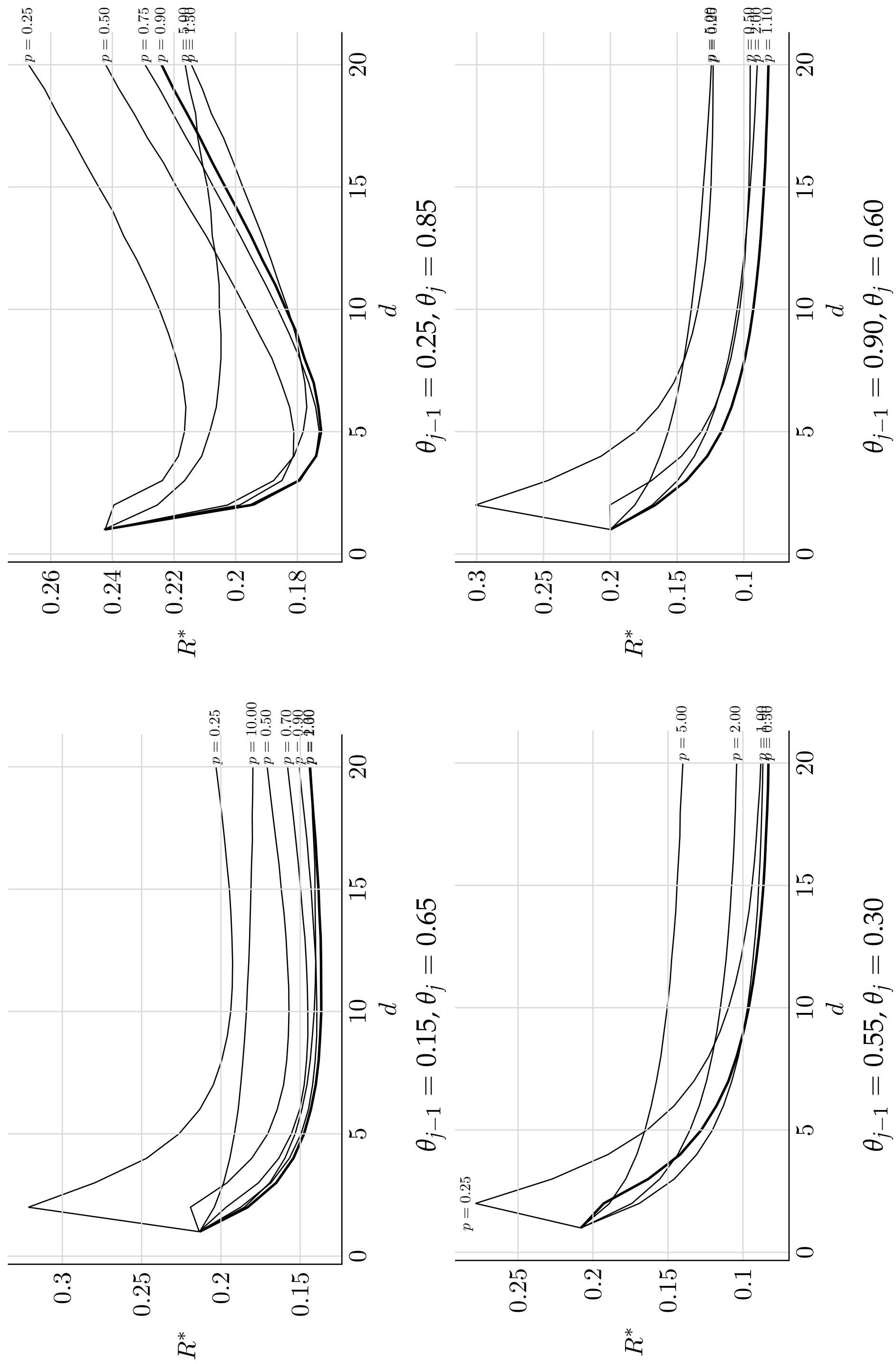

Figure D.3: Illustration of the properties of the weight function w_4 for $m = 20$

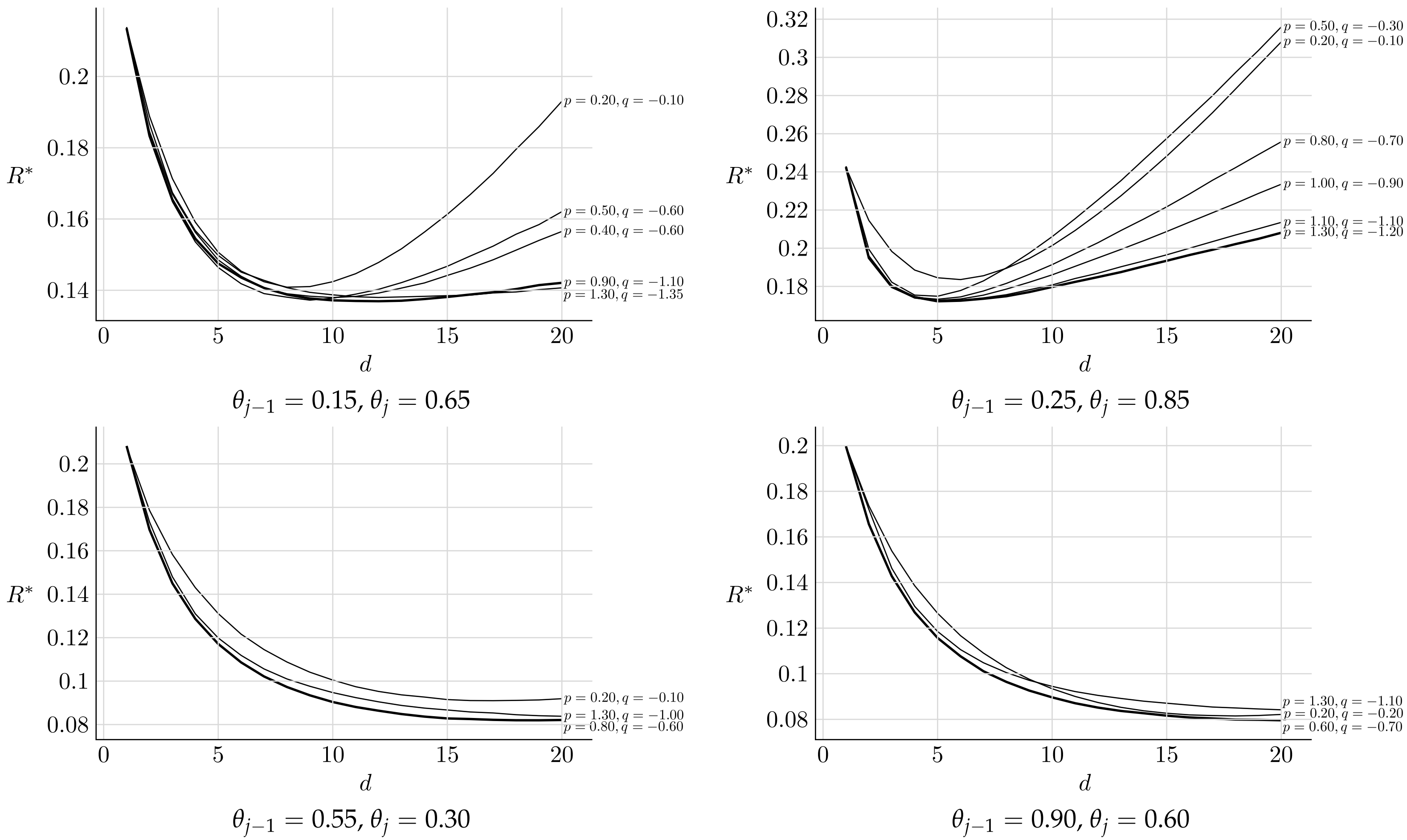

Figure D.4: Illustration of the properties of the weight function w_5 for $m = 20$

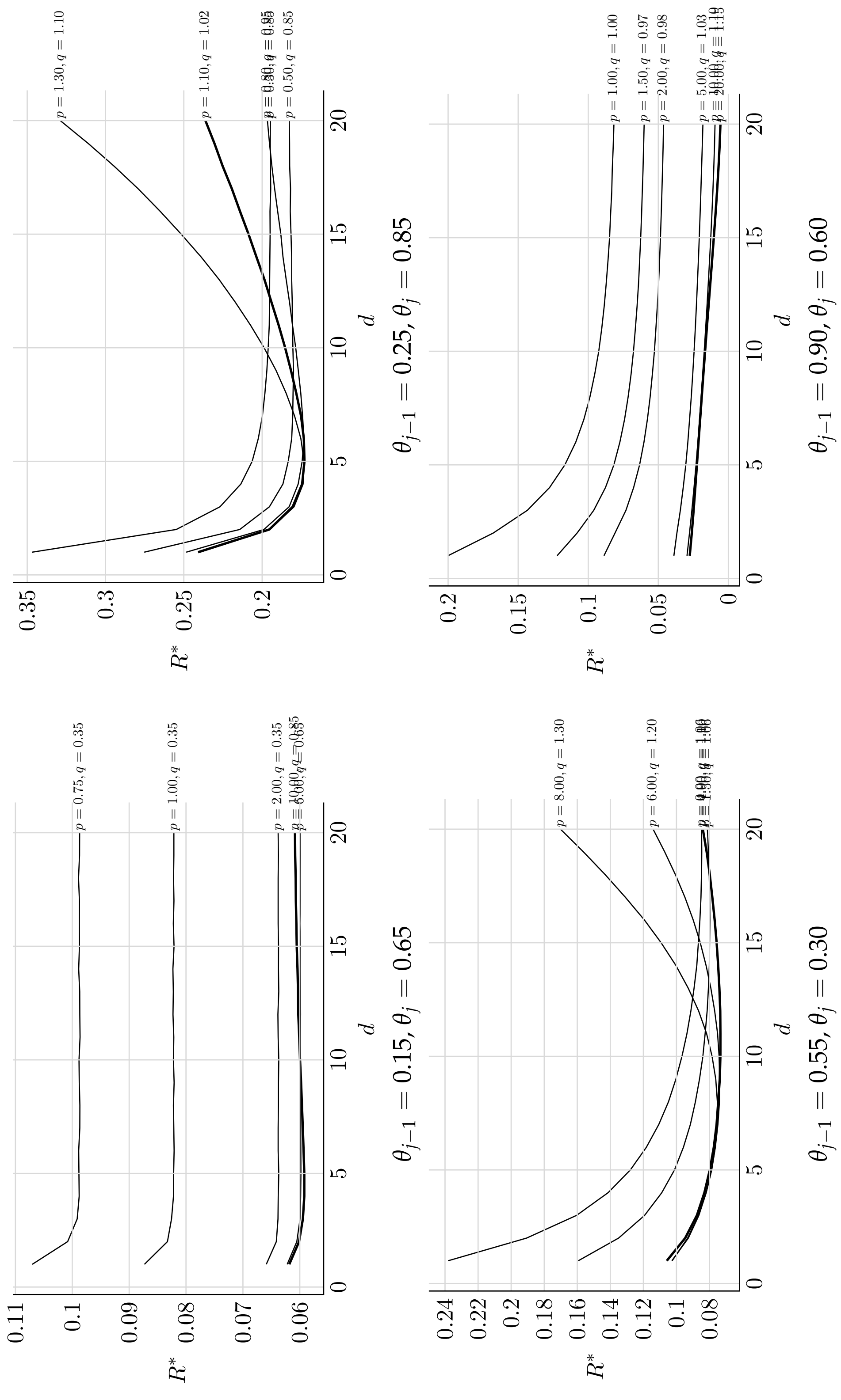

Figure D.5: Illustration of the properties of the weight function w_6 for $m = 20$

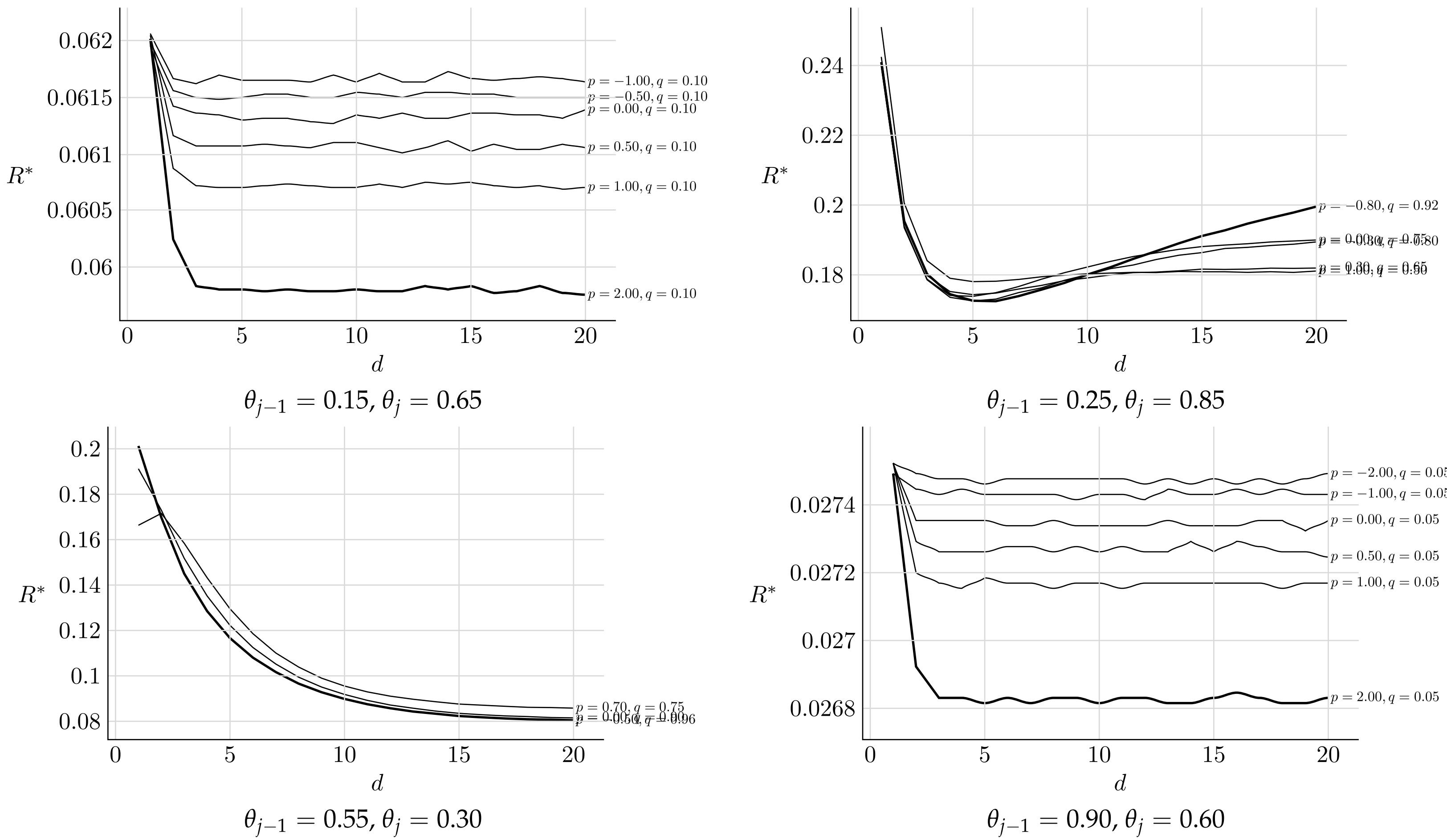

Figure D.6: Illustration of the properties of the weight function w_7 for $m = 20$

Appendix E

Detailed compression results for files of different sizes and similar contents

This appendix contains detailed results of compression ratios and speeds for parts of different sizes of standard data files.

File	Size [kB]	LZMA	ybs	bzip	PPMdH	MI64	DW
di_00008	8	3.773	3.438	3.465	3.219	3.136	3.403
di_00016	16	3.431	3.120	3.207	2.939	2.855	3.119
di_00032	32	3.174	2.843	2.961	2.700	2.615	2.867
di_00064	64	2.983	2.608	2.743	2.496	2.407	2.645
di_00128	128	2.791	2.383	2.526	2.297	2.204	2.427
di_00256	256	2.619	2.188	2.337	2.125	2.028	2.233
di_00512	512	2.459	2.014	2.164	1.971	1.872	2.058
di_01024	1024	2.366	1.934	2.157	1.992	1.821	1.978
di_02048	2048	2.346	1.927	2.202	1.997	1.828	1.971
di_04096	4096	2.288	1.856	2.183	1.922	1.753	1.897
di_08192	8192	2.232	1.794	2.173	1.864	1.712	1.832

Table E.1: Compression ratios (in bpc) for parts of dickens file of different sizes

File	Size [kB]	LZMA	ybs	bzip	PPMdH	MI64	DW
di_00008	8	200	500	571	500	129	333
di_00016	16	320	800	800	800	160	348
di_00032	32	390	1231	1067	1067	176	432
di_00064	64	478	1600	1067	1032	189	627
di_00128	128	460	1488	1164	1049	198	744
di_00256	256	409	1267	1164	941	204	837
di_00512	512	349	1004	839	911	211	839
di_01024	1024	309	853	812	908	193	846
di_02048	2048	275	775	802	867	186	815
di_04096	4096	241	697	793	821	188	794
di_08192	8192	213	622	788	774	188	756

Table E.2: Compression speeds (in kB/s) for parts of dickens file of different sizes

File	Size [kB]	LZMA	ybs	bzip	PPMdH	MI64	DW
os_00008	8	5.098	5.413	5.360	4.968	4.867	5.241
os_00016	16	4.239	4.656	4.612	4.120	4.016	4.496
os_00032	32	3.532	3.905	3.865	3.387	3.275	3.752
os_00064	64	3.035	3.315	3.233	3.381	2.704	3.114
os_00128	128	2.758	2.929	2.798	2.753	2.353	2.653
os_00256	256	2.580	2.663	2.489	2.447	2.135	2.358
os_00512	512	2.470	2.477	2.289	2.104	2.013	2.172
os_01024	1024	2.396	2.337	2.261	2.079	1.949	2.056
os_02048	2048	2.342	2.216	2.221	2.023	1.909	1.970
os_04096	4096	2.298	2.110	2.193	1.963	1.866	1.905
os_08192	8192	2.268	2.017	2.191	1.914	1.835	1.854

Table E.3: Compression ratios (in bpc) for parts of osdb file of different sizes

File	Size [kB]	LZMA	ybs	bzip	PPMdH	MI64	DW
os_00008	8	190	500	400	500	73	308
os_00016	16	296	800	571	667	88	286
os_00032	32	327	1231	800	941	108	421
os_00064	64	352	1455	1067	914	125	525
os_00128	128	344	1362	1067	1085	136	640
os_00256	256	327	1347	1067	1113	147	711
os_00512	512	300	1094	1004	962	156	716
os_01024	1024	277	895	930	890	144	715
os_02048	2048	251	802	925	870	143	696
os_04096	4096	228	730	909	784	144	666
os_08192	8192	209	651	916	702	143	635

Table E.4: Compression speeds (in kB/s) for parts of osdb file of different sizes

File	Size [kB]	LZMA	ybs	bzip	PPMdH	MI64	DW
sa_00008	8	2.966	2.664	2.674	2.463	2.380	2.621
sa_00016	16	2.850	2.576	2.611	2.397	2.305	2.561
sa_00032	32	2.649	2.427	2.505	2.261	2.174	2.438
sa_00064	64	2.547	2.324	2.434	2.317	2.087	2.349
sa_00128	128	2.525	2.708	2.673	2.442	2.254	2.591
sa_00256	256	4.786	5.074	4.956	4.834	4.622	4.846
sa_00512	512	3.243	3.529	3.441	3.272	3.089	3.357
sa_01024	1024	2.205	2.366	2.350	2.174	1.988	2.262
sa_02048	2048	1.766	1.835	1.930	1.730	1.552	1.788
sa_04096	4096	1.256	1.302	1.566	1.321	1.156	1.281
sa_08192	8192	2.131	2.281	2.466	2.126	2.042	2.205
sa_16384	16384	1.503	1.711	1.804	1.518	1.363	1.664

Table E.5: Compression ratios (in bpc) for parts of samba file of different sizes

File	Size [kB]	LZMA	ybs	bzip	PPMdH	MI64	DW
sa_00008	8	190	500	444	500	143	364
sa_00016	16	296	889	727	800	178	400
sa_00032	32	390	1455	1067	1143	198	516
sa_00064	64	478	1882	1067	1333	213	711
sa_00128	128	368	1600	1280	1422	183	736
sa_00256	256	456	1000	853	612	78	463
sa_00512	512	403	1133	1024	826	116	660
sa_01024	1024	334	1112	1077	1144	154	799
sa_02048	2048	303	934	1018	1223	168	796
sa_04096	4096	310	818	1008	1288	192	769
sa_08192	8192	329	718	929	833	137	614
sa_16384	16384	237	732	1001	1136	171	670

Table E.6: Compression speeds (in kB/s) for parts of samba file of different sizes

File	Size [kB]	LZMA	ybs	bzip	PPMdH	MI64	DW
xr_00008	8	4.730	5.430	5.176	5.340	4.499	5.019
xr_00016	16	4.601	5.087	4.857	5.111	4.401	4.745
xr_00032	32	4.472	4.695	4.520	4.822	4.285	4.428
xr_00064	64	4.406	4.406	4.306	4.610	4.219	4.215
xr_00128	128	4.366	4.195	4.160	4.454	4.151	4.053
xr_00256	256	4.181	3.903	3.920	4.131	3.913	3.789
xr_00512	512	4.058	3.668	3.729	3.866	3.715	3.571
xr_01024	1024	4.079	3.585	3.693	3.713	3.611	3.482
xr_02048	2048	4.113	3.542	3.675	3.683	3.550	3.451
xr_04096	4096	4.225	3.626	3.788	3.726	3.617	3.532
xr_08192	8192	4.227	3.608	3.804	3.685	3.587	3.521

Table E.7: Compression ratios (in bpc) for parts of x-ray file of different sizes

File	Size [kB]	LZMA	ybs	bzip	PPMdH	MI64	DW
xr_00008	8	222	444	286	364	68	286
xr_00016	16	308	667	471	500	82	250
xr_00032	32	471	889	800	640	95	390
xr_00064	64	640	1333	1032	744	101	464
xr_00128	128	727	1455	1153	753	104	566
xr_00256	256	731	1438	1158	744	118	610
xr_00512	512	638	1422	1067	716	130	644
xr_01024	1024	555	1247	1122	634	132	651
xr_02048	2048	472	1039	1063	602	129	620
xr_04096	4096	417	845	1001	479	113	582
xr_08192	8192	357	745	1008	419	110	563

Table E.8: Compression speeds (in kB/s) for parts of x-ray file of different sizes

List of Symbols and Abbreviations

Abbreviation	Description	Definition
\$	sentinel character—higher than the last in the alphabet	page 32
0-run	run consisting of zeros	page 34
a	the number of zeros in probability estimation	page 71
a_i	ith character of the alphabet	page 10
$\mathcal{A}$	alphabet of symbols in the input sequence	page 10
$\mathcal{A}^{\text{mtf}}$	alphabet of symbols in the sequence x^{mtf}	page 33
$\mathcal{A}^{\text{mtf,1}}$	alphabet of symbols $\{0, 1, 2\}$	page 52
$\mathcal{A}^{\text{rle-0}}$	alphabet of symbols in the sequence $x^{\text{rle-0}}$	page 35
b	the number of ones in probability estimation	page 71
BWCA	Burrows–Wheeler compression algorithm	page 31
BWT	Burrows–Wheeler transform	page 31
c-competitive	property of LUP solving algorithms	page 47
CT-component	component of the BWT output sequence related to one leaf in the context tree source	page 44
CT-source	context tree source	page 19
CTW	context tree weighting	page 27
DC	distance coder	page 51
DMC	dynamic Markov coder	page 26
Esc	escape code	page 23
FC	frequency count	page 81
FSM	finite-state machine	page 16
FSMX	kind of finite-state machine	page 18
$G(n)$	the number of non-BWT-equivalent sequences	page 67
H_j	the entropy of a memoryless source	page 71
IF	inversion frequencies transform	page 50
k	alphabet size	page 10
KT-estimator	Krichevsky–Trofimov estimator	page 74

Abbreviation	Description	Definition
L	list maintained by the algorithms solving the list update problem	page 33
l	number of symbols on the list L processed by the list update problem solving algorithm	page 47
l_b	the length of the buffer storing the past characters in the LZ77 algorithm	page 21
l_s	the maximum length of the identical subsequences in the LZ77 algorithm	page 21
LIPT	length index preserving transform	page 59
LOE	local order estimation	page 25
LUP	list update problem	page 46
LZ77	kind of Ziv–Lempel algorithms	page 21
LZ78	kind of Ziv–Lempel algorithms	page 22
LZFG	Ziv–Lempel–Fiala–Greene algorithm	page 22
LZRW	Ziv–Lempel–Williams algorithm	page 22
LZSS	Ziv–Lempel–Storer–Szymanski algorithm	page 21
LZW	Ziv–Lempel–Welch algorithm	page 23
$M(\cdot)$	matrix formed from the input sequence by rotating it	page 32
$\widetilde{M}(\cdot)$	lexicographically sorted matrix $M(\cdot)$	page 32
MTF	move-to-front transform	page 33
MTF-1	modified move-to-front transform	page 48
MTF-2	modified move-to-front transform	page 48
ω	finite memory CT-source	page 19
n	sequence length	page 10
$P_e(a,b)$	probability estimator	page 71
$P_0(j,a,b)$	probability that a memoryless source produce a sequence of a zeros and b ones	page 71
$P_c^d(a\|s_c^d)$	probability estimation method	page 88
$P_c^{d_1,d_2}(a\|s_c^{\max(d_1,d_2)})$	weighted probability estimation method	page 89
PPM	prediction by partial matching	page 23
PPMA	PPM algorithm with escape mechanism A	page 24
PPMB	PPM algorithm with escape mechanism B	page 24
PPMC	PPM algorithm with escape mechanism C	page 24
PPMD	PPM algorithm with escape mechanism D	page 25
PPME	PPM algorithm with escape mechanism E	page 25
PPMP	PPM algorithm with escape mechanism P	page 25
PPMX	PPM algorithm with escape mechanism X	page 25
PPMXC	PPM algorithm with escape mechanism XC	page 25
PPM*	PPM algorithm with unbounded context length	page 23
$R(\cdot)$	row number where in the matrix $\widetilde{M}(\cdot)$ the input sequence appears	page 32
$R_w^*(j,d)$	expected redundancy in probability estimation with weight functions	page 74
RLE	run length encoding	page 45

Abbreviation	Description	Definition
RLE-0	zero run length encoding	page 34
$\mathcal{S}$	set of states or contexts	page 17
s	state or context	page 19
s_c^d	last d bits encoded in the context c	page 88
SEE	secondary escape estimation	page 25
SBT	sort-by-time transform	page 82
σ	sequence of requests in the list update problem	page 47
Θ	set of parameters of the source	page 16
θ_j	probability of occurrence of bit 1 in the jth CT-component	page 71
θ_{j-1}	probability of occurrence of bit 1 in the $(j-1)$th CT-component	page 71
$t_c^d(a\|s_c^d)$	the number of occurrence of bit a in the context c, provided the last d coded bits were s_c^d	page 88
$t_{max}^{d_1}$	threshold for counters in arithmetic coding	page 91
$t_{max}^{d_2}$	threshold for counters in arithmetic coding	page 91
TS	time stamp algorithm	page 49
TS(0)	deterministic version of the time stamp algorithm	page 49
$U(n)$	the number of sequences that are not identical to cyclic shifts of themselves	page 67
$w(\cdot)$	weight function	page 72
$w_1(\cdot)$	weight function $w_1(\cdot)$	page 75
$w_2(\cdot)$	weight function $w_2(\cdot)$	page 75
$w_3(\cdot)$	weight function $w_3(\cdot)$	page 75
$w_4(\cdot)$	weight function $w_4(\cdot)$	page 75
$w_5(\cdot)$	weight function $w_5(\cdot)$	page 75
$w_6(\cdot)$	weight function $w_6(\cdot)$	page 75
$w_7(\cdot)$	weight function $w_7(\cdot)$	page 75
$w_8(\cdot)$	weight function $w_8(\cdot)$	page 84
$w_9(\cdot)$	weight function $w_9(\cdot)$	page 84
$W_i(a_j)$	sum assigned to the a_jth character by the WFC transform	page 81
WFC	weighted frequency count	page 81
x	input sequence	page 10
x^{-1}	reversed input sequence	page 10
$x_{i..j}$	subsequence starting from ith character to jth character	page 10
x^{bwt}	sequence after the Burrows–Wheeler transform	page 32
x^{mtf}	sequence after the move-to-front transform	page 33
$x^{\text{mtf,1}}$	part of the sequence x^{mtf}	page 52
$x^{\text{mtf,2}}$	part of the sequence x^{mtf}	page 52
x^{dc}	sequence after the distance coder transform	page 51
x^{if}	sequence after the inversion frequencies transform	page 50

Abbreviation	Description	Definition
x^{lup}	sequence after the algorithm solving the list update problem	page 47
$x^{\text{rle-0}}$	sequence after the zero run length encoding	page 35
$\mathcal{X}(s)$	set of positions where CT-components starts in the x^{bwt} sequence	page 43
ZL	Ziv–Lempel algorithms	page 20

List of Figures

List of Tables

Index