Improving Run Length Encoding (RLE) on bit level by preprocessing

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

Universität Trier
FB IV - Informatikwissenschaften
Lehrstuhl für Theoretische Informatik

Gutachter:	Prof. Dr. Henning Fernau
	Petra Wolf
Betreuer:	Prof. Dr. Henning Fernau & Petra Wolf

Vorgelegt am xx.xx.xxxx von:

Sven Fiergolla
Am Deimelberg 30
54295 Trier
sven.fiergolla@gmail.com
Matr.-Nr. 1252732

Abstract

Hier steht eine Kurzzusammenfassung (Abstract) der Arbeit. Stellen Sie kurz und präzise Ziel und Gegenstand der Arbeit, die angewendeten Methoden, sowie die Ergebnisse der Arbeit dar. Halten Sie dabei die ersten Punkten eher kurz und fokussieren Sie die Ergebnisse. Bewerten Sie auch die Ergebnissen und ordnen Sie diese in den Kontext ein.

Die Kurzzusammenfassung sollte maximal 1 Seite lang sein.

Contents

List of Figures

List of Tables

1. Introduction

1.1 Motivation

In the last decades, digital data transfer became available everywhere and to everyone. This rise of digital data urges the need for data compression techniques or improvements on existing ones. Run-length encoding [1] (abbreviated as RLE) is a simple coding scheme that performs lossless data compression. RLE compression simply represents the consecutive, identical symbols of a string with a run, usually denoted by σ^i, where σ is an alphabet symbol and i is its number of repetitions. To give an example, the string *aaaabbaaabbbba* can be compressed into RLE format as $a^4b^2a^3b^4a^1$. Thanks to its simplicity it is still being used in several areas like fax transmission, where RLE compression is combined with other techniques into Modified Huffman Coding [2]. Most fax documents are typically simple texts on a white background, RLE compression is particularly suitable for fax and often achieves good compression ratios.

1.2 Problem statement

Some strings like *aaaabbbb* achieve a very good compression rate because the string only has two different characters and they repeat more than twice. Therefore it can be compressed to a^4b^4 so from 8 byte down to 4 bytes if you encode it properly. On the other hand, if the input is highly mixed characters with few or no repetitions at all like *abcdefgh*, the run length encoding of the string is $a^1b^1c^1d^1e^1f^1g^1h^1$ which needs up to 16 bytes depending on the implementation. So the inherent problem with run length encoding is obviously the possible explosion in size, due to missing repetitions in the input string. Expanding the string to twice the original size is rather undesirable worst case behavior for a compression algorithm so one has to make sure the input data is fitted for RLE as compression scheme. One goal is to improve the compression ratio on data currently not suited for run length encoding and perform better than the originally proposed RLE, in order for it to work on arbitrary data. Another goal should be to minimize the increase in size in the worst case scenario.

1.3 Main Objective

The main objectives that derives from the problem statement, is to achieve an improved compression ratio compared to regular run length encoding on strings or files that are currently not suited for the method. Additionally it is desirable to further increase its performance in cases it is already reasonable. To unify the measurements, the compression ratio is calculated by encoding all files listed in the

Calgary corpus which will be presented in section 3.1. Since most improvements like permutations on the input, for example a reversible Burros-Wheeler-Transformation to increase the number of consecutive symbols or a different way of reading the byte stream take quite some time, encoding and decoding speed will decrease with increasing preprocessing effort.

1.4 Structure of this work

This work is structured into a first introduction into the basics of compression and the applied methods known to this discipline which will be used further on and an analysis of the current state of the art. Then, the conceptual design is depicted with a following analysis of the results. Afterwards, the implementation of the algorithms are described and the work as a whole is evaluated with a short closing discussion.

2. Principles of compression

Throughout this work a finite alphabet Σ is assumed with a linear order $\leq$ over its elements. A word w is a sequence $a_1, ..., a_n$ of letters $a_i \in \Sigma$, $1 \leq i \leq n$. The set of all such sequences is denoted by Σ^* which is the monoid over Σ, with concatenation as composition and with the empty word ε as neutral element. The set $\Sigma^+ = \Sigma^* \setminus \varepsilon$ consists of all non-empty words. Each word induces a labeling function $\lambda(w)$ as a labeled linear order where position i of w is labeled by $a_i \in \Sigma$ and we write $\lambda_w(i) = a_i$. If, for words u, v, $u = v$ or u is lexicographically smaller than v we write $u \leq v$ in the context of the order over the alphabet. We say that two words u, v are conjugate if $u = st$ and $v = ts$ for some words s and t and u and v are cyclic shifts of one another. The ordered conjugacy class of a word $w \in \Sigma^n$ denoted as $[w] = (w_q, ..., w_n)$ represents the lexicographically distinct ordered cyclic rotations of that word. A *Lyndon word* is the unique minimal element within its conjugacy class. More formally, let $[w] = (w, w_2, ...w_n)$, then $w \in \Sigma^+$ is a Lyndon word if $w < w_i$ for all $i \in \{2, ..., n\}$.

2.1 Compression and Encoding fundamentals

The basic idea of compression is to remove redundancy in data, since all non random data contains redundant information. Pattern or structure identification and exploitation enables storing the original data in less space. Compression can be broken down into two broad categories: Lossless and lossy compression. Lossless compression makes it possible to reproduce the original data exactly while lossy compression allows the some degradation in the encoded data to gain even higher compression at the cost of losing some of the original information. To understand compression, one first has to understand some basic principles of information theory like entropy and different approaches to compress different types of data with different encoding. We will also show the key differences between probability coding and dictionary coding.

2.1.1 Information Theory and Entropy

As Shannon described his analysis about the English language [3], he used the term entropy closely aligned with its definition in classical physics, where it is defined as the disorder of a system. Specifically speaking, it is assumed that a system has a set of states of which it can be in and it exists a probability distribution over those states. Shannon then defines the entropy as:

$$H(S) = \sum_{s \in S} P(s)\, i(s)$$

	$P(a)$	$P(b)$	$P(c)$	$P(d)$	$P(e)$	H
1.	0.2	0.2	0.2	0.2	0.2	2.322
2.	0.94	0.01	0.01	0.01	0.01	0.322

Table 2.1: Entropy in relation to the Probability of Symbols

where S describes all possible states, $P(s)$ is the likelihood of the system being in state s. So generally speaking it means that evenly distributed probabilities imply a higher entropy and vice versa. It also implies that, given a source of information S, its average information content per message from S is also described by this formula.

In addition to that, Shannon defined the term self information $i(s)$ as:

$$i(S) = log_2 \frac{1}{P(s)}$$

indicating that with higher probability of a state, less information can be contained. As an example, the statement "The criminal is smaller than 2 meters." is very likely but doesn't contain much information, whereas the statement "The criminal is larger than 2 meters." is not very likely but contains more information. With these definitions in mind, we can analyze some properties for the English language from an information theory point of view.

Combining those approaches, you will find that $\sum$ being a finite alphabet, and $P(s_i)$ describing the likelihood of s_i and $i(s_i)$ its self information, $i(s_i)$ also describes the amount of bits needed for this symbol, therefore its log_2.

$$H(\sum) = \sum_{i=1}^{n} P(s_i) \cdot log_2 \frac{1}{P(s_i)}$$

So the entropy also describes the theoretical amount of bits needed to persist a message from $\sum$, calculated in *bps* as bits per symbol.

TODO: describe some relations between probability and entropy of information source

2.1.2 General Analysis

To evaluate the efficiency of a specific compression technique, we have to determine how much information the raw data contains. We are looking at ASCII encoded English language sentences as examples for text compression. There have been broad analysis of ASCII entropy, consisting of 96 different printable symbols for the English language, generating approaches for calculating the entropy [4].

If we assume a 96 symbol alphabet and a uniform probability distribution, we find an entropy of $log_2(96) = 6.6\,bps$. An empirically distribution generated by text analysis, the entropy comes down to $4.5\,bps$. Using an encoding which encodes each symbol separately instead of the whole input at once, like the Huffman Coding, we can achieve an entropy of $4.7\,bps$ which is only slightly worse than the assumed entropy. By changing the assumption to blocks of symbols of length 8, we get 96^8 different blocks. Although the probability distribution of the English language implies an entropy as low as $1.3\,bps$ which leads to a possible reduction of symbols

to 3 printable characters without generating longer texts. Consulting newer sources we will find that, up to 500 character long symbol based text analysis, with around 20.3 $Mio.$ different symbols, results in an entropy of around $1,58\,bps$ [5]. This gives a vague limit of how much compression we can theoretically expect from English text.

2.1.3 Probabilistic Coding

The general idea behind Probability Coding is to analyze probabilities for messages and encode them in variable length bit strings according to their probability. This way, messages that are more likely and will repeat more often can be encoded in a smaller bit string representation. The generation of those probability distributions is considered part of the analyzer module by the algorithm and will be discussed later on in Section 4. Probability coding can be further distinguished into variable and fixed unique coding, which will represent each message with a bit string with an amount n of bits $n \in N$, for example Huffman coding as type of prefix coding which will be explained in greater detail in section 2.3. In contrast to Huffman coding there is also arithmetic coding which can represent a set of messages as a single floating point number q where $0.0 < q < 1.0$. By encoding them in nestend intervals in $[0, 1]$ determined by a probability distribution, they can "fuse" messages together and need less space to represent a set of messages than encoding every message separately. Arithmetic coding is usually stronger than Huffman coding but also slower. In 2014 a newer probabilistic approach was produced by Jarek Duda called Asymmetric Numeral Systems (ANS) [6]. This method encodes a string of symbols into a single natural number and combines the strength of arithmetic coding with the speed of Huffman coding.

2.1.4 Dictionary Coding

Dictionary Coding is best suited for data with a small amount of repeating patterns like a text containing repeated words. In this case it is very effective to save the patterns just once and refer to them if they are used later on in the text. If we know a lot about the text itself in advance, a static dictionary is sufficient but in general we want to compress an arbitrary text. This requires a more general approach as used by the well known LZ77 and LZ88 algorithms described by Jacob Ziv and Abraham Lempel [7]. They use a so called sliding window principle, a buffer moving across the input in the case of LZ77 or a dynamic dictionary in the implementation of LZ78. Both of them are still used in modified versions in real world applications as described in section 2.5, although mostly implemented in newer variants or derivatives.

The main difference between probabilistic coding and dictionary coding is, that the so far presented probability based methods like RLE or Huffman Coding are working on single characters or bytes whereas the dictionary based methods encode groups of characters of varying length. This is a clear advantage in terms of compression performance on every day data because of its repeating patterns most textual data has, as shown in Section 3.2, and has been overcome by sophisticated probabilistic methods like arithmetic coding or ASN.

2.1.5 Irreversible Compression

Irreversible Compression or also called "Lossy Compression" is a type of compression which looses information in the compression process and is therefore not completely

reversible, hence the name. There are a lot of use-cases for these type of compression algorithms, mostly used for images, video and audio files. These files typically contain information almost not perceptible for an average human like really small color differences between two pixels of an image or very high frequencies in an audio file. By using approximations to store the information and accept the loss of some information while retaining most of it, lossy image compression algorithms can get twice the compression performance compared to lossless algorithms. Due to the fact that most lossy compression techniques are not suited for text compression which is the main use case for this work, we will not elaborate any further on this topic.

2.2 Run Length Encoding

Run length coding might count as the simplest probability coding scheme that makes use of context and repeating occurrences as described in the Section 1.1. While the example made earlier was in textual representation, it is best suited and mostly used for pallet based bitmap images [8] such as fax transmissions or computer icons.

2.2.1 The History

The ITU-T T4 (Group 3) standard for Facsimile (fax) machines [9] is still in force for all devices used over regular phone lines. Each transmission sends a black and white image, where each pixel is called a *pel* and with a horizontal resolution of 8.05 $\frac{pels}{mm}$ and the vertical resolution depending on the mode. To encode each sequence of black and white pixels, the T4 standard uses RLE to encode each sequence of black and white pixels and since there are only two values, only the run length itself has to be encoded. It is assumed that the first run is always a white run, so there is a dummy white pel at the beginning of each sequence. For example, the sequence *bbbbwwbbbbb* can be encoded as 1,4,2,5 with the leading white dummy pixel.

To further reduce the RLE encoded sequence, a probability coding procedure like Huffman coding (explained in section 2.3) can be used to code the sequence, since shorter run lengths are generally more common than very long runs, which is also done by the T4 standard. Based on a broad analysis of the average run lengths counted, the T4 defined a set of static Huffman codes, using different codes for the black and white pixels. This way for example 20 consecutive white pels are a white run with length 20 an so the message can be encoded and transmitted as the code-word “0001000”, requiring only 7 bit instead of 20, seen in row 6 column 2 in table 2.2.1. Since a fax typically contains more white, the white runs are more frequent and thus get shorter code-words which is explained in greater detail in section 2.3. Since the appearance of text usually only needs a few consecutive black pels, short black runs of length 2 or 3 appear to be the most frequent seen runs so to save space they are encoded in the overall shortest codes. Runs of more than 64 have to be encoded in multiple Huffman codes, for example a run of 150 has to be encoded with the Huffman code of 128 followed by the code for 22.

By combining RLE with a probabilistic approach, the compression ratio increases because we no longer have to encode a run of length 1 or 2 with a fixed size of bits each instead we can write a recognizable Huffman code of varying length, which will be explored in grater detail later on as well as the decoding of the shown Huffman codes.

run length	white codeword	black codeword
0	00110101	0000110111
1	000111	010
2	0111	11
3	1000	10
4	1011	011
...		
20	0001000	00001101000
...		
64+	11011	0000001111
128+	10010	000011001000

Table 2.2: T4 static Huffman codes

2.2.2 Limitations

As mentioned in section 1.2, run length encoding is rarely used for regular text or continuous tone images because its potentially increase in size, due to non repetitive characters or bytes. A detailed analysis of this issue is performed in section 3.2. Several approaches to mitigate this issue have been implemented in this scope, some of the (e.g. Burrows-Wheeler-Transformation) in multiple variants.

2.2.3 Run Length Encoding today

While plain RLE is still in use by fax transmission or other highly specific tasks, it is mostly implemented in combination with other approaches. TODO: give more references [10]

2.3 Prefix Coding

Prefix codes make use of an untypical idea in computer science. Usually we deal with fixed length codes like 7 bit ASCII or 32 bit Integer representations which map each possible value into that fixed amount of bits. For compression it would be a benefit if we could write codes of variable length. The problem with these varying length codes is that as soon as they are part of a sequence, it becomes very hard to tell where one code word starts and finishes, resulting in ambiguous encoding. For example the set of codes $C = \{(a, 1), (b, 01), (c, 101), (d, 011)\}$ and the encoded message is 1011 do not generate distinct decodeable code words because there is no way to tell which message was encoded. From now on a code C for a set of messages S is considered to be in the form of $C = \{(s_1, w_1), (s_2, w_2), ..., (s_m, w_w)\}$.

One way to address this problem is by adding extra stop symbols or encoding a length before each code but those just add additional data. Another approach is to use so called prefix codes and make no code which is prefix of another code, which makes them distinct and uniquely decodable. A Huffman coding is one way of generating a prefix code, which is a uniquely decodable. When no code is prefix of another one, it is always decodable and yields a unique result because once a matching code is read, no other longer could also match. Huffman codes have another property, as they are call optimal prefix codes. To understand this property we first have to define the average length l_a of a code C as

$$l_a(C) = \sum_{(s,w)\in C} p(s)\, l(w)$$

We say that a prefix code is optimal, if $l_a(C)$ is minimized, so there is no other prefix code for a given probability distribution that has a lower average length. The existence of a relation between the average length of a prefix code to the entropy of a set of messages can be shown by making use of the Kraft McMillan Inequality [11]. For a uniquely decodeable code C where $l(w)$ is the length of the code word C

$$\sum_{(s,w)\in C} 2^{-l(w)} \leq 1$$

And it is also proven that the Huffman algorithm generates optimal prefix codes [12]. The algorithm in general is rather simple and the generation of the prefix codes will be demonstrated using an example. Assume we want to generate a prefix code for the message "accbccac". To generate a Huffman tree, first all occurrences of source symbols are counted and then a leaf node with its frequency is added to a priority queue for every symbol. Then, the two nodes with the lowest frequency are removed from the list, combined into one node with the two original as children, which is then added back to the list with the sum of their frequencies. This step is repeated while there are more than one nodes left in the queue. For our example the counts are a = 2, b = 1 and c = 5. So the first two nodes are b and a which are combined and become a new node, then this node is combined with c into another tree. This is resembled in figure 2.1 with the tree leafs in green and the two generated trees in red with the combined frequency. To get the actual mapping between a symbol and its code, simply follow the tree from its root to each symbol, the path from the root is its code. Obviously the most frequent symbol is assigned the shortest code and all codes are prefix free. the For decoding it is required to persists the mapping as well.

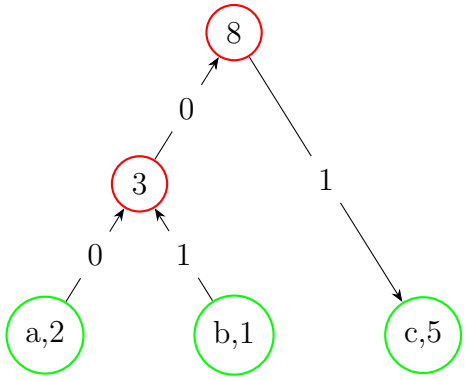

Figure 2.1: example Huffman Tree with 3 Leaf Nodes

Decoding is very trivial due to the prefix codes used b this algorithm. Reading bit by bit one can always see if a code is matching one of the mapping and if not, another bit is read. As soon as the parsed content matches a mapped code, we can decode it to the original symbol because there can not be a longer matching mapping. Due to the implementation via a Priority queue, which needs $O(log\, n)$ time per insertion and a tree with n leaves has $2n - 1$ nodes let this algorithm perform in $O(n\, log\, n)$ time where n is the number of symbols.

2.4 Other methods

There are other well known methods like Move-to-Front coding or arithmetic coding also applied in different compression algorithms with different methods and therefore

their own pros and cons. But other than Huffman coding, no further probability coding approach will be discussed nor implemented in this scope. But its worth mentioning that real world applications are mostly combinations of methods, like Deflate[13] which uses LZSS[14] (derivative of LZ77) and Huffman encoding or more recent ones like the PAQ family which is using a context mixing algorithm [15]. Some combinations and PAQ implementations can be found in table 2.5 where the state of the art compression methods and algorithms are discussed.

2.4.1 Burrows-Wheeler-Transformation

The Burrows-Wheeler-Transformation is not a compression technique but rather a method to prepare text to increase its compression potential, described by M. Burrows and D. J. Wheeler [16]. It is a Transformation of a string w of n characters by forming the n cyclic rotations of w, sorting them lexicographical, and extracting the last character of each of the rotations or in other words, the conjugacy class of w which is $[w]$. Each rotation is induced by a right shift of the original word. The right-shift of a word $w = a_1...a_n$ is dented as $r(w) = a_n a_1...a_{n-1}$ and the i-fold right shift $r^i(w)$ is defined inductively by $r^0(w) = w$ and $r^{i+1}(w) = r(r^i(w)))$ which is also defined for $i`n$ as $r^i(w) = r^j(w)$ where $j = i mod n$. The result of this transformation is a string L, consisting of these last characters of each sorted rotation. The basic algorithm also has to add special termination symbols or compute the index I of the original string S in the sorted list of rotations to be able to revert this transformation. Surprisingly, there is an efficient algorithm to invert the transformation back to the original string S given only L and the Index I [17] and also a modified version, which is not depended on additional information, described in section 2.4.3.

As an example of the creation of L, given the input string $w = crabab$, $n = 6$, and the alphabet $\Sigma = \{a, b, c, r\}$ with $a < b < c < r$. Create a $N \times N$ matrix M whose elements are characters, and with all cyclic rotations of w, which assembles the conjugacy class $[w]$ of word w. All rotations are sorted in lexicographical order, in this example, the index is $I = 4$ and the matrix M is:

row 1	a	b	a	b	c	r
row 2	a	b	c	r	a	b
row 3	b	a	b	c	r	a
row 4	b	c	r	a	b	a
row 5	c	r	a	b	a	b
row 6	r	a	b	a	b	c

Table 2.3: Burrows Wheeler Transformation Matrix (all cyclic rotations)

The resulting string $BWT(w) = L$ corresponds to the last column of M, with characters $M[0, n-1], \ldots, M[n-1, n-1]$. The output of the transformation is the pair (L, i), in the example, $L = rbaabc$ and $i = 5$. Obviously the string L contains consecutive identical characters, which results in better compressibility as shown in 3.3.1. Longer words with more identical characters in general tend to have more equal characters in series. Also this depicts that the word *ababcr* in row one of M is a Lyndon word, because it is the smallest in its conjugacy class and *crabab* is not.

2.4.2 Inverse Burrows-Wheeler-Transformation

The BWT is invertible and given (L, i) where i is the index of w in $M(w)$ one can reconstruct the word w. There is a formal proof [18] available for this as well as the bijective variant described in the next section. To abstract the idea of inverting the permutation using only L and i is described by the following process by carrying on with the example. We now reconstruct the original word from the pair (L, i) by first calculating the standard permutation of L by writing L column-wise, add positions, and then sort the pairs lexicographically. Mapping each index with its sorted neighbor results in the standard permutation.

word	word with position	sorted
r	(r,1)	(a,3)
b	(b,2)	(a,4)
a	(a,3)	(b,2)
a	(a,4)	(b,5)
b	(b,5)	(c,6)
c	(c,6)	(r,1)

Table 2.4: Standard permutation generation of the word L

This yields a standard permutation of:

$$\pi_L = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 2 & 5 & 6 & 1 \end{pmatrix}$$

Following $\pi_L^1(5)$ to $\pi_L^6(5)$ gives us a sequence of positions with $\pi_L(5) = 6$ as a beginning, described by:

$$6 \xrightarrow{\pi_L} 1 \xrightarrow{\pi_L} 3 \xrightarrow{\pi_L} 2 \xrightarrow{\pi_L} 4 \xrightarrow{\pi_L} 5$$

If this sequence is applied to the labeling function of the word L it results in:

$$\lambda_L(6)\,\lambda_L(1)\,\lambda_L(3)\,\lambda_L(2)\,\lambda_L(4)\,\lambda_L(5) = crabab = w$$

This shows that it is possible to reconstruct the word w given the transformation $BWT(w) = L$ and the index i.

2.4.3 Bijective Burrows-Wheeler-Scott Transformation

Although, the bijective variant (abbreviated with BWTS, sometimes BBWT) is mostly used further on it its rather complicated because it maps a word w with $l(w)$ to another word with equal length without any start or stop symbols and no additional index. It strongly relies on the Lyndon factorization, because calculating the BWT of a Lyndon word v always returns 1 as index since we know that v is always the first element in M. Then one can calculate the BWT over all Lyndon factors of the input data, without additional information needed, a proof for this is also mentioned in [18]. The bijective transform is generated by slitting the input into a sequence of Lyndon words. The Chen-Fox-Lyndon [19] theorem states that such a factorization exists can found in linear time [20]. Afterwards all rotations of all words are sorted as in the Burrows–Wheeler-Transformation which results in a

sequence of sorted strings. Again the final character of each string in this sorted list represents the transformation result. Decoding is also similar up until the last step of the inverse BWT. But instead of giving rotations of words, it instead gives rotations of Lyndon words , which are sorted into reverse order and then concatenated to get the original result.

2.5 State of the Art

The current state of the art at the time of writing is depicted in the table below, all algorithms used highest possible compression scheme available. In general there is always a balance between compressed size and compression or decompression speed, where the faster algorithms use mostly some form of dictionary coding sometimes in combination with a Huffman coder to be able to output variable length codes. The more advanced ones like PPMd or ZPAQ use complex context modeling or context mixing approaches where they generate a probability distribution for the next symbol based on just read symbols. In general these complex methods achieve better compression at the expense of using more space or time. The currently modern and most used algorithms have been executed with the Calgary corpus and the results are shown in table 2.5, all files were processed separately. Algorithms like *compress*, *gzip* and *ZIP* use a dictionary based method like LZ77 or some derivative and achieve reasonable compression at very fast rates. *bzip2* and *p7zip* use a Burrows-Wheeler-Transformation (described in 2.4.1) and then a combination of different techniques to improve performance. They sometimes even offer explicit choosing of a method, like *7zip* which achieves even better compression when using an advanced probabilistic method called Prediction by partial matching, also implemented but with even better results by *ZPAQ* which performs even better while still maintaining reasonable speeds. In the last decade methods like [zstandard] developed by Facebook or *brotli* developed by Google arose, mostly implementing ANS and other methods arose but they are either desired to be just faster or designed for a different task like *brotli* for HTML and JSON compression and do not achieved best results on this corpus. In the context of the hutter prize, a competitive compression challenge, other advanced probabilistic methods were developed like *paq8hp1* to *paq8hp12*, a series of algorithms by Alexander Rhatushnyak [21] who won the challenge. A more recent fork of *paq8* from 2014 is the current leader, *cmix*. It is still the best performing algorithm on this list and achieved by far the best results on the Calgary corpus at expense of almost 3 hours of computing and 32GB of needed ram, displayed in 6.2. Unfortunately it was not possible to run the paq8hp* variants due to an major bug in the only available releases. Most algorithms are available from the default repositories of your Linux distribution, or in the case of Fedora and CentOS, preinstalled with your operating system. The paq8hp* algorithms can be found on the Hutter price website and cmix is developed on github.

2.6 Limits of compression

All so far introduced methods and techniques work by removing redundant information but applied to random data without any structure or patterns they will fail to compress. This is explained by the Kolmogorov complexity[22] off a string s which is the shortest possible program with s as output. Consider the following two strings

method	options	size in bytes	compression	*bps*
uncompressed		3,145,718	100.0%	8.00
compress		1,250,382	40.4%	3.24
gzip v1.10	-9	1,021,720	32.4%	2.60
ZIP v3.0	-9	1,019,783	32.4%	2.59
zstandard 1.4.2	–ultra-23 -long=30	887,004	28.1%	2.25
bzip2 v1.0.8	–best	832,443	26.4%	2.11
brotli	-q 11 -w 24	826,638	26.3%	2.10
p7zip 16.02 (deflate)	a -mx10	821,873	26.1%	2.08
p7zip 16.02 (PPMd)	a -mm=ppmd o=32	763,067	24.2%	1.93
ZPAQ v7.15	-m5	659,700	20.9%	1.67
paq8hp*	-	-	-	-
cmix v18	-c -d	554,983	17.6%	1.41

Table 2.5: State of the Art compression ratios

of equal length *abababababab* and *4c1j5b20fg*. While the first string can simply be denoted as $5 \times ab$, the other one has obviously no simple representation hence it has a higher Kolmogorov complexity. Using the pigeonhole principle it is also possible to trivially prove this by assuming s is a binary string and $l(s) = n$. This implies 2^n possible string, all unequal. Assuming we can reduce all of them by 1 character, they only need $m = n - 1$ space and can only have 2^m possible unequal strings. This results in a contradiction because with lossless compression we need a distinct reverse operation and we cannot decode 2^{n-1} distinctly into 2^n strings. So to recap, no algorithm can compress all data of a given length, even by just one byte. If this was not true it would result in algorithm that could compress every string, so it could be applied recursively until any string has length 0. Therefore it is obvious that there exist some hard limits to compression.

3. Conceptual Design

The following chapter contains a detailed analysis of the problem and the resulting design decisions with some further requirements for the algorithm. To have a comparison to some extent, the initial performance analysis was performed on the Calgary corpus but the results of the unmodified run length coding algorithm were very underwhelming as expected.

3.1 The Calgary corpus

The Calgary Corpus [23] is a rather old corpus created by Ian Witten, Tim Bell and John Cleary from the University of Calgary in 1987. It consists of text and some binary data files shown in 3.1 and is still used for comparison between compression algorithms. After some objections were raised [24], it was mostly replaced by the Canterbury Corpus or similar ones but it is still useful for comparing against other compression algorithms. To validate the final results and to insure that the algorithm is not just suited for this corpus, it will be validated on the Canterbury Corpus as well.

file	size	description
bib	111261	ASCII text - 725 bibliographic references
book1	768771	unformatted ASCII text
book2	610856	ASCII text in UNIX “troff” format
geo	102400	32 bit numbers in IBM floating point format
news	377109	ASCII text - USENET batch file on a variety of topics
obj1	21504	VAX executable program
obj2	246814	Macintosh executable program
paper1	53161	UNIX “troff” format
paper2	82199	UNIX “troff” format
pic	513216	1728 x 2376 bitmap image
progc	39611	Source code in C
progl	71646	Source code in Lisp
progp	49379	Source code in Pascal
trans	93695	ASCII and control characters

Table 3.1: The Calgary Corpus

3.2 Initial Findings

The algorithm itself works in a very simple manner. Reading each bit of the input data in a consecutive way, count the consecutive bits and write the count instead

of the bits themselves. So the input of “00011100” would resolve to two counts of length 3 and one of length 2 if we also assume a starting zero. Encoding this can be done by “11 | 11 | 10”. We do not need stop symbols of any kind if we assume a fixed size for the count of each run so we can easily decode this back by making the same assumptions and reading always 2 bit and start with a zero. This implies setting a maximum run length during encoding, limited by the amount of bits used to encode one single count of the input data. If the run is starting with a 1 or a count exceeds this maximum run length, we need to add a pseudo run of length zero, so for example the input of “11000000” would be encoded to “00|10|11|00|11”, corresponding to zero times zero at the beginning, two ones, three zeros, then zero times one then three more zeros. The longer the consecutive runs, the better it can be stored, expecting they do not exceed the maximum run length but on the other hand, a high maximum run length needs more bits per run which implies more overhead if the runs to save are rather short. So in general we assume an improvement when the average run length is close to the maximum run length and not often exceeding it.

Originally developed and used on black and white pallet images containing only two values often in large repetitions, we want to use it for rather arbitrary data, mostly text. The initial algorithm is not suited for that because continuous text as binary representation does not contain runs of any kind, which could be compressed. The ASCII representation of the letter ’e’ which is the most common in general English, has the value 130 or ’01100101’ as 7-bit ASCII or ’001100101’ as byte value of the UTF-8 representation. As you can see, there are no runs of a considerable size and this is the case for most printable characters as they all have a value between 32 and 127 (or 255 for the extended ASCII). Applied to the Calgary Corpus in this simple implementation, there should be an increase in size expected or *negative compression* as one might say.

With rather low expected average run length in general data it was still unclear which amount of bits per run are most suited for the mix of data residing in the corpus, so between 2 and 8 bits per run were tried and the results are shown in table 3.2. They depict the anticipated, an increase in size regardless of the amount of bits used to encode a run. By using 8 bits to encode a single run, in the worst case scenario a byte which is only alternating values like 01010101, would expand to 8 bytes, all encoding a run of length 1. For this reason, many implementations combine RLE with other encoding schemes like Huffman encoding to be able to encode runs with variable length, which will be discussed later on.

bits per rle number	expansion ratio %	*bps*
8	329	26.38
7	288	23.11
6	248	19.87
5	208	16.66
4	168	13.51
3	131	10.50
2	104	8.36

Table 3.2: Binary RLE on the Calgary Corpus

RLE is also applicable on a byte level, because there should be repetitions of any kind like consecutive letters or line endings (EOL). This modified byte level RLE

encodes runs of identical byte values, ignoring individual bits and word boundaries. The most common byte level RLE scheme encodes runs of bytes into 2-byte packets. The first byte contains the run count of 1 to 256, and the second byte contains the value of the byte run. If a run exceeds a count of 256, it has to be encoded twice, one with count 256 and one with any further runs. So for example the word "aaabbbb" will be encoded to "0x02 | 0x61 | 0x03 | 0x62". We do not need runs of length zero because longer runs just have to be encoded more than once so we can use all 256 possible byte values as a count. Using 8 bit for one run is obviously exaggerated because in arbitrary text it is rather rare that a character repeats more that twice. So different sizes of maximum run lengths were tried and the results are shown below.

bits per rle number	ratio in %	*bps*
8	165	13.20
7	154	12.38
6	144	11.57
5	134	10.77
4	125	10.00
3	116	9.29
2	109	8.74

Table 3.3: Byte-wise RLE on the Calgary Corpus

However after some analysis of the corpus data, it was shown that most runs had a value of one and almost no runs larger than 4 occurred, which lead to the conclusion, two bit for the run count should be plenty. But even with a run size of just two bits, there is still an increase in size of about 9% and uses 8.74 *bps*. This is still useful as a kind of a base line. Interestingly the binary implementation performs better on 2 bits per RLE number (4 % increase in size) than the byte implementation (9 % increase in size) but also worse with a higher amount of bits per run, where it expands the data to more than triple in size. It is unclear which kind of implementation will profit most of preprocessing, so both will be further analyzed, but the benefit of byte wise RLE is the better worst case performance of 1.5 up to 2 times the original size compared to binary RLE with up to 4.5 times the original size.

If we take a more detailed look, we can see that while most files expand with larger RLE numbers regardless which implementation of RLE, but some files have their minimum size when encoded with higher RLE numbers of up to 7 bit. With the simple binary based RLE, almost all files of the Calgary Corpus expand linear related to the amount of bits used for the encoding however the file *pic* decreases in size until 7 bits per RLE number used to a sizes of just 19.5% of its original size with only 1.56 *bps* while the other files just doubled or even tripled in size. Using the byte wise operating RLE we see a similar result with the file *pic*, but not as decent with 27.2% of its original size using 2.17 *bps* encoding with 6 bits per run. Now it is quite clear why run length encoding is very suited for monochromatic images where it achieves a compression ratio close to the theoretical expected maximum compression because the file mostly consists of long runs of repeating bytes depicting the same color value.

file	size original	$\frac{bits}{RLEnumber}$	size encoded	ratio in %	*bps*
pic	513216	2	350292	68.25	5.46
		3	235067	45.80	3.66
		4	165745	32.29	2.58
		5	126349	24.61	1.96
		6	106773	20.80	1.66
		7	100098	19.50	1.56
		8	101014	19.68	1.57

Table 3.4: The file *pic* with increasing bits per RLE encoded number

An additional step of the improvement could be the detection of high efficiency with regular binary RLE to simply apply this to files highly suited for this method.

3.3 Possible Improvements by Preprocessing

The broad idea of preprocessing is to manipulate the input data in a way that results in data which can be compressed more efficiently than the original data. This can be done in various ways, some of them will be explored in greater detail to find out if it is implementable or not. One way of doing so is a Burrows-Wheeler-Transformation.

3.3.1 Burrows-Wheeler-Transformation

To understand how a Burrows-Wheeler-Transformation improves the effectiveness of compression, consider the effect in a common word in English text. Examine the letter 't' in the word 'the', in an input string holding multiple instances of this word. Sorting all rotations of a string results in all rotations starting with 'he ' will be sorted together and most of them are going to to end in the letter 't'. This implies that the transformed string L has a large number of the letter t, combined with some other characters, such as space, 's', 'T', and 'S'. This is true for all characters, so any substring of L is likely to contain a large number of some distinct characters. "The overall effect is that the probability that given character *c* will occur at a given point in L is very high if *c* occurs near that point in L, and is low otherwise" [16].

It is obvious that this should always improve the performance of byte level RLE because the transformation is taking place at character level but it should not effect the binary implementations.

3.3.2 Vertical byte reading

Instead of performing compute intense operation on the data, we could also interpret the data in a different way and apply the original run length encoding on binary data. This idea is also know for binary RLE on images, where the encoding in the image follows a specific path.
TODO showcase

By reading the data in chunks of a fixed size, it is possible to read all most significant bits of all bytes, then the second most significant bits of all bytes and so on. This interpretation results in longer runs as shown in the example below.

The binary UTF-8 interpretation of the example string from earlier S = ‘abraca’ results in 8 runs of length 1, 9 runs of length 2 as well as 3 runs of length 3 and 4.

48 ELEMENTS:

Reading the data in a different way, all most significant bits, then al second most significant bits and so forth, results in much longer runs. This becomes clear if we read each row in the example below.

2-D:

Now we have 5 runs of length 6, 2 runs of length 3, 3 runs of length 2 and just 6 runs of length 1 as opposed by the simple interpretation. This is because the binary similarity between the used characters, as the character for a and b only differ in one bit. It is clear that simply a different way of reading the input does not compress the actual data, instead it enables a better application of existing compressions.

3.3.3 Byte remapping

The effect of very long runs in the last example was mainly because the binary representations of the used characters are very similar, so the range of byte values used was very small (between a = 97 and r = 114). Introducing other used symbols like uppercase letters, space or new lines, the used range expands.

The binary representation of a String like S’ = ”Lorem ipsum dolor sit amet, consectetur adipiscing elit.”, results in a worse result as shown below. The usage of other characters expanded the used byte range to between 32 and 117 which results in shorter average runs. Interestingly the most significant bit is always 0, a fragment from the backwards compatibility with standard ASCII encoding.

2-D:

One idea to solve the shorter runs might be a dynamic byte remapping, as the input data is read in parts, where the most frequently used bytes are mapped to the lowest value. This way the values are not alternating in the whole range of 0 to 255 but rather in a smaller subset and the most frequent ones will be the smallest values, so in theory our average runs should increase because we should encounter more consecutive zeros. Some sections have more specific characters or bytes than others but this idea can also be applied to the whole file however it is unclear at this point what method outperforms which. A single map for each block of data should result in lower average values used but also creates a kind of overhead because the mapping has to be stored in the encoded file as well. Applying a simple mapping to lower values results in the following horizontally interpreted rows.

2-D:

0 0 0 0 0 0 0 1 0 0	<Row 20>
0 0 0 0 0 0 0 1 1 1	<Row 21>
0 0 0 0 0 0 0 0 0 0	<Row 22>
0 0 0 0 0 1 0 0 1 0	<Row 23>
0 0 0 0 0 0 0 0 0 1	<Row 24>
0 0 0 0 0 0 0 0 1 1	<Row 25>
0 0 0 0 0 0 0 1 1 1	<Row 26>
0 0 0 0 0 1 0 0 0 0	<Row 27>

Using this method, the 4 most significant bits all result in zero columns, even row 5 has long runs while it is worth noting that the mapping itself has to be persisted in the encoded file as well. It is also still unclear if this idea scales well or is applicable to other files.

3.3.4 Combined approaches

The idea of combining different compression methods into a superior method is not new and was also performed on RLE as mentioned in Section 2.2.1. While the idea of encoding the RLE numbers with Huffman codes is already known and analyzed, it is mostly in a static sense and optimized for special purpose applications. However the vertical byte reading enables new approaches, even more in combination with the idea of byte remapping and might become applicable to more than just binary Fax transmissions or DNA sequencing **TODO: reference** with longer runs of any kind in average.

It might be interesting to see how well the appliance performs using the vertical binary encoding, in combination with the byte mapping. We expect the more significant the bit, the longer the runs because alternating values should be mostly on the lower significance bits. Using larger run length numbers for these rows while using smaller RLE numbers or even another encoding scheme like simple Huffman encoding we should improve our initial results.

3.3.5 Huffman encoding of the RLE runs

Another interesting approach is the improvement achieved by the combination of different methods like performing this modified RLE run on arbitrary data and then encode the results with Huffman codes, using shorter codes for more frequent runs. This idea is not new and also used in the current Fax Transmission Protocol but only for the simple binary RLE in combination with modified Huffman Codes. Other papers also mentioned these combined approaches and seemed to achieved good compression ratios, not much worse than the theoretical limit of around 1.5 *bps* shown in Section 2.1. This was for example done by M. Burrows and D.J. Wheeler in 1994 with their Transformation, in combination with a Move-to-Front Coder and a Huffman Coder [16]. Encoding the Calgary Corpus resulted in a decrease in size to just 27% of its original with a mean *bps* of just 2.43. This approach would no longer be considered preprocessing, but if more compression could be achieved by adding a post-processing step it is still worth trying out. It clearly has some benefits over the encoding of regular RLE numbers with a fixed size because we can encode with varying lengths, which also implies the absence of additional zeros, needed in the initial implementations.

3.4 Summary

In summary there seems to be a lot of possibilities to improve Run-length encoding and it should be feasible to implement them in a reasonable amount of time. Different preprocessing steps will probably generate longer average runs and therefore hopefully achieve an overall compression instead of very good compression on just highly specific files like pallet based images. But it is not clear which step will influence the results the most so all of them will be tried out, probably even in combination with one-another and then we can draw conclusions.

4. Analysis

As Section 3 showed, there are some potentially promising improvements to be made to Run Length encoding, but how well they scale and work on a larger input with versatile symbols or bytes has to be determined.

4.1 Vertical Byte Reading

First implementation required an input of chunks with a size divisible by 8 for the vertical reading of the input shown in Section 3.3.2. Otherwise parsing it into an array of bytes results in the last byte having some padding which might cause problems later on. By using the library explained in section 5.6.1 we avoid this completely by directly working on a stream of either bytes or bits. This is especially useful for working with larger files. Nonetheless both ways of handling the input, collecting all bits of identical significance and processing on the fly, are implemented.

4.1.1 Compression improvements trough vertical reading

First results of just plain binary RLE on the vertical interpretation improved its overall performance and achieved a small edge over regular binary RLE with a slightly smaller expansion but it is still not as good as byte wise RLE with a small run value of 2 bits.

bits per rle number	ratio in %	*bps*
8	255.22	20.41
7	224.45	17.95
6	194.74	15.57
5	167.04	13.36
4	142.58	11.40
3	127.80	10.22
2	139.79	11.18

Table 4.1: Binary RLE on vertical interpreted data

If we take a closer look on each file, we see similar results compared to the originally proposed binary RLE, where most files had a compression ratio of above 1 with 3 bits per RLE encoded number, except for the file *pic*. Average sizes are increasing again with more bits per run up to 2.5 times its original size and also the file *pic* has the best compression ratio. This time, it is at its peak using 6 bits per RLE run although only a compression of 3.67 *bps* is achieved, compared to 1.56 *bps* with simple binary RLE. These results were quite far from the desired outcome which

mainly arose because increasing the bits per run improved the result for the most significant bits but also degraded the results for the other bits. A combination of encoding the bits of different significance with different RLE schemes with varying bits per run could solve this issue.

4.2 Varying maximum run lengths

Most significant bits have been encoded with more bits per run than the other ones and after bench marking every combination, it turned out, with 2 bits per run and 5 bits per run for the 3 most significant bits, it improved by another 4 percent with most files having a *bps* ratio of only slightly above 9. More specifically some textual files are close to 8, which relates to the earlier mentioned ASCII fragments in UTF-8 encoding. This time higher possible runs on this position resulted in fewer runs in total, so in a better compression overall. Applying this increase to more than the most significant bit lead to a decrease in performance, which most likely related to the shorter runs on lower order bits, seen in Section 3.3.2. Therefore this idea was applied again but with a much finer granularity and every combination of different run lengths of every bit could be tried out. For reference, this mapping can be described by a vector v_i with 8 components $v_1, v_2, ..., v_8$ each with values greater 1. Each component corresponds to the bits per run stored for the bit number i of each byte.

By measuring some combinations we can selectively make small changes to improve further, all results depicted in 4.1. This way the results were improved and the compression result was lowered around 7 percent points to 112.41% of its original size instead of 127% with a fix length for every bits position as shown in figure 4.2, but this was still far from the desired state. Most files still increased in size except the file *pic*, even though all bits could be encoded differently but the average run length of the lower bits must be very low if 2 bits per RLE number achieved best results. To increase the average run length overall, the already described byte mapping was applied to the input data.

4.3 Byte remapping

As shown in Section 3.3.3 this effect could become useful if it resulted in higher average runs. To apply the mapping, the individual file has to be analyzed at first to find the occurrence of each byte in the input data. We expect a map with the most occurring byte is assigned the lowest value or 0 as byte, the second most frequently read byte the subsequent value which should be 1 and so on. Then while reading the file during steps, every byte read will be mapped. If a small mapping is generated (e.g. 62 entries), we know all bytes to encode will have zeros only on the first and second most significant bit value because the highest value after the mapping took place is 61. This should also be taken into account later on during encoding and finding the optimal maximum run lengths. To reverse the mapping after decoding the runs, the mapping has to be persisted into the encoded file and parsed back during the decoding process. This is done by adding the length of the mapping and then just all mapping keys to the header of the output file.

The effect was also seen in the second and third most significant bit, as higher value bytes became unlikely in the input data after the remapping so the idea of

file	size original	size encoded	ratio in %	*bps*
bib	111261	129424	116.32	9.31
book1	768771	820463	106.72	8.54
book2	610856	659811	108.01	8.64
geo	102400	162274	158.47	12.68
news	377109	400810	106.28	8.50
obj1	21504	31592	146.91	11.75
obj2	246814	379591	153.80	12.30
paper1	53161	57654	108.45	8.68
paper2	82199	88121	107.20	8.58
pic	513216	533254	103.90	8.31
progc	39611	41360	104.42	8.35
progl	71646	74554	104.06	8.32
progp	49379	53403	108.15	8.65
trans	93695	99818	106.54	8.52
all files	3145718	3536225	112.41	8.99

Table 4.2: Calgary Corpus encoded, vertical encoding, using bits per run: (2, 2, 2, 2, 3, 4, 3, 7)

varying maximum run lengths for different significant bits became more appealing again. To determine the best combination of maximum run lengths and how many most significant bits should be encoded with a higher maximum run length, most promising looking combinations of these were tested and the results plotted below.

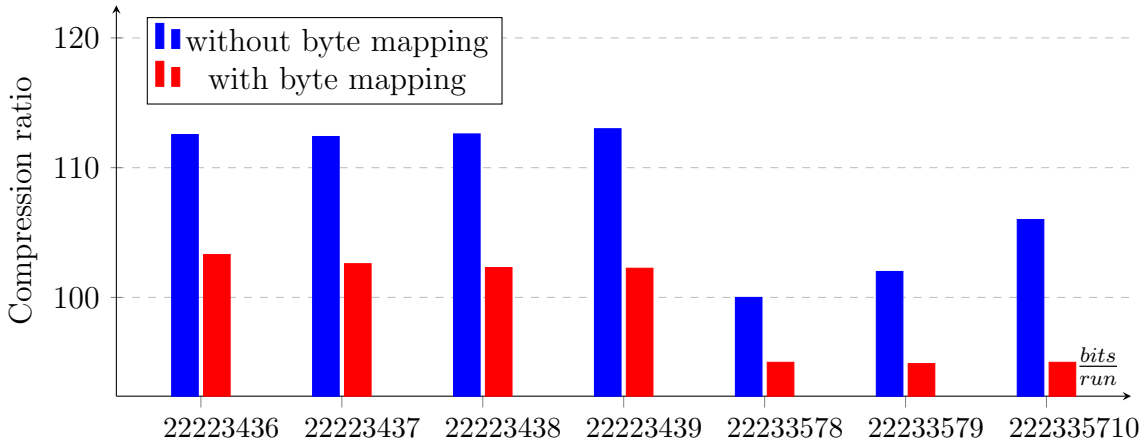

Figure 4.1: Byte mapping and varying maximum run lengths

Using this combined approach a real compression was achieved for the corpus instead of just one very specific file. The combination of 2 bits for the most insignificant bits, 3 for the fourth and fifth most insignificant bit 5 for the sixths most one, 7 bits for the second most significant bit and 9 bits for the most significant one yielded the overall best results with 94.9% of its original size and 7.59 *bps* as shown in Figure 4.1. Some files got a little smaller while other files still expanded which is only somewhat of an enhancement over regular RLE which performs really well on specific files.

But on this corpus a slight reduction in size was achieved using preprocessing and a modified RLE.

file	size original	size encoded	ratio in %	*bps*
bib	111261	111579	100.29	8.02
book1	768771	669578	87.10	6.97
book2	610856	551757	90.33	7.23
geo	102400	144974	141.58	11.33
news	377109	363010	96.26	7.70
obj1	21504	30166	140.28	11.22
obj2	246814	340165	137.82	11.03
paper1	53161	50074	94.19	7.54
paper2	82199	71747	87.28	6.98
pic	513216	408136	79.53	6.36
progc	39611	38490	97.17	7.77
progl	71646	63765	89.00	7.12
progp	49379	46093	93.35	7.47
trans	93695	94729	101.10	8.09
all files	3145718	2988359	94.99	7.59

Table 4.3: Calgary Corpus encoded with vertical reading, byte remapping, using bits per run (2, 2, 3, 3, 3, 4, 5, 8)

4.4 Applying the Burrows-Wheeler-Transformation

Another possible preprocessing step which promised an improvement is the mentioned Burrows-Wheeler-Transformation from Section 2.4.1, applied to all RLE implementations. Initially very simple transformation implementation was chosen, working by adding additional start and stop symbols to the input string (0x02 as STX, start of text and 0x03 as ETX, end of text). Some basic testing and playing around worked great but later on it revealed some major issues. For example the Calgary Corpus consists of more than text, in fact the files geo, obj1, obj2 and pic contain some binary data including the symbols STX or ETX, so we wont be able to apply the transformation to these. Another shortcoming was the very poor time complexity of almost $O(n^2)$ because under the hood, it uses a dual pivot Quicksort algorithm from the JDK 11, which is typically faster than traditional one pivot Quick-sort. This algorithm offers $\Theta(n\ log(n))$ average time complexity but in the worst case, its time complexity is still cubic. This problem was partially solved by reading the input data in parts and performing the transformation on each part, result in a much smaller length n and thus better run time at the expense of a slightly worse transformation result. As all chunks are individual transformations, they can also be computed in parallel.

bits per rle number	ratio in %	*bps*
3	95.41	7.63
2	91.39	7.31

Table 4.4: Initial BWT implementation on byte wise RLE

While it was only applicable to textual data and very slow, even when divided into smaller parts and computed in parallel, it improved the overall results of byte wise RLE by 16% to a compression ratio of slightly over 7 *bps* as table 4.4 depicts, which seemed like a good start. Regular binary RLE did not really benefit from this transformation as expected but on vertical interpretation, consecutive characters result in successive bits on every significance. Still this implementation had to be dropped and switched against one that could handle arbitrary input to be able to transform all files. This time all files could be processed and the resulting compression with byte wise RLE improved further as shown in table 4.5.

bits per rle number	ratio in %	*bps*
3	91.62	7.33
2	89.46	7.15

Table 4.5: Burrows Wheeler Transformation on byte wise RLE

In general a Burrows-Wheeler-Transformation should also increase the runs in the implementation of Section 3.3.2 and 3.3.3 so those preprocessing steps were also applied in combination. To do so, it was first swapped against an sufficient implementation provided by a paper from M. Burrows and D. J. Wheeler [16] from 1994. Their method is also the one described in Section 3.3.1 and could handle arbitrary input but it also had some downsides like the additional index I of the transformation, which had to be persisted as well. The major downside of this implementation is the at least quadratic time complexity which made it still rather slow with increasing sizes of chunks, so again the input had to be spliced into small parts. If chunks exceeded a length of more than one kilobyte it became unacceptably slow even though it strongly improved the transformation results so most of the time and in table 4.5 and ref4.6 the transformation was performed on chunks of size 512 byte. To overcome this degradation of the original algorithm and the necessity of saving additional indices, the implementation had to be swapped once more against one that was first described by [17] in 2009 which claimed to perform in linear time complexity.

In form of the C library libdivsufsort a working implementation of BWTS was found, the bijective Burrows-Wheeler-Scott-Transformation described in [25]. This kind of Burrows-Wheeler-Transformation does not require additional information, no start and stop symbols neither an index of its original position. Briefly, it does not construct a matrix of all cyclic rotations, instead it is computed with a suffix array sorted with DivSufSort [26], closer described in the paper [27], which is the fastest currently known method of constructing the transformation. To use it properly the code was ported to Kotlin but there are also ports of this library in Java and Go available which are recommended because the original code is neither documented nor readable and the functionality can easily be used via a dependency.

The simple binary RLE did not really benefit from this transformation which has to be expected, because it generates repetitions of bytes, but if a byte needs many short runs it will still expand in size because the average runs are not that much influenced. For example just the letter “e” needs 6 different runs, no matter on which position or surrounded by what letters. The byte wise implementation on the other hand did very strongly benefit from this transformation, it will always create longer runs of

equal bytes. Swapping the implementation of the Burrows-Wheeler-Transformation resulted in way better results, the simple byte wise RLE archives compression ratios around 59 % of its original size while using 4 bits per run, see table 4.6. This was mainly because of the longer repetitions possible after the transformation was performed on the whole input instead of small chunks. Interestingly average runs of characters increased so much, that 4 bits per run achieved the maximum result. Another reason for this vast improvement compared to the old implementations is the lack of additional information needed to store because we do no longer need to store the transformation index of every chunk or additional characters.

bits per rle number	ratio in %	*bps*
8	74.42	5.95
7	69.90	5.59
6	65.58	5.24
5	61.71	4.93
4	58.98	4.71
3	59.18	4.73
2	67.69	5.41

Table 4.6: Modified Burrows Wheeler Transformation on byte wise RLE

Working on the whole input data and no longer on small chunks, this BWTS generates extreme long runs of identical byte values, which in turn enhances the performance of the byte wise RLE vastly. If we take a closer look we can see in table 4.7 that all files have a compression ratio below 100 while the total size is nearly reduced to half. The file *geo* is still close to uncompressed but half of the files only need less than 5 *bps*. The file *pic* is still the best compressible with only 2.12 *bps*.

file	size original	size encoded	compression	*bps*
bib	111261	59285	53.28	4.26
book1	768771	590879	76.86	6.15
book2	610856	374742	61.35	4.91
geo	102400	101192	98.82	7.91
news	377109	246047	65.25	5.22
obj1	21504	16467	76.58	6.13
obj2	246814	126626	51.30	4.10
paper1	53161	34130	64.20	5.14
paper2	82199	56507	68.74	5.50
pic	513216	136074	26.51	2.12
progc	39611	24312	61.38	4.91
progl	71646	31466	43.92	3.51
progp	49379	20862	42.25	3.38
trans	93695	32835	35.04	2.80
all files	3145718	1855520	58.98	4.71

Table 4.7: Calgary Corpus encoded with byte wise RLE after a Burrows-Wheeler-Transformation with 4 bit per run

There should still be room for some optimizations because as seen in section 4.2, the remapping of the input resulted in longer runs on the higher order bits and

the vertical interpretation made it possible to encode different sections with different maximum run lengths. It was expected that applying the Burrows-Wheeler-Transformation to the vertical encoding variant should improve its efficiency as much as the byte wise RLE did benefit, which turned out to be wrong. The vertical interpretation did indeed perform at its best when applying the BWTS and the mapping but it did not outperform the combination of BWTS and byte wise RLE which is kind of expected because the transformation creates repetitions on byte level.

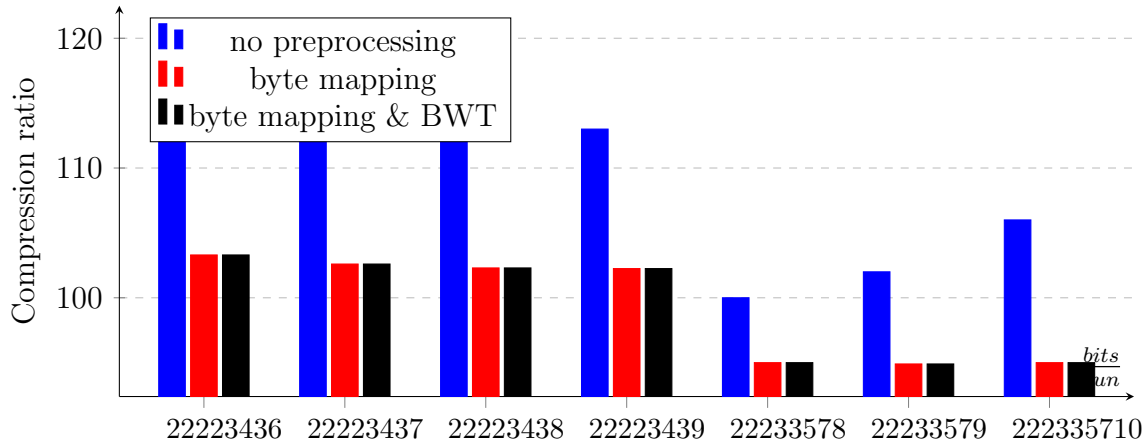

Figure 4.2: Byte mapping and varying maximum run lengths

file	size original	size encoded	ratio in %	*bps*
bib	111261	73843	66.37	5.31
book1	768771	570348	74.19	5.94
book2	610856	409639	67.06	5.36
geo	102400	145950	142.53	11.40
news	377109	275396	73.03	5.84
obj1	21504	27023	125.66	10.05
obj2	246814	213392	86.46	6.92
paper1	53161	37344	70.25	5.62
paper2	82199	56490	68.72	5.50
pic	513216	227914	44.41	3.55
progc	39611	28275	71.38	5.71
progl	71646	38144	53.24	4.26
progp	49379	27029	54.74	4.38
trans	93695	49314	52.63	4.21
all files	3145718	2184197	69.43	5.55

Table 4.8: Calgary Corpus encoded, all preprocessing steps, using bits per run (4, 4, 4, 4, 5, 7, 8, 10)

One last option was the encoding of the lowest significant bits with another, more suited scheme like Huffman encoding was tried out but with rather poor results. It was found that encoding the last or the last few rows seen in 3.3.2 with did not improve overall results and it was therefore discarded. This might be related to the high improvement in RLE after a Burrows-Wheeler-Transformation and other factors like additional overhead because the mapping of the Huffman encoding has

to be persisted with the encoded data. But the idea of combing the RLE methods with Huffman encoding still stuck around and was picked up again later on in a modified way in Section 4.5. No further attempts to add or improve preprocessing steps of this kind were made, instead the results were analyzed and compared with other results in section 6.

4.5 Huffman encoding of the RLE runs

Encoding produced run lengths using Huffman codes was implemented, similar to the Fax Transmission Standard mentioned in section 2.2.1, but in a dynamic way instead of predefined static codes. This idea was also used by Burrows and Wheeler in their paper [16] but instead of RLE they combined a Move to Front Coder with their transformation and then encoded the result using Huffman codes. This way it would be possible to encode more frequent results of the Move-to-Front Encoder or the Run Length Encoder could be encoded into shorter codes and thus save even more space. This step is not really considered preprocessing anymore but could improve the compression furthermore, however the benefit of Huffman encoding is the possibility to output codes of varying length as described in section 2.3. Combining the binary or vertical encoded RLE with a Huffman encoder would be a huge benefit, because as of now the algorithm needs to output a number of bits for each number to encode. Combined, the more frequent short runs of 1 or 2 can be encoded into shorter Huffman codes which should increase the overall compression algorithm. In general it was assumed that there was no longer a benefit in using different maximum run lengths on bits of different significance, because a run of length is going to be encoded with a Huffman code, not with just a fixed amount of bits. To reverse the Huffman encoding, the Huffman tree has to be persisted into the file as well.

file	size original	size encoded	ratio in %	*bps*
bib	111261	44156	39.69	3.17
book1	768771	340279	44.26	3.54
book2	610856	243092	39.80	3.18
geo	102400	63006	61.53	4.92
news	377109	173207	45.93	3.67
obj1	21504	14405	66.99	5.36
obj2	246814	119957	48.60	3.89
paper1	53161	24917	46.87	3.75
paper2	82199	35939	43.72	3.50
pic	513216	82136	16.00	1.28
progc	39611	18890	47.69	3.82
progl	71646	24649	34.40	2.75
progp	49379	17416	35.27	2.82
trans	93695	31235	33.34	2.67
all files	3145718	1237380	39.33	3.14

Table 4.9: Calgary Corpus encoded with vertical reading, byte mapping and a BWTS, using Huffman encoding for all counted runs, 8 bit per run

This combination of vertical encoding with byte remapping and the sophisticated BWTS as preprocessing steps so far best results have been found. Although, at

first glance it was suggested to eliminate the maximum run restriction to possibly compress all bits of one significance into a single Huffman code, it turned out to be most efficient if the run length process is limited to 8 bits per run. This way still a total of 255 bits of same significance can be encoded into one during the RLE step, but there will be only a maximum of 256 different Huffman codes generated. Average Huffman code length will be significantly shorter. Also it makes no sense to use different maximum run lengths for bits of different significance, because there will still be one Huffman code for each different run value.

4.6 Summary

Using a composition of preprocessing steps, another data interpretation in form of the vertical reading and optimal prefix codes as encoding due to the Huffman encoder, acceptable results have been achieved. Not only was there a reasonable compression for every file, the file *pic* was compressed even better than before, although it is highly suited for the original proposed RLE.

5. Implementation

All algorithms described have been implemented using Kotlin, because it can be compiled for the Java Virtual Machine as well as native, so it seemed like a good balance between a native implementation and higher language conciseness and fault-tolerance. Also there were some libraries available for byte and bit operations on streams which proved to be quite useful, although there was almost no documentation available. This one and all other libraries are described in section 5.6. The main focus is on the encoder and decoder classes but the other modules will be discussed as well. The project is realized as a maven project, to simplify dependency management.

5.1 Binary and byte wise RLE

The simple binary and byte wise RLE are rather trivial and implemented together in one encoder, the *StringRunLengthEncoder*. The binary version is implemented with the mentioned BitStream from the IOStreams library. It allows working on a stream and reading and writing bit by bit and also reading the next n bit as signed or unsigned number which comes in handy during decoding. In general, it is called with a variable b bitsPerRun which sets the used bits to store one run. At first a maximum run length is determined by the maximum value b can store as a binary string, $l(b) = n$ implies a maximum value of 2^{n-1}. Then the input is read consecutively in bits and the runs of equal bits are counted. We are always assuming the run starts with zero, if this is not the case a leading run of 0 is added, which means there are zero times 0 at the beginning. If a run exceeds the maximum run length, the maximum is written and again an artificial zero is added to the output stream to signal a length higher than the maximum. Each run can simply be written to the output stream with the desired amount of bits per run. During decoding, we assume the same b and can then always read n bits of the stream as unsigned value, know each run again and can therefore reconstruct the original data.

Byte wise RLE is working with a similar idea of counting n runs of equal information. This time it is applied on a byte level, reading byte after byte. If the next byte is the same as the current one, the counter is incremented, if not the run and the byte value are encoded as pair $(n, byte)$ to the output. The byte value itself still needs 8 bit of information but the run does not. Most raw untreated data does not contain long runs of consecutive identical byte values average run length is rather small. This implied storing a count of 1 or 2 in 8 bits of space which in turn explained the expansion in size seen in table 3.3. Therefore this version was also implemented with an arbitrary amount of bits to save per run, to minimize the overhead. If a run exceeds the maximum, which is again determined by the amount of bits stored per run, it is encoded twice, once with the maximum count and once with the remaining

count. We do not need a zero run in this version because we also store the value itself, therefore we can count without a zero. This means for an example run of 4 times the value *0xFF* and 4 bits per run saved, it will be encoded as the pair $(0011, 0xFF)$ or 001111111111 as consecutive binary stream.

5.2 Vertical binary RLE

Basically the ideas described in sections 3.3.2 and 4.1 oppose only a small variance compared to regular binary RLE. It is realized with the use of BitStreams again. Its stream interface offers a position p, which corresponds to the byte value and a offset o, which is a bit value with significance o of byte p, which allows reading all bits of the same significance in order. This was done for significance zero to seven to read all bits in a vertical manner as in the examples. Then each run is again counted with the same method including a maximum run length defined by the amount of bits used to count a run and the runs are then written with the fixed amount of bits to the output stream. Afterwards, the amount of runs per bit position is written to the tail of the encoded file, without the information how many runs are expected it would be much more difficult to decide which run belongs where, but it is still possible. The average overhead of this additional information is around 34 byte, two for each count and a two byte stop symbol which is only needed seven times, no additional stop at the end of the file. Even though it was originally designed to work on chunks of bytes, in the end the transformations worked on the file or on the stream itself, which was significantly faster.

During decoding, the expected amount of runs are parsed from the end of the file. Then the actual decoding happens, with the fixed n bits per code for this bits significance. Knowing the exact amount of RLE numbers for each bit position makes it easy to decode, because the variable length of encoded numbers can be chosen accordingly while reading the stream once. Assuming a starting run on zero, all runs are written back to one file, each bit position sequentially, to then assemble the original data. This is done for each bit position in sequential order so we need to write 8 times to the output file. It might be worth trying out building the byte stream in memory and the write the output only once, which might be faster but also requires more internal data structures and holding the whole file in memory at once.

5.3 Byte Remapping

To start of with the preprocessing, the byte remapping was implemented. The *Analyzer* is responsible for generating a overall probability distribution over the values of bytes contained in the file. This serves as a input for the map generation, where every byte value is sorted accordingly to its occurrence and mapped to increasing byte values, so the most frequent byte to *0x00*, the second most often to *0x01* and so on. Afterwards, a temporary file is generated where each byte from the original file is mapped, which allows streaming during the encoding process. Decoding requires access to the original mapping, therefore it is persisted at the start of the encoded file. To do so, we only need to know the number of mapped values and then the original byte, not the mapped value since we know they are sorted acceding.

5.4 Burrows Wheeler Transformation

As mentioned earlier, the Burrows-Wheeler-Transformation is implemented in 3 different versions, starting of with the naive vs. unsophisticated. The *transformation.BurrowsWheelerTransformation* is implemented with the use of start and stop symbols (0x02 as STX and 0x03 as ETX) and with the creating and sorting of all cyclic rotations of the input string. This can be done for all text input files but not for binary data because files containing the start or stop symbols confused the algorithm and made the inverse transformation impossible. Additionally it is extremely slow due to its at least quadratic complexity, even when working on small chunks which messes up the overall transformation result. It is not further described as it was only used for some initial testing to see if and how much RLE benefits from this transformation.

The second implementation is realized by following the original algorithm description provided in greater detail in the paper by M. Burrows and D. J. Wheeler [16] (algorithm C and D). The *transformation.BurrowsWheelerTransformationModified* works on parts of the data so the transformation result is still strongly depending on the size of the chunks, but it could at least handle arbitrary input. Higher chunk sizes greatly increased the transformation because more equal characters are in the same chunk but also really slowed down the process. Due to fact that both the mapping and the modified transformation work on an array of bytes and do not interfere with one-another, they can be performed in any order. The further on used advanced implementation of the bijective Burrows-Wheeler-Scott-Transformation is ported from Java from an external source or directly usable as dependency and therefore described in section 5.6.

5.5 Huffman encoding

Following the pseudo-code provided by M. Liśkiewicz and H. Fernau in [4] on page 21 the implementation was straight forward. Internally a small set of data structures are provided for assembling the Huffman tree, a *HuffmanTree*, a *HuffmanNode* and a *HuffmanLeaf* class is implemented. The HuffmanTree is abstract and only holds the frequency of a tree, since every tree itself has its own frequency. It also implements a compareTo function, to draw comparisons between different trees. A HuffmanNode extends the HuffmanTree, consists of a left and right HuffmanTree and has their sum of frequencies as frequency. The leaf is itself also a tree and holds a value of type byte and a frequency which resembles its occurrence. To build the Huffman tree for a given set of bytes, the algorithm expects an array of integer I, assembling the occurrences o of bytes $b \in [0, 255]$ in the form of $I[b] = o$. Then a leaf is created for every entry of I with a frequency of o and collected into a PriorityQueue of HuffmanTrees. Then while there are still at least two trees left in the queue, the two lowest frequencies are removed from the queue, merged into a single tree and reinserted into the queue. After the creation of the tree, all paths are followed and every time a leaf is reached, the current path is added to a map in form of a StringBuffer b. This buffer contains the path of left and right trees traversed into the original one, adding a zero for every left descent and a one for every right one. Finally the mapping of type byte to StringBuffer is returned. To apply this algorithm to the runs of the RLE encoding, the same is done except that instead

byte values, every possible run length value is counted and collected into the same structure.

To reverse this Huffman coding it is required to have access to the performed mapping, otherwise decoding would be impossible. Therefore the map itself is written to the beginning of the file, similar to the mapping from section 5.3. This time though we need triplets of values, because the Huffman code can have a variable length, so the mapping is encoded to $(b, l(b), b)$, with b and $l(b)$ each assuming one byte. This also implies a maximum length for Huffman codes of 255. During decoding of the mapping, first the total number of mappings is parsed. Then while there are still more mappings expected, we parse one byte which is the mapping value, then one byte which contains the length of the following mapping as unsigned byte value and then the Huffman code of the given length is parsed bit by bit and saved as a StringBuffer. This way it is possible to write and read the variable length codes continuously from the stream.

5.6 External libraries

Some external libraries are used in throughout this project. Most of them provide a set of sealed functionality, like *io.github.jupf.staticlog* which just facilitate the logging features and are not further interesting, therefore most of them are mention in section 5.6.4, only extensively used ones are described in greater detail.

5.6.1 IOStreams for Kotlin

Alexander Kornilov created and released this library, which was found in the Kotlin forum and is currently hosted on Sourceforge. It has a very light documentation [28] and is released with an Apache v2.0 license. It has to be mentioned that for the time being there is only a pre-release available, this version 0.33 is used throughout this project. Moving to this library rendered it possible to work entirely on streams of data as well as reading the next n bit(s). This greatly improved the performance of the initial algorithm and reduced required data structures and memory. There was also some odd or unexpected behavior seen which might be changed in further versions, for example if the stream is currently at the first byte and we want to write to its bit with the highest significance. This represents a so called bit offset of 7, so we can write the position as 0:7. Writing a 1 or setting this bit to true advances the position of the stream to the next byte and offset zero, so 1:0 while writing a 0 keeps the stream at its current position so it is still at 0:7 and the next bit gets written on the same position. Basically the interface provides good functionality with the drawback of some inconsistencies.

5.6.2 libDivSufSort

The original code for the modified BWTS algorithm and the necessary sorting algorithms called *DivSufSort* for efficient suffix array sorting was provided by Yuta Mori in the library libDivSufSort [26] and is as already mentioned, closer described in the paper by Johannes Fischer and Florian Kurpicz [27]. It runs in $O(nlogn)$ worst-case time using only $5n + O(1)$ bytes of memory space, where n is the length of the input. The code is available under the MIT license at Github but written in C and thus, rather hard to use from Kotlin. Further research lead to a port to Java which uses the same structures and methods but is already close to the desired state, since Kotlin enables using Java classes by default.

5.6.3 libDivSufSort in Java

Porting the Java implementation provided by kanzi [29], a collection of state of the art compression methods, all available in Java, C++ and Go, was easy but also gratuitous, since there are releases available for the Java version which is still maintained at a high frequency. To use them we simply add the dependency to our maven project. This library basically provides one functionality, the implementation of a sophisticated bijective Burrows-Wheeler-Scott-Transformation in linear time.

To skim over its functionality, it provides a clean API to work with advanced compression algorithms in Java, most noticeable *DivSufSort* and *BWTS*. The BWTS class offers just the two directions of the transformation as methods, both working on arrays of bytes, which requires reading all the input into memory. This functionality is encapsulated by the class *BWTSWrapper* to work on file level and generate a temporary file with the transformed contents to further work on a stream of that file. At first the whole input is spliced into Lyndon words. Then the suffix array is generated and sorted, using the fastest known suffix sorting algorithm DivSufSort. The result is also written to a temporary file, like the mapping which was quite useful for debugging, and enables streaming its contents during encoding.

5.6.4 Others

Besides logging, some other basic functionality was added which was quite simple through the use of maven. *org.junit.jupiter* provided the packages *junit*, *engine* and *api* in version 5.6.0, which allows a simple test unit creation and execution. Additionally *assert-j* in version 3.13.2 added extended assertion capabilities to the encoder as well as to the test cases. A modest approach of multithreading was enabled by use of Kotlin *coroutines* in version 1.3.2, provided by Jetbrains. Just by convenience, Googles *guava* was used in version 28.2 for new collection types and a comparator for an array of bytes.

5.7 Implementation Evaluation

Obviously the assembled tool does not compete with the state of the art methods used today, neither in comparison of their compression results nor in terms of speed. There is most likely still a lot of unused potential to speed things up, for example by excessive multithreading. Also, decoding could be vastly speed up by writing each output byte only once instead of up to 8 times if the byte is of value 0xFF. Even compression results could probably be further improved but more on that in chapter 7. Nonetheless, the desired concept was proven and the results show a clear advantage over regular RLE achieved through preprocessing.

5.8 Usage

To enable a convenient usage, the algorithm is obtainable as jar file but it can also be built from sources. It provides a simple command line interface which expects a desired action, either compressing *-c* or decompressing *-d* and a method, either vertical RLE *-v*, binary *-bin* or byte wise RLE *-byte*. Binary and byte wise RLE can optionally be called with a parameter N where N is the amount of bits used to encode a single RLE number, vertical encoding can also run with an arbitrary

amount of bits, but expects 8 comma separated numbers *-v N,N,N,N,N,N,N,N*. As preprocessing options, mapping *-map*, applying the sophisticated Burrows-Wheeler-Transformation *-bwt* and Huffman encoding *-huf* are available via the parameters. Additional information can be acquired by launching the application with *-h* for further help. To enable debug logging and get detailed insight into the compression and decompression steps, the parameter *-D* has to be set.

6. Evaluation

Moving on to the evaluation, we start by comparing the achieved results using different preprocessing options or combination of those. To start of it is clear that the best compression ratio was not accomplished by using a combination of different mapping techniques and a vertical interpretation of the input data but we now take a closer look at the discrepancy between each result. It is also clear that with the help of such methods, run length encoding can become a suitable compression algorithm for more than just pellet based images although it is clearly not as sophisticated as advanced state of the art compression methods mentioned in section 2.5.

6.1 Functional Evaluation

The functional evaluation was performed in two steps. The first question is weather the algorithm works and no data is lost during the process of encoding. The decoder shows that this is the case and all information can be reconstructed, hence the suggested algorithm works as intended. The next question was, if it is just a method suited for this corpus and as it turned out, it performed even better on the newer and more frequently suggested Canterbury corpus than it did on the Calgary corpus.

file	size original	size encoded	ratio in %	*bps*
alice29.txt	152089	65445	43.03	3.44
asyoulik.txt	125179	59291	47.36	3.79
cp.html	24603	11073	45.01	3.60
fields.c	11150	5183	46.48	3.72
grammar.lsp	3721	1923	51.68	4.13
kennedy.xls	1029744	229823	22.32	1.79
lcet10.txt	426754	170593	39.97	3.20
plrabn12.txt	481861	215628	44.75	3.58
ptt5	513216	82136	16.01	1.28
sum	38240	19616	51.30	4.10
xargs.1	4227	2515	59.50	4.76
all files	2814880	867322	30.81	2.46

Table 6.1: Canterbury encoded, all preprocessing steps, using Huffman encoding for all counted runs

The two desired goals could also be achieved, a significant compression for the whole corpus is possible while even improving against regular RLE on files highly suited for this method and the results are pretty similar on both corpora.

6.2 Benchmarks

To benchmark the proposed algorithms, it was compared against the state of the art on the same hardware. All benchmarks were performed on an AMD Ryzen 5 2600X six core processor (12 threads) with a 3.6 GHz base clock and a 4.2 GHz boost clock speed. For memory, 16GB 3200MHz ram and a Samsung evo ssd was used for persistent storage. The algorithm cmix explicitly demanded 32 GB of ram which could not be provided, which in turn lead to a lot of paging and thus, very poor timing benchmark results.

method	size in bytes	compression	*bps*	time	
				encoding	decoding
uncompressed	3,145,718	100.0%	8.00		
compress	1,250,382	40.4%	3.24	0.039s	0.025s
modified vertical RLE	1,237,380	39.3%	3.14	6.840s	15.637s
gzip v1.10	1,021,720	32.4%	2.60	0.232s	0.025s
ZIP v3.0	1,019,783	32.4%	2.59	0.214s	0.022s
zstandard 1.4.2	887,004	28.1%	2.25	0.951s	0.011s
bzip2 v1.0.8	832,443	26.4%	2.11	0.191s	0.088s
brotli	826,638	26.3%	2.10	4.609s	0.015s
p7zip 16.02 (deflate)	794,098	26.1%	2.08	0.431s	0.045s
p7zip 16.02 (PPMd)	763,067	24.2%	1.93	0.345s	0.282s
ZPAQ v7.15	659.700	20.9%	1.67	7.452s	7.735s
paq8hp*	-	-	-	-	-
cmix v18	554,983	17.6%	1.41	>3h	>2h

Table 6.2: Benchmark on the Calgary Corpus

Avoiding internal operations and large or complex data structures to hold all the input data or even collecting the values of same significance in memory into byte arrays greatly improved time performance of the algorithm described. Encoding is reasonable fast with measured 6.8 seconds but the decoding is rather slow with 15.6 seconds although is has to be mentioned that there is still some potential in performance optimization and parallelization.

6.3 Conclusion

In conclusion, the desired state was achieved and it was shown that with the help of preprocessing and a different encoding technique, RLE can achieve compression results comparable to modern methods. It is still far from opposing a real competition, neither in speed nor compression ratios, but it is only a few percent points behind daily used algorithms. Possible additional improvements will be discussed in the next section, although at this point one has to decide if compression or speed should be in the main focus. Anyhow, to show the efficiency of the suggested preprocessing methods it is definitely adequate.

7. Discussion

- use other probability approach
- use huffman for byte wise rel and bwt

Bibliography

[1] A. H. Robinson and C. Cherry, eds., *Results of a prototype television bandwidth compression scheme, Proceedings of the IEEE 3*, vol. 55, March 1967.

[2] R. Hunter and A. H. Robinson, eds., *International digital facsimile coding standards, Proceedings of the IEEE 68*, 1980.

[3] C. E. Shannon, *Prediction and entropy of printed English*, vol. 30. Institute of Electrical and Electronics Engineers, Jan 1951.

[4] M. Liśkiewicz and H. Fernau, *Datenkompression*. Institut fur Theoretische Informatik, Medizinische Universit zu Lübeck, 2013.

[5] F. G. Guerrero, ed., *A new look at the classical entropy of written English*, 2009.

[6] J. Duda, "Asymmetric numeral systems as close to capacity low state entropy coders," *CoRR*, vol. abs/1311.2540, 2013.

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

[8] M. Dallwitz, "An introduction to computer images," *TDWG Newsletter*, vol. 7, no. 10, p. 2, 1992.

[9] I. T. U. (ITU), *T.4 : Standardization of Group 3 facsimile terminals for document transmission*, 2004.

[10] P. Lehotay-Kéry and A. Kiss, "Genpress: A novel dictionary based method to compress dna data of various species," in *Intelligent Information and Database Systems* (N. T. Nguyen, F. L. Gaol, T.-P. Hong, and B. Trawiński, eds.), (Cham), pp. 385–394, Springer International Publishing, 2019.

[11] B. McMillan, "Two inequalities implied by unique decipherability," *IRE Transactions on Information Theory*, vol. 2, pp. 115–116, December 1956.

[12] G. E. blelloch, *Introduction to Data Compression*. Computer Science Department Carnegie Mellon University, 2013.

[13] P. Deutsch, "Rfc1951: Deflate compressed data format specification version 1.3," 1996.

[14] J. A. Storer and T. G. Szymanski, "Data compression via textual substitution," *J. ACM*, vol. 29, p. 928–951, Oct. 1982.

[15] M. Mahoney, "Data compression programs," 2009.

[16] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," tech. rep., Systems Research Center, 1994.

[17] D. Okanohara and K. Sadakane, "A linear-time burrows-wheeler transform using induced sorting," in *String Processing and Information Retrieval* (J. Karlgren, J. Tarhio, and H. Hyyrö, eds.), pp. 90–101, Springer Berlin Heidelberg, 2009.

[18] M. Kufleitner, "On bijective variants of the burrows-wheeler transform," *arXiv preprint arXiv:0908.0239*, 2009.

[19] M. Lothaire, *Combinatorics on words. Foreword by Roger Lyndon. 2nd ed.*, vol. 17. Cambridge: Cambridge University Press, 2nd ed. ed., 1997.

[20] J. P. Duval, "Factorizing words over an ordered alphabet.," *J. Algorithms*, vol. 4, pp. 363–381, 1983.

[21] M. Mahoney, "Large text compression benchmark," *URL: http://www.mattmahoney.net/text/text.html*, 2011.

[22] A. N. Kolmogorov, *On tables of random numbers.* MR, 1963.

[23] I. Witten, T. Bell, and J. Cleary, "Calgary corpus," *University of Calgary, Canada*, 1987.

[24] R. Arnold and T. Bell, "A corpus for the evaluation of lossless compression algorithms," in *Proceedings DCC '97. Data Compression Conference*, pp. 201–210, March 1997.

[25] J. Y. Gil and D. A. Scott, "A bijective string sorting transform," *CoRR*, vol. abs/1201.3077, 2012.

[26] Y. Mori, *LibDivSufSort, suffix sorting algorithm in C*, 2015.

[27] J. Fischer and F. Kurpicz, "Dismantling divsufsort," *CoRR*, vol. abs/1710.01896, 2017.

[28] A. Kornilov, *IOStreams for Kotlin, library*, 2019.

[29] flanglet, *Kanzi, State-of-the-art lossless data compression in Java*, 2019.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich diese Bachelor-/Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe. Die Arbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vor-gelegt. Sie wurde bisher auch nicht veröffentlicht.

Trier, den xx. Monat 20xx