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 Informatik I

Gutachter:	Prof. Dr. Ingo J. Timm
	xxxxxxxxxx
Betreuer:	xxxxxxxxxx

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

Die Einleitung besteht aus der Motivation, der Problemstellung, der Zielsetzung und einem erster Überblick über den Aufbau der Arbeit.
TODO:
- explain compression ratio
- define unit of compression to quantify question and results

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 [9] (abbreviated as RLE) is a simple coding schemes that performs lossless data compression. RLE compression simply represents the consecutive, identical symbols of a string with a run, usually denoted by $\sigma\ i$, 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 $a^4b^2a^3b^4a^1$. Its simplicity and efficiency make run-length encoding still usable in several areas like fax transmission, where RLE compression is combined with other techniques into Modified Huffman Coding [5]. Most fax documents are typically simple texts on a white background, RLE compression is particularly suitable for fax and often achieves good compression ratios. Another appliance of RLE is optical character recognition, in which the inputs are usually images of large scales of identically valued pixels [1].

1.2 Problem statement

Some strings like aaaabbbb archive 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 abababab, the run length encoding of the string is $a^1b^1c^1d^1e^1f^1g^hb^1i^1j^k$ 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 not really a good compression 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 suited for run length encoding and perform better than the originally proposed RLE. Another goal should be to minimize the increase in size in the worst case scenario.

1.3 Main Objective

Was ist das Ziel der Arbeit. Wie soll das Problem gelöst werden?

- bessere kompressionsrate im vergleich zu konventionellem RLE
- gleiche oder ähnliche decoding zeit ?
- ansatz beschreiben

The main objectives that derives from the problem statement is to archive an improved compression ratio compared to regular run length encoding. To unify the measurements, the compression ratio is calculated by encoding all files listed in the Galgary corpus and then normalize the results.

Since most improvements like permutations on the input, for example a revertable Burros-Wheeler-Transformation to increase the number of consecutive symbols or a different way of reading the byte stream take quite some time, encoding speed will increase. A second objective might be to keep decoding speed close to the original run length encoding.

1.4 Structure of this work

Was enthalten die weiteren Kapitel? Wie ist die Arbeit aufgebaut? Welche Methodik wird verfolgt?

- describe following structure
- use references
- try to keep idea of a recurrent theme

2. Principles of compression

The basic idea of compression is to remove redundancy in data. 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 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 Compression and Encoding fundamentals

TBD
...

2.1.1 Information Theory and Entropy

As Shannon described his analysis about the English language [10], 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 it can be in and it exists a probability distribution over those states. Shannon then defines the Entropy as:

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

where S describes all possible States, $P(s)$ is the likelihood of $s \in S$. 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 the higher the 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 lots of information. With these definitions in mind, we can analyze some properties for the English language from an information theory perspective 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: fix caption

	$P(a)$	$P(b)$	$P(c)$	$P(d)$	$P(e)$	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

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. In this case for textual compression at first, we are talking about the English language. There have been broad analysis of ASCII Entropy, consisting of 96 different printable symbols for the English language, generating approaches for calculating the entropy [7].

If we assume a 96 Symbol alphabet and a uniform probability distribution, we have a quite high entropy of $log_2(96) = 6,6\,bps$. In a empirically distribution generated by text analysis, the entropy comes down to $4.5\,bps$. Using an encoding with separated encoding per Symbol like the Huffman Coding, we can archive an entropy of $4.7\,bps$ which is only slightly worse that the theoretical entropy. By changing the assumption to blocks of symbols of length 8, we get 96^8 different blocks. With a probability distribution close to the English language we get an entropy as low as $1.3\,bps$ which implies 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* [4]. This gives a clear limit of how much compression we can theoretically expect from English text. Using a combination of different approaches like Huffman Coding and Run Length Coding, and encoding different bits of a single symbol with different approaches, we do no longer encode every symbol separately, which might result in a better compression.

2.1.3 Probabilistic Coding

The general idea behind Probability Coding is to analyze probabilities for messages and the encode them in 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 (chap: design, chap: impl). Probability coding compression can be discerned into fixed unique coding, which will represent each message with a bit string with the amount of bits being an integer, for example Huffman coding as type of prefix coding. In contrast to Huffman coding there are also arithmetic codes which can represent a message with a floating point number of bits in the corresponding bit string. By doing so they can "fuse" messages together and need less space to represent a set of messages. ...

2.1.4 Dictionary Coding

Dictionary Coding is best suited for Data with a small amount of repeating patterns like a text representing specific words. In this case it is very effective to save the patterns just once and refer to them if the 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 [11]. They use a so called sliding window principle, a buffer moving across the data to encode 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.

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 character or byte 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 3.1.

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 Coding

Run length coding might count as the simplest 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 [3] such as fax transmissions or computer icons.

2.2.1 The history

The ITU-T T4 (Group 3) standard for Facsimile (fax) machines [6] 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 can be used do code the sequence, since shorter runs like lengths are generally more common than very long runs, which is also done by the T4 standard. Based on a broad analysis of these run lengths, the T4 defined a set of static Huffman codes, using a different codes for the black and white pixels.

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

A small subset of the static T4 Huffman codes are given in Table 2.2.1 which also shows the most likely runs to occur. 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 ratio of compression rises 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.

2.2.2 Limitations

As mentioned in the 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. To reduce this problem there are several approaches known, partly implemented and used some of which will be addressed like a Burrows-Wheeler-Transformation.

2.2.3 Run Length Encoding today

While it is still in use by fax transmission or other highly specific tasks, it is mostly used in combination with other approaches. TODO: give more references

2.3 Prefix Coding

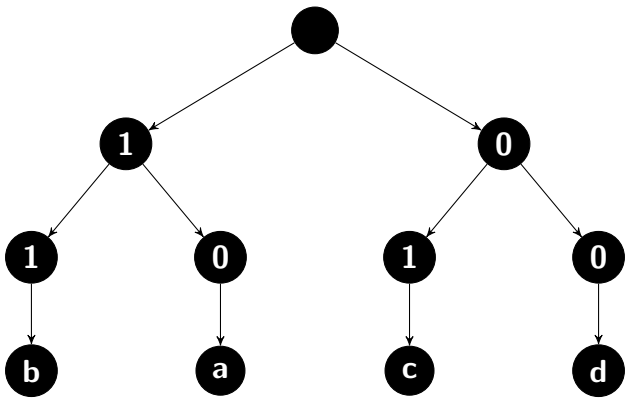
Prefix codes make use of an untypical idea in computer science. Usually we deal with fixed length codes like 7 bit ASCII or 32 bit Integer representations which map each possible value into that fixed amount of bits. For compression it would be a benefit if we could write codes of variable length. The 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 starts and finishes, resulting in ambiguous encoding. For example the set of codes $\{(a, 1), (b, 01), (c, 101), (d, 011)\}$ and the encoded message is 1011, there is no way to tell which message was encoded.

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 uniquely decodable as show in the next section.

2.3.1 Huffman Coding

Huffman coding

TODO: fix dang tree to be able to have only one child node



TODO: explain stuff ...

2.4 Other methods

There are other well known methods like
- move to front
- arithmetic coding
- real world applications are mostly combinations of those
...

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 [2]. It is a Transformation of a string S of n characters by forming the n rotations of S, sorting them lexicographically, and extracting the last character of each of the rotations. The result of this transformation is a string

L, consisting of these last characters of each sorted rotation. The algorithm also has to add special termination symbols or compute the index I of the original string S in the sorted list of rotations. Surprisingly, there is an efficient algorithm to invert the transformation back to the original string S given only L and the Index I [8].

As an example of the creation of L, given the input string S = 'abraca', $n = 6$, and the alphabet $X = \{a, b, c, r\}$. Create a $N \times N$ matrix M whose elements are characters, and whose rows are rotations of S. Sort all rotations in lexicographical order.

In this example, the index is $I = 1$ and the matrix M is

row 0	a	a	b	r	a	c
row 1	a	b	r	a	c	a
row 2	a	c	a	a	b	r
row 3	b	r	a	c	a	a
row 4	c	a	a	b	r	a
row 5	r	a	c	a	a	b

The resulting string 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, $L =$ '$caraab$' and $I = 1$. Obviously the string L contains consecutive identical characters, which result in better compressibility as shown in 3.2.1.

TODO: explain reversing a bwt

2.5 State of the Art

State of the art compression
- techniques
- use cases
- limits

method	size in bytes	compression	ratio in $\frac{bits}{symbol}$
uncompressed	3,141,622	100%	8.00
compress	1,272,772	40.0%	3.24
ZIP v2.32	1,020,781	32.4%	2.59
gzip v1.3.5	1,017,624	32.3%	2.58
bzip2 v9.12b	828,347	26.3%	2.10
ppmd	740,737	23.5%	1.88
ZPAQ v7.15	659,709	20.9%	1.67

2.6 Limits of compression

- existence of not compressable strings
-https://www.quora.com/Is-there-a-theoretical-limit-to-data-compression-If-so-how-was-it-found
- unable to compress random data
- Kolmogorow-Komplexität

3. Analysis

The following chapter contains a detailed analysis of the problem and some fundamental 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 where very underwhelming as expected.

3.1 Initial Findings

Originally developed and used on black and white pallet images containing only two values, on a binary level so to speak, 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.

bits per rle number	expansion ratio	bits per symbol in $\frac{bits}{symbol}$
8	3.298046423741734	26.38437138993387
7	2.8897459975751163	23.11796798060093
6	2.4839251325134675	19.87140106010774
5	2.0825541259578895	16.66043300766311
4	1.6895640359371056	13.516512287496845
3	1.3130541262757818	10.504433010206254
2	1.04555716691706	8.36445733533648

The results in Table 3.1 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 encding to be able to encode runs with variable length.

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

bits per rle number	ratio in %	bits per symbol in $\frac{bits}{symbol}$
8	165	13.2
7	154	12.38
6	144	11.57
5	134	10.77
4	125	10.00
3	116	9.29
2	109	8.74

However after some analysis of the corpus data, it was shown that most runs had a value of one and almost no runs larger that 4 occurred, which lead to the conclusion, two bit for the run count should be plenty, which is also shown in Table 3.1. Even with a run size of just two bits, there is still a increase in size of about 9% and uses 8.74 $\frac{bits}{symbol}$. 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, 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 $\frac{bits}{symbol}$ while the other files just doubled or even tripled in size. Using the byte wise operating RLE we see a similar result but not as decent with 27.2% of its original size using 2.17 $\frac{bits}{symbol}$ encoding with 6 bits per run.

3.2 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.

3.2.1 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 a 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.” [2]

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.

3.2.2 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. 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, it results in much longer runs as shown below.

2-D:

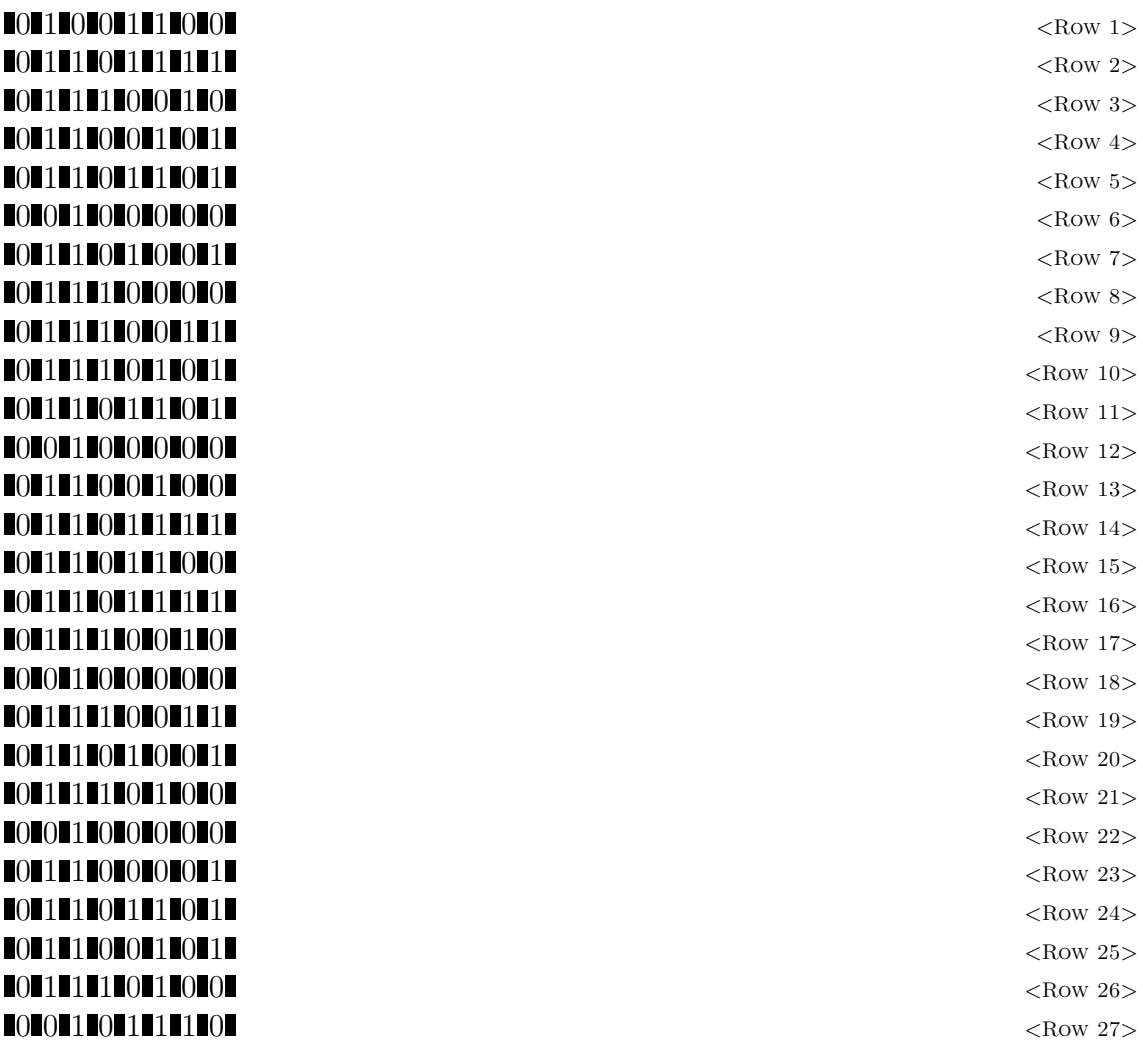
Now we have 5 runs of length 6, 2 runs of length 3, 3 runs of length 2 and just 6 runs of length 1 as opposed by the simple interpretation. This is because the binary similarity between the used characters, as the character for a and b only differ in one bit. It is clear that simply a different way of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

3.2.3 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 lover values results in the following horizontally interpreted rows.

2-D:

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.

3.2.4 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 well, the vertical byte reading enables new approaches, even more in combination with the idea of byte remapping.

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 Summary

TODO

Am Ende sollten ggf. die wichtigsten Ergebnisse nochmal in *einem* kurzen Absatz zusammengefasst werden.

4. Conceptual Design and Implementation

Without further ado, design and implementation of the initial ideas began. 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 Burrows-Wheeler-Transformation Appliance

By mistake a 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 does consist of more than textual data, in fact the files geo, obj1, obj2 and pic contain of some binary data of include 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 Quicksort. This algorithm offers $\Theta(n\ log(n))$ average time complexity but in the worst case, its time complexity is 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 without much effort.

TODO: show native bwt results for binary and byte wise RLE

bits per rle number	ratio in %	bits per symbol in $\frac{bits}{symbol}$
3	95.4169856525027	7.633358852200216
2	91.391	7.311309412577118

TODO: change impl & describe actual impl and its benefits

4.2 Vertical Byte Reading

Implementing the vertical reading of the input shown in Section 4.2 was not hard but it should be kept in mind that the size of the input chunks has to be divisible by the size of 8. Otherwise parsing it into an Array of Bytes results in the last Byte having some padding which might cause problems later on. By collecting the bits into a proprietary data structure, we avoid this problem but working with bytes internally should be easier, nonetheless both ways of collecting all bits of identical significance are implemented.

4.2.1 First Ideas

Initially some other ideas have been followed with very poor results. One idea was arranging all input bits in a Matrix or square Matrix in a way it would still be receivable later on. This way other methods from linear algebra would have been applicable to the input data, so that the construction of a triangular matrix or other preprocessing would result in long runs of zeros. Difficulties in the construction and transformation of the Data lead to the abandonment of this approach and the already described vertical interpretation was used further on.

4.2.2 Performance Improvements

...

4.3 Preprocessing

...

4.3.1 Byte Mapping to reduce Input space

...

4.3.2 Dynamic Encoding

...

4.4 Alternative Compression for partial data

...

4.4.1 Huffman Coding

...

4.5 Implementation Decisions

- why kotlin
- performance improvements with the graalvm
- other decisions

...

4.6 Implementation Detail

- detailed information about specific modules and classes

...

4.6.1 Parsing

- explain universal parsing

4.6.2 Burrows Wheeler Transformation

- TBD

4.6.3 Byte Remapping

- show use case

4.6.4 Dynamic Encoding

- different sizes and optima for different files and chunk sizes

4.7 Implementation Evaluation

- evaluation of implementation choices made

4.8 Summary

Am Ende sollten ggf. die wichtigsten Ergebnisse nochmal in *einem* kurzen Absatz zusammengefasst werden.

5. Evaluation

Hier erfolgt der Nachweis, dass das in Kapitel 4 entworfene Konzept funktioniert. Leistungsmessungen einer Implementierung werden immer gerne gesehen.

5.1 Functional Evaluation

- Mathematical Comparison
- Comparison of encoded file sizes
- comparing to regular REL and Huffman coding

...

5.2 Benchmarks

- Benchmark with the Galgary corpus
http://www.data-compression.info/Corpora/

...

5.2.1 Burrows-Wheeler-Transformation

bits per rle number	ratio in %	bits per symbol in $\frac{bits}{symbol}$
3	95.4169856525027	7.633358852200216
2	91.391	7.311309412577118

5.2.2 Vertical encoding

5.3 Conclusion

Am Ende sollten ggf. die wichtigsten Ergebnisse nochmal in *einem* kurzen Absatz zusammengefasst werden.

6. Discussion

(Keine Untergliederung mehr)

Bibliography

[1] Al-Okaily A, Almarri B, Al Yami S, and Huang CH. Toward a better compression for dna sequences using huffman encoding. *J Comput Biol.*, 24(4):280–288, 2017.

[2] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[3] MJ Dallwitz. An introduction to computer images. *TDWG Newsletter*, 7(10):2, 1992.

[4] Fabio G. Guerrero, editor. *A new look at the classical entropy of written English*, 2009.

[5] Roy Hunter and A. Harry Robinson, editors. *International digital facsimile coding standards, Proceedings of the IEEE 68*, 1980.

[6] International Telecommunication Union (ITU). *T.4 : Standardization of Group 3 facsimile terminals for document transmission*, 2004.

[7] Maciej Liśkiewicz and Henning Fernau. *Datenkompression.*

[8] Daisuke Okanohara and Kunihiko Sadakane. A linear-time burrows-wheeler transform using induced sorting. In Jussi Karlgren, Jorma Tarhio, and Heikki Hyyrö, editors, *String Processing and Information Retrieval*, pages 90–101. Springer Berlin Heidelberg, 2009.

[9] A. Harry Robinson and C. Cherry, editors. *Results of a prototype television bandwidth compression scheme, Proceedings of the IEEE 3*, volume 55, March 1967.

[10] C. E. Shannon. *Prediction and entropy of printed English*, volume 30. Jan 1951.

[11] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

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