hochschule mannheim

Comparison of compression tools for biological data and analysis of possible improvements

Gabriel Eichelkraut

Bachelor Thesis

for the acquisition of the academic degree Bachelor of Science (B.Sc.)

Course of Studies: Computer Science

Department of Computer Science

University of Applied Sciences Mannheim

01.12.22

Tutors

Prof. Elena Fimmel, Hochschule Mannheim

TBD

Eichelkraut, Gabriel:
Comparison of compression tools for biological data and analysis of possible improvements
/ Gabriel Eichelkraut. –
Bachelor Thesis, Mannheim: University of Applied Sciences Mannheim, 2022. 34 pages.

Eichelkraut, Gabriel:
Vergleich von Kompressionswerkzeugen für biologische Daten und Analyse von Verbesserungsmöglichkeiten
/ Gabriel Eichelkraut. –
Bachelor-Thesis, Mannheim: Hochschule Mannheim, 2022. 34 Seiten.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mannheim, 01.12.22

Max Mustermann

Gabriel Eichelkraut

Abstract

Comparison of compression tools for biological data and analysis of possible improvements

TBD.

Vergleich von Kompressionswerkzeugen für biologische Daten und Analyse von Verbesserungsmöglichkeiten

TBD.

Contents

1. Introduction 1

2. The Structure of the Human Genome and how its Digital Form is Compressed 3

2.1. Structure of Human DNA . . . 3

2.2. File Formats used to Store DNA . . . 5

2.2.1. FASTA and FASTq . . . 7

2.2.2. Sequence Alignment Map . . . 8

2.3. Compression aproaches . . . 9

2.3.1. Dictionary coding . . . 10

2.3.2. Shannons Entropy . . . 11

2.3.3. Arithmetic coding . . . 12

2.3.4. Huffman encoding . . . 14

2.4. Implementations in Relevant Tools . . . 17

2.4.1. Genome Compressor (GeCo) . . . 17

2.4.2. Samtools . . . 18

3. Environment and Procedure to Determine the State of The Art Efficiency and Compressionratio of Relevant Tools 22

3.1. Sever specifications and test environment . . . 23

3.2. Operating System and Additionally Installed Packages . . . 24

3.3. Selection, Receivement, and Preperation of Testdata . . . 24

4. Results and Discussion 27

4.1. Interpretation of Results . . . 27

4.2. View on Possible Improvements . . . 30

List of Abbreviations v

List of Tables vi

List of Figures vii

Listings viii

Bibliography ix

Index xiii

A. Erster Anhang: Lange Tabelle xiii

Chapter 1

Introduction

Understanding how things in our cosmos work, was and still is a pleasure, that the human being always wants to fulfill. Getting insights into the rawest form of organic life is possible through storing and studying information, embedded in genetic codes. Since live is complex, there is a lot of information, which requires a lot of memory.
With compression tools, the problem of storing information got restricted. Compressed data requires less space and therefore less time to be transported over networks. This advantage is scalable and since genetic information needs a lot of storage, even in a compressed state, improvements are welcomed. Since this field is, compared to others, like computer theory and compression approaches, relatively new, there is much to discover and new findings are not unusual. From some of this findings, new tools can be developed. They optimally increase two factors: the speed at which data is compressed and the compresseion ratio, meaning the difference between uncompressed and compressed data.

New discoveries in the universal rules of stochastical organization of genomes might provide a base for new algorithms and therefore new tools or an improvement of existing ones for genome compression. The aim of this work is to analyze the current state of the art for probabilistic compression tools and their algorithms, and ultimately determine whether mentioned discoveries are already used. `might be thrown out due to time limitations` -> If this is not the case, there will be an analysation of how and where this new approach could be imprelented and if it would improve compression methods.

To reach a common ground, the first pages will give the reader a quick overview on the structure of human DNA. There will also be an superficial explanation for some basic terms, used in biology and computer science. The knowledge basis of this work is formed by describing differences in file formats, used to store genome data. In addition to this, a section relevant for compression will follow. This will go through the state of the art in coding theory.
In order to meassure an improvement, first a baseline must be set. Therefore the efficiency and effecitity of suiteable state of the art tools will be meassured. To be as precise as possible, the main part of this work focuses on setting up an environment, picking input data, installing and executing tools and finaly meassuring and documenting the results.
With this information, a static code analysis of mentioned tools follows. This will show where a possible new algorithm or an improvement to an existing one could be implemented. Running a proof of concept implementation under the same conditions and comparing runtime and compression ratio to the defined baseline shows the potential of the new approach for compression with probability algorithms.

Chapter 2

The Structure of the Human Genome and how its Digital Form is Compressed

2.1. Structure of Human DNA

To strengthen the understanding of how and where biological information is stored, this section starts with a quick and general rundown on the structure of any living organism.

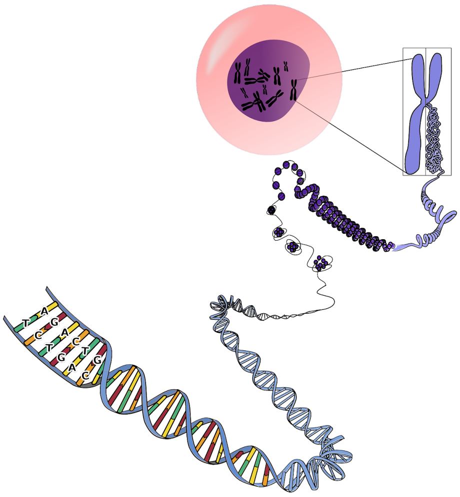


Figure 2.1.: A superficial representation of the physical positioning of genomes. Showing a double helix (bottom), a chromosome (upper rihgt) and a chell (upper center).

All living organisms, like plants and animals, are made of cells (a human body can consist out of several trillion cells) [1].

A cell in itself is a living organism; The smallest one possible. It consists out of two layers from which the inner one is called nucleus. The nucleus contains chromosomes and those chromosomes hold the genetic information in form of Deoxyribonucleic Acid (DNA).

DNA is often seen in the form of a double helix. A double helix consists, as the name suggests, of two single helix.

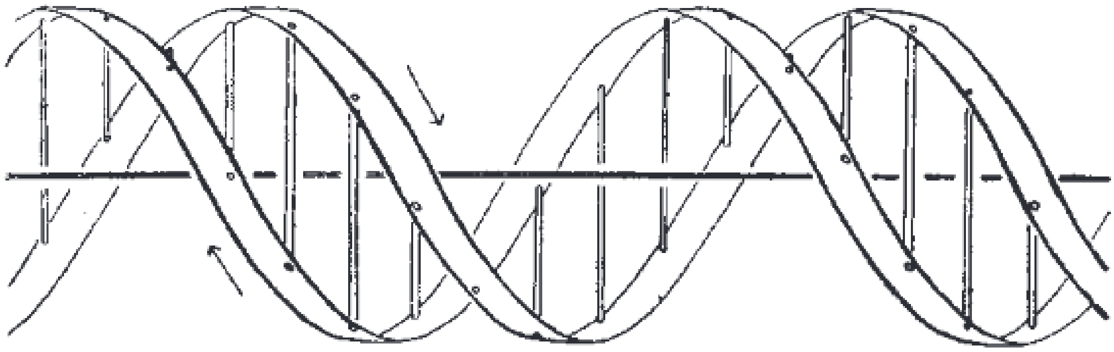


Figure 2.2.: A purely diagrammatic figure of the components DNA is made of. The smaller, inner rods symbolize nucleotide links and the outer ribbons the phosphate-sugar chains [2].

Each of them consists of two main components: the Sugar Phosphate backbone, which is not relevant for this work and the Bases. The arrangement of Bases represents the Information stored in the DNA. A base is an organic molecule, they are called Nucleotides [2].

For this work, nucleotides are the most important parts of the DNA. A Nucleotide can occur in one of four forms: it can be either adenine, thymine, guanine or cytosine. Each of them got a Counterpart with which a bond can be established: adenine can bond with thymine, guanine can bond with cytosine.
From the perspective of an computer scientist: The content of one helix must be stored, to persist the full information. In more practical terms: The nucleotides of only one (entire) helix needs to be stored physically, to save the information of the whole DNA because the other half can be determined by "inverting" the stored one. An example would show the counterpart for e.g.: `adenine, guanine, adenine` chain which would be a chain of `thymine, cytosine, thymine`. For the sake of simplicity, one does not write out the full name of each nucleotide, but only its initiat. So the example would change to `AGA` in one Helix, `TCT` in the other.
This representation ist commonly used to store DNA digitally. Depending on the sequencing procedure and other factors, more information is stored and therefore

more characters are required but for now ’A’, ’C’, ’G’ and ’T’ should be the only concern.

2.2. File Formats used to Store DNA

As described in previous chapters DNA can be represented by a string with the buildingblocks A,T,G and C. Using a common file format for saving text would be impractical because the ammount of characters or symbols in the used alphabet, defines how many bits are used to store each single symbol.
The American Standard Code for Information Interchange (ASCII) table is a character set, registered in 1975 and to this day still in use to encode texts digitally. For the purpose of communication bigger character sets replaced ASCII. It is still used in situations where storage is short.
Storing a single *A* with ASCII encoding, requires 8 bit (excluding magic bytes and the bytes used to mark End of File (EOF)) . Since there are at least 2^8 or 128 displayable symbols. The buildingblocks of DNA require a minimum of four letters, so two bits are needed In most tools, more than four symbols are used. This is due to the complexity in sequencing DNA. It is not 100% preceice, so additional symbols are used to mark nucelotides that could not or could only partly get determined. Further a so called quality score is used to indicate the certainty, for each single nucleotide, that it got sequenced correctly.
More common everyday-usage text encodings like unicode require 16 bits per letter. So settling with ASCII has improvement capabilities but is, on the other side, more efficient than using bulkier alternatives like unicode.

Formats for storing uncompressed genomic data, can be sorted into several categories. Three noticable ones would be [3]:

- sequenced reads
- aligned data
- sequence variation

The categories are listed on their complexity, considering their usecase and data structure, in ascending order. Starting with sequence variation, also called haplotype describes formats storing graph based structures that focus on analysing variations in different genomes **haplo, survey!**. Sequenced reads focus on storing continous protein chains from a sequenced genome **survey!**. Aligned data is somwhat simliar to sequenced reads with the difference that instead of a whole chain of genomes, overlapping subsequences are stored. This could be described as a rawer form of sequenced reads. This way aligned data stores additional information on how certain a specific part of a genome is read correctly. The focus of this work lays on compression of sequenced data but not on the likelyhood of how accurate the data might be. Therefore, only formats that include sequenced reads will be worked with.

Several people and groups have developed different file formats to store genomes. Unfortunaly, the only standard for storing genomic data is fairly new [4], [5]. Therefore, formats and tools implementing this standard are mostly still in development. In order to not go beyond scope, this work will focus only on file formats that fulfill following criteria:

- the format has reputation
  - through a scientific paper, that prooved its superiority to other relevant tools.
  - through a broad ussage of the format determined by its use on ftp servery that focus on supporting scientific research.
- the format should not specialize on only one type of DNA.
- the format stores nucleotide seuqences and does not neccesarily include International Union of Pure and Applied Chemistry (IUPAC) codes besides A, C, G and T [6].
- the format is open source. Otherwise, improvements can not be tested, without buying the software and/or requesting permission to disassemble and reverse engineer the software or parts of it.

Information on available formats where gathered through various Internet platforms [7]–[9]. Some common file formats found:

- File Format for Storing Genomic Data (FASTA)
- File Format Based on FASTA (FASTq)
- Sequence Alignment Map (SAM)/Binary Alignment Map (BAM) [10], [11]

- Compressed Reference-oriented Alignment Map (CRAM) [10], [11]
- twoBit [12]

Since methods to store this kind of Data are still in development, there are many more file formats. From the selection listed above, FASTA and FASTq seem to have established the reputation of a inoficial standard for sequenced reads **cram-origin**, [3], [13], [14].
Considering the first criteria, by searching through anonymously accesable **ftp!** servers, only two formats are used commonly: FASTA or its extension FASTq and the BAM Format **ftp-igsr, ftp-ncbi, ftp-ensembl!**.

2.2.1. FASTA and FASTq

The rather simple FASTA format consists of two repeated sections. The first section consists of one line and stores metadata about the sequenced genome and the file itself. This line, also called header, contains a comment section starting with > followed by a custom text [15], [16]. The comment section is usually used to store information about the sequenced genome and sometimes metadata about the file itself like its size in bytes.
The other section contains the sequenced genome whereas each nucleotide is represented by character `A, C, G or T`. There are three more nucleotide characters that store additional information and some characters for representing amino acids, but in order to not go beyond scope, only `A, C, G, and T` will be paid attention to.
The second section can take multiple lines and is determined by a empty line. After that the file end is reached or another touple of header and sequence can be found.
In addition to its predecessor, FASTq files contain a quality score. The file content consists of four sections, wherby no section is stored in more than one line. All four lines contain information about one sequence. The exact structure of FASTq is formated in this order [17]:

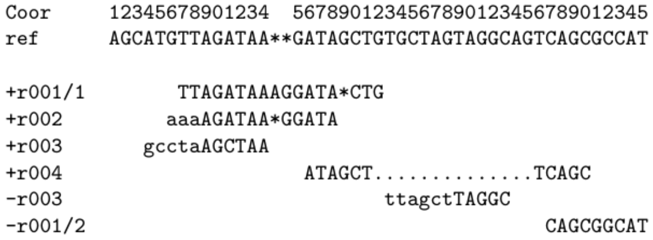
- Line 1: Sequence identifier aka. Title, starting with an @ and an optional description.
- Line 2: The seuqence consisting of nucleoids, symbolized by A, T, G and C.
- Line 3: A '+' that functions as a seperator. Optionally followed by content of Line 1.

- Line 4: quality line(s). consisting of letters and special characters in the ASCII scope.

The quality scores have no fixed format. To name a few, there is the sanger format, the solexa format introduced by Solexa Inc., the Illumina and the QUAL format which is generated by the PHRED software [16].
The quality value shows the estimated probability of error in the sequencing process.

2.2.2. Sequence Alignment Map

SAM often seen in its compressed, binary representation BAM with the fileextension `.bam`, is part of the SAMtools package, a uitlity tool for processing SAM/BAM and CRAM files. The SAM/BAM file is a text based format delimited by TABs [10]. It uses 7-bit US-ASCII, to be precise Charset ANSI X3.4-1968 [18]. The structure is more complex than the one in FASTq and described best, accompanied by an example:



```
Coor     12345678901234  5678901234567890123456789012345
ref      AGCATGTTAGATAA**GATAGCTGTGCTAGTAGGCAGTCAGCGCCAT

+r001/1        TTAGATAAAGGATA*CTG
+r002         aaaAGATAA*GGATA
+r003       gcctaAGCTAA
+r004                     ATAGCT..............TCAGC
-r003                            ttagctTAGGC
-r001/2                                        CAGCGGCAT
```

Figure 2.3.: SAM/BAM file structure example

Compared to FASTA SAM and further compression forms, store more information. As displayed in **??** this is done by adding, identifier for Reads e.g. **+r003**, aligning subsequences and writing additional symbols like dots e.g. **ATAGCT......** in the split alignment +r004 [10]. A full description of the information stored in SAM files would be of little value to this work, therefore further information on is left out but can be found in [11] or at [10]. Samtools provide the feature to convert a FASTA file into SAM format. Since there is no way to calulate mentioned, additional information from the information stored in FASTA, the converted files only store two lines. The first one stores metadata about the file and the second stores the nucleotide sequence in just one line.

2.3. Compression aproaches

The process of compressing data serves the goal to generate an output that is smaller than its input data.

In many cases, like in gene compressing, the compression is ideally lossless. This means it is possible for every compressed data, to receive the whole information, which were available in the origin data, by decompressing it.

Before going on, the difference between information and data should be emphasized.

Data contains information. In digital data clear, physical limitations delimit what and how much of something can be stored. A bit can only store 0 or 1, eleven bits can store up to $2^1 1$ combinations of bits and a 1 Gigabyte drive can store no more than 1 Gigabyte data. Information on the other hand, is limited by the way how it is stored. In some cases the knowledge received in a earlier point in time must be considered too, but this can be neglected for reasons described in the subsection 2.3.1.

The boundaries of information, when it comes to storing capabilities, can be illustrated by using the example mentioned above. A drive with the capacity of 1 Gigabyte could contain a book in form of images, where the content of each page is stored in a single image. Another, more resourceful way would be storing just the text of every page in **UTF-16!**. The information, the text would provide to a potential reader would not differ. Changing the text encoding to ASCII and/or using compression techniques would reduce the required space even more, without loosing any information.

In contrast to lossless compression, lossy compression might excludes parts of data in the compression process, in order to increase the compression rate. The excluded parts are typically not necessary to persist the origin information. This works with certain audio and pictures formats, and in network protocols [19]. For DNA a lossless compression is needed. To be precise a lossy compression is not possible, because there is no unnecessary data. Every nucleotide and its position is needed for the sequenced DNA to be complete. For lossless compression two mayor approaches are known: the dictionary coding and the entropy coding. Methods from both fields, that aquired reputation, are described in detail below [15], [20]–[22].

2.3.1. Dictionary coding

Disclaimer Unfortunally, known implementations like the ones out of LZ Family, do not use probabilities to compress and are therefore not in the main scope for this work. To strenghten the understanding of compression algortihms this section will remain. Also a hybrid implementation described later will use both dictionary and entropy coding.

Dictionary coding, as the name suggest, uses a dictionary to eliminate redundand occurences of strings. Strings are a chain of characters representing a full word or just a part of it. For a better understanding this should be illustrated by a short example: Looking at the string 'stationary' it might be smart to store 'station' and 'ary' as seperate dictionary enties. Which way is more efficient depents on the text that should get compressed. The dictionary should only store strings that occour in the input data. Also storing a dictionary in addition to the (compressed) input data, would be a waste of resources. Therefore the dicitonary is made out of the input data. Each first occourence is left uncompressed. Every occurence of a string, after the first one, points to its first occurence. Since this 'pointer' needs less space than the string it points to, a decrease in the size is created.

The LZ Family

The computer scientist Abraham Lempel and the electrical engineere Jacob Ziv created multiple algorithms that are based on dictionary coding. They can be recognized by the substring `LZ` in its name, like `LZ77 and LZ78` which are short for Lempel Ziv 1977 and 1978. The number at the end indictates when the algorithm was published. Today LZ78 is widely used in unix compression solutions like gzip and bz2. Those tools are also used in compressing DNA.
LZ77 basically works, by removing all repetition of a string or substring and replacing them with information where to find the first occurence and how long it is. Typically it is stored in two bytes, whereby more than one one byte can be used to point to the first occurence because usually less than one byte is required to store the length.

Lempel Ziv 1977 (LZ77) basically works, by removing all repetition of a string or substring and replacing them with information where to find the first occurence and how long it is. Typically it is stored in two bytes, whereby more than one one byte can be used to point to the first occurence because usually less than one byte is required to store the length.

2.3.2. Shannons Entropy

The founder of information theory Claude Elwood Shannon described entropy and published it in 1948 [23]. In this work he focused on transmitting information. His theorem is applicable to almost any form of communication signal. His findings are not only usefull for forms of information transmition.

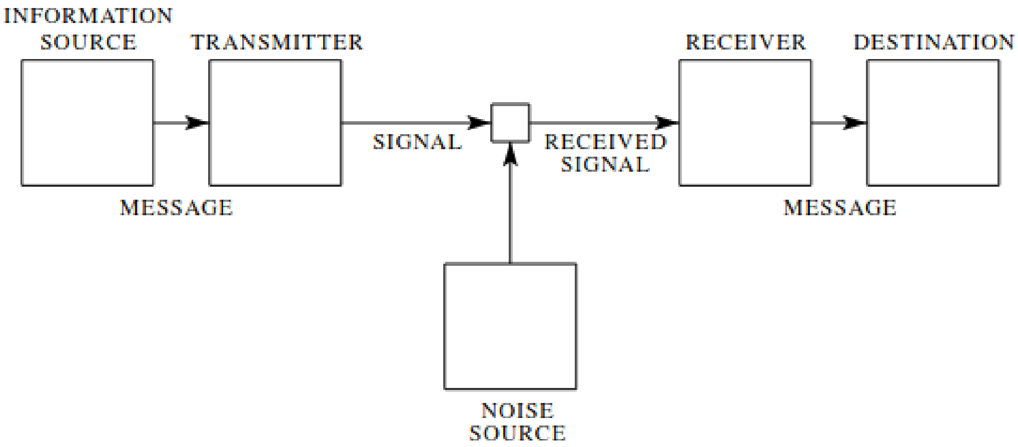


Figure 2.4.: Schematic diagram of a general communication system by Shannons definition. [23]

Altering 2.4 would show how this can be applied to other technology like compression. The Information source and destination are left unchanged, one has to keep in mind, that it is possible that both are represented by the same phyiscal actor. Transmitter and receiver would be changed to compression/encoding and decompression/decoding and inbetween ther is no signal but any period of time [23].

Shannons Entropy provides a formular to determine the 'uncertainty of a probability distribution' in a finite field.

$$H(X) := \sum_{x \in X, prob(x) \neq 0} prob(x) \cdot log_2(\frac{1}{prob(x)}) \equiv - \sum_{x \in X, prob(x) \neq 0} prob(x) \cdot log_2(prob(x)). \tag{2.1}$$

He defined entropy as shown in figure (2.1). Let X be a finite probability space. Then x in X are possible final states of an probability experimen over X. Every state that actually occours, while executing the experiment generates infromation which is meassured in *Bits* with the part of the formular displayed in 2.2 [23], [24]:

$$log_2(\frac{1}{prob(x)}) \equiv -log_2(prob(x)). \tag{2.2}$$

2.3.3. Arithmetic coding

This coding method is an approach to solve the problem of wasting memeory due to the overhead which is created by encoding certain lenghts of alphabets in binary. For example: Encoding a three-letter alphabet requires at least two bit per letter. Since there are four possilbe combinations with two bits, one combination is not used, so the full potential is not exhausted. Looking at it from another perspective and thinking a step further: Less storage would be required, if there would be a possibility to encode more than one letter in two bit.
Dr. Jorma Rissanen described arithmetic coding in a publication in 1976 [25]. This works goal was to define an algorithm that requires no blocking. Meaning the input text could be encoded as one instead of splitting it and encoding the smaller texts or single symbols. He stated that the coding speed of arithmetic coding is comparable to that of conventional coding methods [25].

This is possible by projecting the input text on a binary encoded fraction between 0 and 1. To get there, each character in the alphabet is represented by an interval between two floating point numbers in the space between 0.0 and 1.0 (exclusively). This interval is determined by the symbols distribution in the input text (interval start) and the the start of the next character (interval end). The sum of all intervals will result in one.
To encode a text, subdividing is used, step by step on the text symbols from start to the end. The interval that represents the current character will be subdivided.

Meaning the choosen interval will be divided into subintervals with the proportional size of the intervals calculated in the beginning.

To store as few informations as possible and due to the fact that fractions in binary have limited accuracity, only a single number, that lays between upper and lower end of the last intervall will be stored. To encode in binary, the binary floating point representation of any number inside the interval, for the last character is calculated, by using a similar process, described above. To summarize the encoding process in short:

- The interval representing the first character is noted.
- Its interval is split into smaller intervals, with the ratios of the initial intervals between 0.0 and 1.0.
- The interval representing the second character is choosen.
- This process is repeated, until a interval for the last character is determined.
- A binary floating point number is determined wich lays in between the interval that represents the represents the last symbol.

For the decoding process to work, the EOF symbol must be be present as the last symbol in the text. The compressed file will store the probabilies of each alphabet symbol as well as the floatingpoint number. The decoding process executes in a simmilar procedure as the encoding. The stored probabilies determine intervals. Those will get subdivided, by using the encoded floating point as guidance, until the EOF symbol is found. By noting in which interval the floating point is found, for every new subdivision, and projecting the probabilies associated with the intervals onto the alphabet, the origin text can be read.

In computers, arithmetic operations on floating point numbers are processed with integer representations of given floating point number [26]. The number 0.4 + would be represented by $4 \cdot 10^{-}1$.

Intervals for the first symbol would be represented by natural numbers between 0 and 100 and $... \cdot 10^{-}x$. x starts with the value 2 and grows as the intgers grow in length, meaning only if a uneven number is divided. For example: Dividing a uneven number like $5 \cdot 10^{-}1$ by two, will result in $25 \cdot 10^{-}2$. On the other hand, subdividing $4 \cdot 10^{y}$ by two, with any negativ real number as y would not result in a greater x the length required to display the result will match the length required to display the input number.

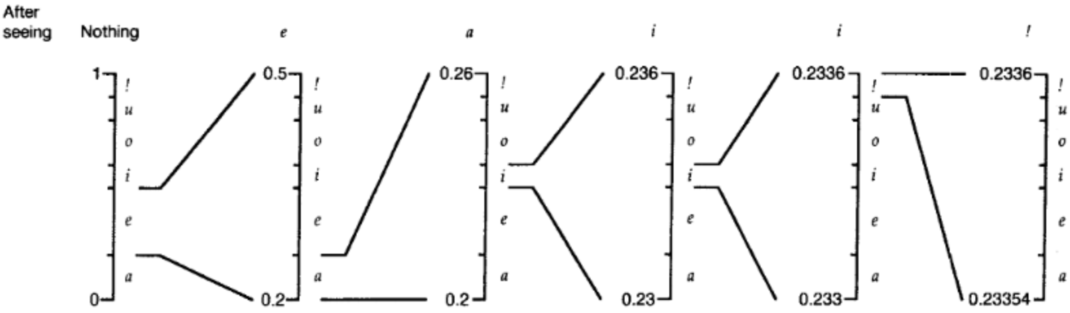


Figure 2.5.: Illustrative rescaling in arithmetic coding process. [27]

The described coding is only feasible on machines with infinite percission. As soon as finite precission comes into play, the algorithm must be extendet, so that a certain length in the resulting number will not be exceeded. Since digital datatypes are limited in their capacity, like unsigned 64-bit integers which can store up to $2^{6}4 - 1$ bits or any number between 0 and 18,446,744,073,709,551,615. That might seem like a great ammount at first, but considering a unfavorable alphabet, that extends the results lenght by one on each symbol that is read, only texts with the length of 63 can be encoded (62 if EOF is exclued).

2.3.4. Huffman encoding

D. A. Huffmans work focused on finding a method to encode messages with a minimum of redundance. He referenced a coding procedure developed by Shannon and Fano and named after its developers, which worked similar. The Shannon-Fano coding is not used today, due to the superiority in both efficiency and effectivity, in comparison to Huffman. Even though his work was released in 1952, the method he developed is in use today. Not only tools for genome compression but in compression tools with a more general ussage [28].

Compression with the Huffman algorithm also provides a solution to the problem, described at the beginning of 2.3.2, on waste through unused bits, for certain alphabet lengths. Huffman did not save more than one symbol in one bit, like it is done in arithmetic coding, but he decreased the number of bits used per symbol in a message. This is possible by setting individual bit lengths for symbols, used in the text that should get compressed [29]. As with other codings, a set of symbols must be defined. For any text constructed with symbols from mentioned alphabet, a binary tree is constructed, which will determine how the symbols will be encoded. As in arithmetic coding, the probability of a letter is calculated for given text. The binary tree will be constructed after following guidelines [15]:

- Every symbol of the alphabet is one leaf.
- The right branch from every knot is marked as a 1, the left one is marked as a 0.
- Every symbol got a weight, the weight is defined by the frequency the symbol occours in the input text. This might be a fraction between 0 and 1 or an integer. In this scenario it will described as the first.
- The less weight a leaf has, the higher the probability is, that this node is read next in the symbol sequence.
- The leaf with the lowest probability is most left and the one with the highest probability is most right in the tree.

A often mentioned difference between Shannon-Fano and Huffman coding, is that first is working top down while the latter is working bottom up. This means the tree starts with the lowest weights. The nodes that are not leafs have no value ascribed to them. They only need their weight, which is defined by the weights of their individual child nodes [15], [21].

Given `K(W,L)` as a node structure, with the weigth or probability as W_i and codeword length as L_i for the node K_i. Then will L_{av} be the average length for `L` in a finite chain of symbols, with a distribution that is mapped onto `W` **huf**.

$$L_{av} = \sum_{i=0}^{n-1} w_i \cdot l_i \tag{2.3}$$

The equation (2.3) describes the path, to the desired state, for the tree. The upper bound `n` is assigned the length of the input text. The touple in any node `K` consists of a weight w_i, that also references a symbol, and the length of a codeword l_i. This codeword will later encode a single symbol from the alphabet. Working with digital codewords, an element in `l` contains a sequence of zeros and ones. Since there in this coding method, there is no fixed length for codewords, the premise of `prefix free code` must be adhered to. This means there can be no codeword that match the sequence of any prefix of another codeword. To illustrate this: 0, 10, 11 would

be a set of valid codewords but adding a codeword like 01 or 00 would make the set invalid because of the prefix 0, which is already a single codeword.
With all important elements described: the sum that results from this equation is the average length a symbol in the encoded input text will require to be stored [21], [29].

For this example a four letter alphabet, containing `A, C, G and T` will be used. The exact input text is not relevant, since only the resulting probabilities are needed. With a distribution like `<A,` $100\frac{1}{1} = 0.11$`>, <C,` $100\frac{7}{1} = 0.71$`>, <G,` $100\frac{1}{3} = 0.13$`>, <T,` $100\frac{5}{=}0.05$>, a probability for each symbol is calculated by dividing the message length by the times the symbol occured.
For an alphabet like the one described above, the binary representation encoded in ASCI is shown here `A -> 01000001, C -> 01000011, G -> 01010100, T -> 00001010`. The average length for any symbol encoded in ASCII is eight, while only using four of the available 2^8 symbols, a overhead of 252 unused bit combinations. For this example it is more vivid, using a imaginary encoding format, without overhead. It would result in a average codeword length of two, because four symbols need a minimum of 2^2 bits.
So starting with the two lowest weightened symbols, a node is added to connect both.
`<A, T>, <C>, <G>`
With the added, blank node the count of available nodes got down by one. The new node weights as much as the sum of weights of its child nodes so the probability of 0.16 is assigned to `<A,T>`. From there on, the two leafs will only get rearranged through the rearrangement of their temporary root node. Now the two lowest weights are paired as described, until there are only two subtrees or nodes left which can be combined by a root.
`<C, <A, T», <G>`
The `<C, <A, T»` has a probability of 0.29. Adding the last node `G` results in a root node with the probability of 1.0.
With the fact in mind, that left branches are assigned with 0 and right branches with 1, following a path until a leaf is reached reveals the encoding for this particular leaf. With a corresponding tree, created from with the weights, the binary sequences to encode the alphabet would look like this:
`A -> 0, C -> 11, T -> 100, G -> 101.`
Since high weightened and therefore often occuring leafs are positioned to the left,

short paths lead to them and so only few bits are needed to encode them. Following the tree on the other side, the symbols occur more rarely, paths get longer and so do the codeword. Applying (2.3) to this example, results in 1.45 bits per encoded symbol. In this example the text would require over one bit less storage for every second symbol.
Leaving the theory and entering the practice, brings some details that lessen this improvement by a bit. A few bytes are added through the need of storing the information contained in the tree. Also, like described in 2.2 most formats, used for persisting DNA, store more than just nucleotides and therefore require more characters. What compression ratios implementations of huffman coding provide, will be discussed in **??**.

2.4. Implementations in Relevant Tools

This section should give the reader a quick overview, how a small variety of compression tools implement described compression algorithms.

2.4.1. GeCo

This tool has three development stages, the first GeCo released in 2016 **geco!**. This tool happens to have the smalles codebase, with only eleven C files. The two following extensions GeCo2, released in 2020 and the latest version GeCo3 have bigger codebases. They also provide features like the ussage of a neural network, which are of no help for this work. Since the file, providing arithmetic coding functionality, do not differ between all three versions, the first release was analyzed.
The header files, that this tool includes in `geco.c`, can be split into three categories: basic operations, custom operations and compression algorithms. The basic operations include header files for general purpose functions, that can be found in almost any c++ Project. The provided functionality includes operations for text-output on the command line inferface, memory management, random number generation and several calculations on up to real numbers.
Custom operations happens to include general purpose functions too, with the difference that they were written, altered or extended by GeCos developer. The last

category cosists of several C Files, containing implementations of two arithmetic coding implementations: **first** `bitio.c` and `arith.c`, **second** `arith_aux.c`.
The first two were developed by John Carpinelli, Wayne Salamonsen, Lang Stuiver and Radford Neal (is only mentioned in the latter). Comparing the two files, `bitio.c` has less code, shorter comments and much more not functioning code sections. Overall the conclusion would be likely that `arith.c` is some kind of official release, wheras `bitio.c` severs as a experimental file for the developers to create proof of concepts. The described files adapt code from Armando J. Pinho licenced by University of Aveiro DETI/IEETA written in 1999.
The second implementation was also licensed by University of Aveiro DETI/IEETA, but no author is mentioned. From interpreting the function names and considering the lenght of function bodys `arith_aux.c` could serve as a wrapper for basic functions that are often used in arithmetic coding.
Since original versions of the files licensed by University of Aveiro could not be found, there is no way to determine if the files comply with their originals or if changes has been made. This should be considered while following the static analysis.

Following function calls in all three files led to the conclusion that the most important function is defined as `arithmetic_encode` in `arith.c`. In this function the actual artihmetic encoding is executed. This function has no redirects to other files, only one function call `ENCODE_RENORMALISE` the remaining code consists of arithmetic operations only.

Following function calls int the `compressor` section of `geco.c`, to find the call of `arith.c` no sign of multithreading could be identified. This fact leaves additional optimization possibilities and will be discussed in **??**.

2.4.2. Samtools

BAM

Compression in this fromat is done by a implementation called BGZF, which is a block compression on top of a widely used algorithm called DEFLATE.

DEFLATE The DEFLATE compression algorithm combines LZ77 and huffman coding. It is used in well known tools like gzip. Data is split into blocks. Each block stores a header consisting of three bits. A single block can be stored in one of three forms. Each of which is represented by a identifier that is stored with the last two bits in the header.

- 00 No compression.
- 01 Compressed with a fixed set of Huffman codes.
- 10 Compressed with dynamic Huffman codes.

The last combination 11 is reserved to mark a faulty block. The third, leading bit is set to flag the last data block [30]. As described in **??** a compression with LZ77 results in literals, a length for each literal and pointers that are represented by the distance between pointer and the literal it points to. The LZ77 algorithm is executed before the huffman algorithm. Further compression steps differ from the already described algorithm and will extend to the end of this section.

Besides header bits and a data block, two Huffman code trees are store. One encodes literals and lenghts and the other distances. They happen to be in a compact form. This archived by a addition of two rules on top of the rules described in 2.3.3: Codes of identical lengths are orderd lexicographically, directed by the characters they represent. And the simple rule: shorter codes precede longer codes. To illustrated this with an example: For a text consisting out of C and G, following codes would be set for a encoding of two bit per character: C: 00, G: 01. With another character A in the alphabet, which would occour more often than the other two characters, the codes would change to a representation like this:

Symbol	Huffman code
A	0
C	10
G	11

Since A precedes C and G, it is represented with a 0. To maintain prefix-free codes, the two remaining codes are not allowed to start with a 0. C precedes G lexicographically, therefor the (in a numerical sense) smaller code is set to represent C. With this simple rules, the alphabet can be compressed too. Instead of storing codes itself, only the codelength stored [30]. This might seem unnecessary when looking at a single compressed bulk of data, but when compressing blocks of data, a samller alphabet can make a relevant difference.

BGZF extends this by creating a series of blocks. Each can not extend a limit of 64 Kilobyte. Each block contains a standard gzip file header, followed by compressed data.

CRAM

The improvement of BAM **cram-origin** called CRAM, also features a block structure [10]. The whole file can be seperated into four sections, stored in ascending order: File definition, a CRAM Header Container, multiple Data Container and a final CRAM EOF Container.
The File definition consists of 26 uncompressed bytes, storing formating information and a identifier. The CRAM header contains meta information about Data Containers and is optionally compressed with gzip. This container can also contain a uncompressed zero-padded section, reseved for SAM header information [10]. This saves time, in case the compressed file is altered and its compression need to be updated. The last container in a CRAM file serves as a indicator that the EOF is reached. Since in addition information about the file and its structure is stored, a maximum of 38 uncompressed bytes can be reached.
A Data Container can be split into three sections. From this sections the one storing the actual sequence consists of blocks itself, displayed in **??**IGURE as the bottom row.

- Container Header.
- Compression Header.
- A variable amount of Slices.
  - Slice Header.
  - Core Data Block.
  - A variable amount of External Data Blocks.

The Container Header stores information on how to decompress the data stored in the following block sections. The Compression Header contains information about what kind of data is stored and some encoding information for SAM specific flags. The actual data is stored in the Data Blocks. Those consist of encoded bit streams. According to the Samtools specification, the encoding can be one of the following:

External, Huffman and two other methods which happen to be either a form of huffman coding or a shortened binary representation of integers. The External option allows to use gzip, bzip2 which is a form of multiple coding methods including run length encoding and huffman, a encoding from the LZ family called LZMA or a combination of arithmetic and huffman coding called rANS.

Chapter 3

Environment and Procedure to Determine the State of The Art Efficiency and Compressionratio of Relevant Tools

Since improvements must be measured, defining a baseline which would need to be beaten bevorhand is necessary. Others have dealt with this task several times with common algorithms and tools, and published their results. But since the test case, that need to be build for this work, is rather uncommon in its compilation, the available data are not very useful. Therefore, new test data must be created.
The goal of this is, to determine a baseline for efficiency and effectivity of state of the art tools, used to compress DNA. This baseline is set by two important factors:

- Efficiency: **Duration** the Process had run for
- Effectivity: The difference in **Size** between input and compressed data

As a third point, the compliance that files were compressed losslessly should be verified. This is done by comparing the source file to a copy that got compressed and than decompressed again. If one of the two processes should operate lossy, a difference between the source file and the copy a difference in size should be recognizable.

3.1. Sever specifications and test environment

To be able to recreate this in the future, relevant specifications and the commands that reveiled this information are listed in this section.

Reading from /proc/cpuinfo reveals processor specifications. Since most of the information displayed in the seven entries is redundant, only the last entry is shown. Below are relevant specifications listed:

```
cat /proc/cpuinfo
```

- available logical processors: 0 - 7
- vendor: GenuineIntel
- cpu family: 6
- model nr, name: 58, Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
- microcode: 0x15
- MHz: 2280.874
- cache size: 8192 KB
- cpu cores: 4
- fpu and fpu exception: yes
- address sizes: 36 bits physical, 48 bits virtual

Full CPU secificaiton can be found in appendix.

The installed Random Access Memory (RAM) was offering a total of 16GB with four 4GB instances. For this paper relevant specifications are listed below: Command used to list

```
dmidecode --type 17
```

- Total/Data Width: 64 bits
- Size: 4GB
- Type: DDR3
- Type Detail: Synchronous
- Speed/Configured Memory Speed: 1600 Megatransfers/s

3.2. Operating System and Additionally Installed Packages

To leave the testing environment in a consistent state, not project specific processes running in the background, should be avoided. Due to following circumstances, a current Linux distribution was chosen as a suitable operating system:

- factors that interfere with a consistent efficiency value should be avoided
- packages, support and user experience should be present to an reasonable ammount

Some background processes will run while the compression analysis is done. This is owed to the demand of an increasingly complex operating system to execute complex programs. Considering that different tools will be exeuted in this environment, minimizing the background processes would require building a custom operating system or configuring an existing one to fit this specific use case. The boundary set by the time limitation for this work rejects named alternatives. Choosing **Debian GNU/Linux** version **11** features enough packages to run every tool without spending to much time on the setup.
The graphical user interface and most other optional packages were omitted. The only additional package added in the installation process is the ssh server package. Further a list of packages required by the compression tools were installed. At last, some additional packages were installed for the purpose of simplifying work processes and increasing the safety of the environment.

- installation process: ssh-server
- tool requirements:, git, libhts-dev, autoconf, automake, cmake, make, gcc, perl, zlib1g-dev, libbz2-dev, liblzma-dev, libcurl4-gnutls-dev, libssl-dev, libncurses5-dev, libomp-dev
- additional packages: ufw, rsync, screen, sudo

A complete list of installed packages as well as individual versions can be found in the appendix.

3.3. Selection, Receivement, and Preperation of Testdata

Following criteria is reqiured for test data to be appropriate:

- The test file is in a format that all or at least most of the tools can work with, meaning FASTA or FASTq files.
- The file is publicly available and free to use (for research).

A second, bigger set of testfiles were required. This would verify the test results are not limited to small files. A minimum of one gigabyte of average filesize were set as a boundary. This corresponds to over five times the size of the first set.
Since there are multiple open File Transfere Protocol (FTP) servers which distribute a variety of files, finding a suitable first set is rather easy. The ensembl database featured defined criteria, so the first available set called Homo_sapiens.GRCh38.dna.chromosome were chosen [31]. This sample includes over 20 chromosomes, whereby considering the filenames, one chromosome is contained in each single file. After retrieving and unpacking the files, write privileges on them was withdrawn. So no tool could alter any file contents. Finding a second, bigger set happened to be more complicated. FTP offers no fast, reliable way to sort files according to their size, regardless of their position. Since available servers **ftp-ensembl, ftp-ncbi, ftp-isgr!** offer several thousand files, stored in variating, deep directory structures, mapping filesize, filetype and file path takes too much time and resources for the scope of this work. This problematic combined with a easily triggered overflow in the samtools library, resulted in a set of several, manualy searched and tested FASTq files. Compared to the first set, there is a noticable lack of quantity, but the filesizes happen to be of a fortunate distribution. With pairs of two files in the ranges of 0.6, 1.1, 1.2 and one file with a size of 1.3 gigabyte, effects on scaling sizes should be clearly visible.

Following tools and parameters where used in this process:

```
\$ wget http://ftp.ensembl.org/pub/release-107/fasta/homo_sapiens/dna/
    Homo_sapiens.GRCh38.dna.chromosome.{2,3,4,5,6,7,8,9,10}.fa.gz
\$ gzip -d ./*
\$ chmod -w ./*
```

The chosen tools are able to handle the FASTA format. However Samtools must convert FASTA files into their SAM format bevor the file can be compressed. The compression will firstly lead to an output with BAM format, from there it can be compressed further into a CRAM file. For CRAM compression, the time needed for each step, from converting to two compressions, is summed up and displayed as one. For the compression time into the BAM format, just the conversion and the single compression time is summed up. The conversion from FASTA to SAM is not

displayed in the results. This is due to the fact that this is no compression process, and therefor has no value to this work.
Even though SAM files are not compressed, there can be a small but noticeable difference in size between the files in each format. Since FASTA should store less information, by leaving out quality scores, this observation was counterintuitive. Comparing the first few lines showed two things: the header line were altered and newlines were removed. The alteration of the header line would result in just a few more bytes. To verify, no information was lost while converting, both files were temporary stripped from metadata and formatting, so the raw data of both files can be compared. Using `diff` showed no differences between the stored characters in each file.

Chapter 4

Results and Discussion

The two tables A, A contain raw measurement values for the two goals, described in 3. The first table visualizes how long each compression procedure took, in milliseconds. The second one contains file sizes in bytes. Each row contains information about one of the files following this naming scheme:

`Homo_sapiens.GRCh38.dna.chromosome.x.fa`

To improve readability, the filename in all tables were replaced by `File`. To determine which file was compressed, simply replace the placeholder with the number following `File`.

4.1. Interpretation of Results

The units milliseconds and bytes store a high precision. Unfortunately they are harder to read and compare, solely by the readers eyes. Therefore the data was altered. Sizes in 4.1 are displayed in percentage, in relation to the respective source file. Meaning the compression with GeCo on:

Homo_sapiens.GRCh38.dna.chromosome.11.fa

resulted in a compressed file which were only 17.6% as big. Runtimes in 4.1 were converted into seconds and have been rounded to two decimal places. Also a line was added to the bottom of each table, showing the average percentage or runtime for each process.

Table 4.1.: File sizes in different compression formats in **percent**

ID.	GeCo %	Samtools BAM%	Samtools CRAM %
File 1	18.32	24.51	22.03
File 2	20.15	26.36	23.7
File 3	19.96	26.14	23.69
File 4	20.1	26.26	23.74
File 5	17.8	22.76	20.27
File 6	17.16	22.31	20.11
File 7	16.21	21.69	19.76
File 8	17.43	23.48	21.66
File 9	18.76	25.16	23.84
File 10	20.0	25.31	23.63
File 11	17.6	24.53	23.91
File 12	20.28	26.56	23.57
File 13	19.96	25.6	23.67
File 14	16.64	22.06	20.44
File 15	79.58	103.72	92.34
File 16	19.47	25.52	22.6
File 17	19.2	25.25	22.57
File 18	19.16	25.04	22.2
File 19	18.32	24.4	22.12
File 20	18.58	24.14	21.56
File 21	16.22	22.17	19.96
Total	21.47	28.24	25.59

Overall, Samtools BAM resulted in 71.76% size reduction, the CRAM methode improved this by rughly 2.5%. GeCo provided the greatest reduction with 78.53%. This gap of about 4% comes with a comparatively great sacrifice in time.

Table 4.2.: Compression duration in seconds

ID.	GeCo	Samtools BAM	Samtools CRAM
File 1	23.5	3.786	16.926
File 2	24.65	3.784	17.043
File 3	2.016	3.123	13.999
File 4	19.408	3.011	13.445
File 5	18.387	2.862	12.802
File 6	17.364	2.685	12.015
File 7	15.999	2.503	11.198
File 8	14.828	2.286	10.244
File 9	12.304	2.078	9.21
File 10	13.493	2.127	9.461
File 11	13.629	2.132	9.508
File 12	13.493	2.115	9.456
File 13	99.902	1.695	7.533
File 14	92.475	1.592	7.011
File 15	85.255	1.507	6.598

File 16	82.765	1.39	6.089
File 17	82.081	1.306	5.791
File 18	79.842	1.277	5.603
File 19	58.605	0.96	4.106
File 20	64.588	1.026	4.507
File 21	41.198	0.721	3.096
Total	42.57	2.09	9.32

As 4.1 is showing, the average compression duration for GeCo is at 42.57s. That is a little over 33s, or 78% longer than the average runtime of samtools for compressing into the CRAM format.

Since CRAM requires a file in BAM format, the third row is calculated by adding the time needed to compress into BAM with the time needed to compress into CRAM. While SAM format is required for compressing a FASTA into BAM and further into CRAM, in itself it does not features no compression. However, the conversion from SAM to FASTA can result in a decrease in size. At first this might be contra intuitive since, as described in 2.2.1 SAM stores more information than FASTA. This can be explained by comparing the sequence storing mechanism. A FASTA sequence section can be spread over multiple lines whereas SAM files store a sequence in just one line, converting can result in a SAM file that is smaller than the original FASTA file. Before interpreting this data further, a quick view into development processes: GeCo stopped development in the year 2016 while Samtools is being developed since 2015, to this day, with over 70 people contributing.

For the second set of testdata, the file identifier was set to follow the scheme `File 2.x` where x is a number between zero and seven. While the first set of testdata had names that matched the file identifiers, considering its numbering, the second set had more variating names. The mapping between identifier and file can be found in **??**. Reviewing 4.1 one will notice, that GeCo reached a runtime over 60 seconds on every run. Instead of displaying the runtime solely in seconds, a leading number followed by an m indicates how many minutes each run took.

Table 4.3.: File sizes in different compression formats in **percent**

ID.	GeCo %	Samtools BAM%	Samtools CRAM %
File 1	1.00	6.28	5.38
File 2	0.98	6.41	5.52
File 3	1.21	8.09	7.17
File 4	1.20	7.70	6.85

File 5	1.08	7.58	6.72
File 6	1.09	7.85	6.93
File 7	0.96	5.83	4.63
Total 1.07	7.11	6.17	

Table 4.4.: Compression duration in seconds

ID.	**GeCo**	**Samtools BAM**	**Samtools CRAM**
File 1	1m58.427	16.248	23.016
File 2	1m57.905	15.770	22.892
File 3	1m09.725	07.732	12.858
File 4	1m13.694	08.291	13.649
File 5	1m51.001	14.754	23.713
File 6	1m51.315	15.142	24.358
File 7	2m02.065	16.379	23.484
Total	1m43.447	13.474	20.567

In both tables 4.1 and 4.1 the already identified pattern can be observed. Looking at the compression ratio in 4.1 a maximum compression of 99.04% was reached with GeCo. In this set of test files, file seven were the one with the greatest size (1̃.3 Gigabyte). Closely folled by file one and two (1̃.2 Gigabyte).

4.2. View on Possible Improvements

So far, this work went over formats for storing genomes, methods to compress files (in mentioned formats) and through tests where implementations of named algorithms compress several files and analyzed the results. The test results show that GeCo provides a better compression ratio than Samtools and takes more time to run through. So in this testrun, implementations of arithmetic coding resulted in a better compression ratio than Samtools BAM with the mix of huffman coding and LZ77, or Samtools custom compression format CRAM. Comparing results in [3], supports this statement. This study used FASTA/Multi-FASTA files from 71MB to 166MB and found that GeCo had a variating compression ratio from 12.34 to 91.68 times smaller than the input reference and also resulted in long runtimes up to over 600 minutes [3]. Since this study focused on another goal than this work and therefore used different test variables and environments, the results can not be compared. But what can be taken from this, is that arithmetic coding, at least in GeCo is in need of

a runtime improvement.
The actual mathematical proove of such an improvemnt and its implementation can not be covered because it would to beyond scope. But in order to set up a foundation for this task, the rest of this work will consist of considerations and problem analysis, which should be thought about and dealt with to develop a improvement.

S.V. Petoukhov described his findings about the distribution of nucleotides [32]. With the probability of one nucleotide, in a sequence of sufficient length, information about the direct neighbours is revealed. For example, with the probability of `C`, the probabilities for sets (n-plets) of any nucleotide `N`, including `C` can be determined without counting them [32].

Considering this and the meassured results, an improvement in the arithmetic coding process and therefore in GeCos efficiency, would be a good start to equalize the great gap in the compression duration. Combined with a tool that is developed with todays standards, there is a possibility that even greater improvements could be archived.
How would a theoretical improvement approach look like? As described in 2.3.2, entropy coding requires to determine the probabilies of each symbol in the alphabet. The simplest way to do that, is done by parsing the whole sequence from start to end and increasing a counter for each nucleotide that got parsed. With new findings discovered by Petoukhov in cosideration, the goal would be to create an entropy coding implementation that beats current implementation in the time needed to determine probabilities. A possible approach would be that the probability of one nucleotide can be used to determine the probability of other nucelotides, by a calculation rather than the process of counting each one. This approach throws a few questions that need to be answered in order to plan a implementation [32]:

- How many probabilities are needed to calculate the others?
- Is there space for improvement in the parsing/counting process?
- How can the variation between probabilities be determined?

The question for how many probabilities are needed, needs to be answered, to start working on any kind of implementation. This question will only get answered by theoretical proove. It could happen in form of a mathematical equtaion, which prooves that counting all ocurences of one nucleotide reveals can be used to determin all probabilities. Since this task is time and resource consuming and there is more to discuss, finding a answer will be postponed to another work.

The Second point must be asked, because the improvement in counting only one nucleotide in comparison to counting three, would be to little to be called relevant. Especially if multithreading is a option. Since in the static codeanalysis in **??** revealed no multithreading, the analysis for improvements when splitting the workload onto several threads should be considered, before working on an improvement based on Petoukhovs findings. This is relevant, because some improvements, like the one described above, will loose efficiency if only subsections of a genomes are processed. A tool like OpenMC for multithreading C programs would possibly supply the required functionality to develop a prove of concept [32], [33]. But how could a improvement look like, not considering possible difficulties multithreading would bring? To answer this, first a mechanism to determine a possible improvement must be determined. To compare parts of a programm and their complexity, the Big-O notation is used. Unfortunally this is only covering loops and coditions as a whole. Therefore a more detailed view on operations must be created: Considering a single threaded loop with the purpose to count every nucleotide in a sequence, the process of counting can be split into several operations, defined by this pseudocode.

```
while (sequence not end)
do
    next_nucleotide = read_next_nucleotide(sequence)
    for (element in alphabet_probabilities)
    do
        if (element equals next_nucleotide)
            element = element + 1
        fi
    done
done
```

This loop will itterate over a whole sequence, counting each nucleotide. In line three, a inner loop can be found which itterates over the alphabet, to determine which symbol should be increased. Considering the findings, described above, the inner loop can be left out, because there is no need to compare the read nucleotide against more than one symbol. The Big-O notation for this code, with any sequence with the length of n, would be decreseased from O(n^2) to O($n \cdot 1$) or simply O(n) [34]. Which is clearly an improvement in complexety and therefor also in runtime. The runtime for calculations of the other symbols probabilities must be considered as well and compared against the nested loop to be certain, that the overall runtime was improved.

Getting back to the question how multithreading would impact improvements: A implementation like the one described above, could also work with multithreading. Since the ratio of the difference between O(n^2) and O(n) does not differ with the reduction of n. Multiple threads, processing parts of a sequence with the length of n, would also benefit, because any fraction of n^2 will always be greater than the corresponding fraction of n. This results can either sumed up for global probabilities or get used individually on each associated subsequence. Either way, the presented improvement approach should be appliable to both parsing methods.

This leaves a list of problems, which needs to be regarded in the approach of developing a improvement. If there space for improvement in the parsing/counting process, what problems needs to be addressed:

- reducing one process by adding aditional code must be estimated and set into relation.
- for a tool that does not feature multithreading, how would multithreading affect the improvement reulst?

A important question that needs answered would be: If Petoukhovs findings show that, through simliarities in the distribution of each nucleotide, one can lead to the aproximation of the other three. Entropy codings work with probabilities, how does that affect the coding mechanism? With a equal probability for each nucleotide, entropy coding can not be treated as a whole. This is due to the fact, that huffman coding makes use of differing probabilities. A equal distribution means every character will be encoded in the same length which would make the encoding process unnecessary. Arithmetic coding on the other hand is able to handle equal probabilities. The fact that there are obviously chains of repeating nucleotides in genomes. For example `File 2.2`, which contains this subsequence is found at line 90:

```
AAAAAAAAAAAAAAAAAAAAAATAAATATTTTATTT
```

Without determining probabilities, one can see that the amount of `A`s outnumbers `T`s and neither `C` nor `G` are present. With the whole 1.2 gigabytes, the distribution will align more, but by cutting out a subsection, of relevant size, with unequal distributions will have an impact on the probabilities of the whole sequence. If a greater sequence would lead to a more equal distribution, this knowledge could be used to help determining distributions on subsequences of one with equaly distributed probabilities.

List of Abbreviations

ASCII	American Standard Code for Information Interchange
BAM	Binary Alignment Map
CRAM	Compressed Reference-oriented Alignment Map
DNA	Deoxyribonucleic Acid
EOF	End of File
FASTA	File Format for Storing Genomic Data
FASTq	File Format Based on FASTA
FTP	File Transfere Protocol
GeCo	Genome Compressor
IUPAC	International Union of Pure and Applied Chemistry
LZ77	Lempel Ziv 1977
RAM	Random Access Memory
SAM	Sequence Alignment Map

List of Tables

4.1. Compression Effectivity 27
4.2. Compression Efficiency 28
4.3. Compression Effectivity for greater files 29
4.4. Compression Effectivity for greater files 30

A.1. Compression Efficiency xiii
A.2. Compression Effectivity xiii

List of Figures

2.1. A superficial representation of the physical positioning of genomes. Showing a double helix (bottom), a chromosome (upper rihgt) and a chell (upper center). 3

2.2. A purely diagrammatic figure of the components DNA is made of. The smaller, inner rods symbolize nucleotide links and the outer ribbons the phosphate-sugar chains [2]. 4

2.3. SAM/BAM file structure example 8

2.4. Schematic diagram of a general communication system by Shannons definition. [23] 11

2.5. Illustrative rescaling in arithmetic coding process. [27] 14

Listings

Bibliography

[1] Eva Bianconi, Allison Piovesan, Federica Facchin, *et al.*, "An estimation of the number of cells in the human body", *Annals of Human Biology*, vol. 40, no. 6, pp. 463–471, Jul. 2013. DOI: 10.3109/03014460.2013.807878.

[2] J. D. WATSON and F. H. C. CRICK, "Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid", *Nature*, vol. 171, no. 4356, pp. 737–738, Apr. 1953. DOI: 10.1038/171737a0.

[3] Morteza Hosseini, Diogo Pratas, and Armando Pinho, "A survey on data compression methods for biological sequences", *Information*, vol. 7, no. 4, p. 56, Oct. 2016. DOI: 10.3390/info7040056.

[4] ISO Central Secretary, "Mpge-g", en, International Organization for Standardization, Standard ISO/IEC 23092:2019, 2019. [Online]. Available: https://www.iso.org/standard/23092.html.

[5] Claudio Albert, Tom Paridaens, Jan Voges, *et al.*, "An introduction to MPEG-g, the new ISO standard for genomic information representation", Sep. 2018. DOI: 10.1101/426353.

[6] Andrew D. Johnson, "An extended IUPAC nomenclature code for polymorphic nucleic acids", *Bioinformatics*, vol. 26, no. 10, pp. 1386–1389, Mar. 2010. DOI: 10.1093/bioinformatics/btq098.

[7] Paul Flicek. "Ensembl project". (Oct. 24, 2022), [Online]. Available: http://www.ensembl.org/.

[8] Santa Cruz UCSC - University of California. "Ucsc genome browser". (Oct. 28, 2022), [Online]. Available: https://genome.ucsc.edu/ (visited on 10/28/2022).

[9] "Global alliance for genomics and health". (Oct. 10, 2022), [Online]. Available: https://github.com/samtools/hts-specs..

[10] The SAM/BAM Format Specification Working Group. "Sequence alignment/map format specification". version 44b4167. (Aug. 22, 2022),

[Online]. Available: https://github.com/samtools/hts-specs (visited on 09/12/2022).

[11] Petr Danecek, James K Bonfield, Jennifer Liddle, *et al.*, “Twelve years of SAMtools and BCFtools”, *GigaScience*, vol. 10, no. 2, Jan. 2021. DOI: 10.1093/gigascience/giab008.

[12] UCSC University of California Sata Cruz, Ed. “Twobit file format”. (Sep. 22, 2022), [Online]. Available: https://genome-source.gi.ucsc.edu/gitlist/kent.git/raw/master/src/inc/twoBit.h.

[13] Diogo Pratas, Armando J. Pinho, and Paulo J. S. G. Ferreira, “Efficient compression of genomic sequences”, in *2016 Data Compression Conference (DCC)*, IEEE, Mar. 2016. DOI: 10.1109/DCC.2016.60.

[14] Kelvin V. Kredens, Juliano V. Martins, Osmar B. Dordal, *et al.*, “Vertical lossless genomic data compression tools for assembled genomes: A systematic literature review”, *PLOS ONE*, vol. 15, no. 5, Rashid Mehmood, Ed., e0232942, May 2020. DOI: 10.1371/journal.pone.0232942.

[15] Anas Al-Okaily, Badar Almarri, Sultan Al Yami, and Chun-Hsi Huang, “Toward a better compression for DNA sequences using huffman encoding”, *Journal of Computational Biology*, vol. 24, no. 4, pp. 280–288, Apr. 1, 2017. DOI: 10.1089/cmb.2016.0151.

[16] Peter J. A. Cock, Christopher J. Fields, Naohisa Goto, Michael L. Heuer, and Peter M. Rice, “The sanger FASTQ file format for sequences with quality scores, and the solexa/illumina FASTQ variants”, *Nucleic Acids Research*, vol. 38, no. 6, pp. 1767–1771, Dec. 2009. DOI: 10.1093/nar/gkp1137.

[17] Illumina. “Illumina fastq file structure explained”. (Nov. 17, 2022), [Online]. Available: https://support.illumina.com/bulletins/2016/04/fastq-files-explained.html.

[18] K. Simonsen and, “Character mnemonics and character sets”, RFC 1345, Jun. 1992.

[19] Manish RajShivare, Yogendra P. S. Maravi, and Sanjeev Sharma, “Analysis of header compression techniques for networks: A review”, *International Journal of Computer Applications*, vol. 80, no. 5, pp. 13–20, Oct. 2013. DOI: 10.5120/13856-1701.

[20] Kashfia Sailunaz, Mohammed Rokibul Alam Kotwal, and Mohammad Nurul Huda, “Data compression considering text files”,

International Journal of Computer Applications, vol. 90, no. 11, pp. 27–32, Mar. 2014. DOI: 10.5120/15765-4456.

[21] Alistair Moffat, “Huffman coding”, *ACM Computing Surveys*, vol. 52, no. 4, pp. 1–35, Jul. 2020. DOI: 10.1145/3342555.

[22] Alistair Moffat, Radford M. Neal, and Ian H. Witten, “Arithmetic coding revisited”, *ACM Transactions on Information Systems*, vol. 16, no. 3, pp. 256–294, Jul. 1998. DOI: 10.1145/290159.290162.

[23] C. E. Shannon, “A mathematical theory of communication”, *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, Jul. 1948. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[24] Hans Delfs and Helmut Knebl, *Introduction to Cryptography, Principles and Applications (Information Security and Cryptography)*. Springer, 2007, p. 368.

[25] J. J. Rissanen, “Generalized kraft inequality and arithmetic coding”, *IBM Journal of Research and Development*, vol. 20, no. 3, pp. 198–203, May 1976. DOI: 10.1147/rd.203.0198.

[26] “Ieee standard for floating-point arithmetic”, *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019. DOI: 10.1109/IEEESTD.2019.8766229.

[27] Ian H. Witten, Radford M. Neal, and John G. Cleary, “Arithmetic coding for data compression”, *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, Jun. 1987. DOI: 10.1145/214762.214771. [Online]. Available: https://doi.org/10.1145/214762.214771.

[28] L. Peter Deutsch, Jean-Loup Gailly, Mark Adler, L. Peter Deutsch, and Glenn Randers-Pehrson, “Gzip file format specification version 4.3”, RFC Editor, RFC 1952, May 1996, http://www.rfc-editor.org/rfc/rfc1952.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1952.txt.

[29] David A. Huffman, “A method for the construction of minimum-redundancy codes”, *Proceedings of the Institute of Radio Engineers*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952.

[30] L Peter Deutsch, “DEFLATE compressed data format specification version 1.3”, Tech. Rep., May 1996. DOI: 10.17487/rfc1951. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1951.

[31] “Ensembl rapid release”. (Oct. 15, 2022), [Online]. Available: https://ftp.ensembl.org.

[32] Sergey V. Petoukhov, “Tensor rules in the stochastic organization of genomes and genetic stochastic resonance in algebraic biology”, Oct. 2021. DOI: 10.20944/preprints202110.0093.v1.

[33] Michael J. Quinn, *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[34] Firdous Mala and Rouf Ali, “The big-o of mathematics and computer science”, vol. 6, pp. 1–3, Jan. 2022. DOI: 10.26855/jamc.2022.03.001.

Appendix A

Erster Anhang: Lange Tabelle

Table A.1.: Compression duration meassured in milliseconds

ID.	GeCo	Samtools BAM	Samtools CRAM
File 1	235005	3786	16926
File 2	246503	3784	17043
File 3	20169	3123	13999
File 4	194081	3011	13445
File 5	183878	2862	12802
File 6	173646	2685	12015
File 7	159999	2503	11198
File 8	148288	2286	10244
File 9	12304	2078	9210
File 10	134937	2127	9461
File 11	136299	2132	9508
File 12	134932	2115	9456
File 13	999022	1695	7533
File 14	924753	1592	7011
File 15	852555	1507	6598
File 16	827651	1390	6089
File 17	820814	1306	5791
File 18	798429	1277	5603
File 19	586058	960	4106
File 20	645884	1026	4507
File 21	411984	721	3096

Table A.2.: File sizes in different compression formats

ID.	Source File	GeCo	Samtools CRAM
File 1	253105752	46364770	55769827
File 2	136027438	27411806	32238052
File 3	137338124	27408185	32529673
File 4	135496623	27231126	32166751
File 5	116270459	20696778	23568321
File 6	108827838	18676723	21887811
File 7	103691101	16804782	20493276
File 8	91844042	16005173	19895937
File 9	84645123	15877526	20177456
File 10	81712897	16344067	19310998
File 11	59594634	10488207	14251243
File 12	246230144	49938168	58026123
File 13	65518294	13074402	15510100

File 14	47488540	7900773	9708258
File 15	51665500	41117340	47707954
File 16	201600541	39248276	45564837
File 17	193384854	37133480	43655371
File 18	184563953	35355184	40980906
File 19	173652802	31813760	38417108
File 20	162001796	30104816	34926945
File 21	147557670	23932541	29459829