hochschule mannheim

Comparison of Compression Tools for Biological Data and Analysis of Possible Optimization

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. Dr. Elena Fimmel

Prof. Dr. Markus Gumbel

Eichelkraut, Gabriel:
Comparison of Compression Tools for Biological Data and Analysis of Possible Optimization / Gabriel Eichelkraut. –
Bachelor Thesis, Mannheim: University of Applied Sciences Mannheim, 2022. 42 pages.

Eichelkraut, Gabriel:
Vergleich von Kompressionswerkzeugen für biologische Daten und Analyse von Verbesserungsmöglichkeiten / Gabriel Eichelkraut. –
Bachelor-Thesis, Mannheim: Hochschule Mannheim, 2022. 42 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

Gabriel Eichelkraut

Abstract

Comparison of Compression Tools for Biological Data and Analysis of Possible Optimization

A variety of algorithms is used to compress sequenced DNA. New findings in the patterns of how the building blocks of DNA are distributed, might provide a chance to improve long used compression algorithms. The comparison of four widely used compression methods and an analysis on their implemented algorithms, leads to the conclusion that improvements are feasible. The closing discussion provides insights, to possible optimization approaches and which challenges they might involve.

Vergleich von Kompressionswerkzeugen für biologische Daten und Analyse von Verbesserungsmöglichkeiten

Verschiedene Algorithmen werden verwendet, um sequenzierte DNA zu speichern. Eine neue Entdeckung darüber wie die Bausteine der DNA angeordnet sind, bietet die Möglichkeit vorhandene Kompressionsmethoden zum Speichern von sequenzierten DNA zu verbessern.

Diese Arbeit vergleicht vier weit verbreitete Kompressionsmethoden und analysiert deren Verwendung von Algorithmen. Durch die Ergebnisse lässt sich der Schluss ziehen, dass Verbesserungen in der Implementation von arithmetischer Codierung möglich sind. Die abschließende Diskussion betrachtet mögliche Vorgehensweisen zur Verbesserung und welche Aufgaben diese mit sich ziehen könnten.

Contents

Chapter 1

Introduction

Understanding the biological code of living things-, is an alsways developing task which plays a significant role in multiple aspects of our lives. The results of research in this area provides knowledge that helps development in the medical sector, in agriculture and more [20], [28], [44]. Getting insights into the biological code is possible through storing and studying information, embedded in genonmes [45]. Since life is complex, there is a lot of information, that requires a lot of memory space [15], [30].
Compression algorithms and their implementation has helped towards resolving the problem of storing information. Compressed data requires less space and therefore less time to be transported over networks [40]. This advantage is scalable and, since genetic information needs a lot of storage even in a compressed state, improvements are welcomed [27]. Since this field is relatively new compared to others, such as computer theory, which created the foundation for compression algorithms, there is much to discover and new findings are not unusual [7], [33], [40]. From some of these findings, new tools can be developed. In general they focus on increasing at least one of two factors: the speed at which data is compressed and the compression ratio, meaning the difference between uncompressed and compressed data [27], [30], [40].

New discoveries in the universal rules of the stochastical organization of genomes might provide a base for new algorithms and therefore new tools or an improvement of existing ones for genome compression [31]. The aim of this work is to analyze the current state of the art for compression tools for biological data and implemented probabilistic algorithms. Furthermore this work will determine if there is room for optimization.

The discussion will include a superficial analysis of how and where this new approach could be implemented and what problems possibly need to be taken care of in the process.

To reach a common ground, the first pages will give the reader a quick overview on the structure of human DNA. This will include explanations for some basic terms, used in biology and computer science. The first step into the theory of genome compression will be taken, by describing differences in common file formats, used to store genome information. From there, a section which is relevant for understanding compression will follow. It will analyze differences between compression approaches, go over some history of coding theory and lead to a deeper look into the fundamentals of state of the art compression algorithms. The chapter will end with a few pages about implementations of compression algorithms in relevant tools.

In order to measure an optimization, a baseline must be set. Therefore, the efficiency and effectivity of suitable state of the art tools will be measured. To be as precise as possible, the middle part of this work focuses on setting up an environment, picking input data, installing and executing tools and finally meassuring and documenting the results.

These results compared with the understanding of how the tools work, will show if there is the need of an improvement and on what factor it should focus. The end of this work will be used to discuss the properties of a possible optimization, how feasibility could be determined and which problems such a project would need to overcome.

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 of the structure of any living organism.

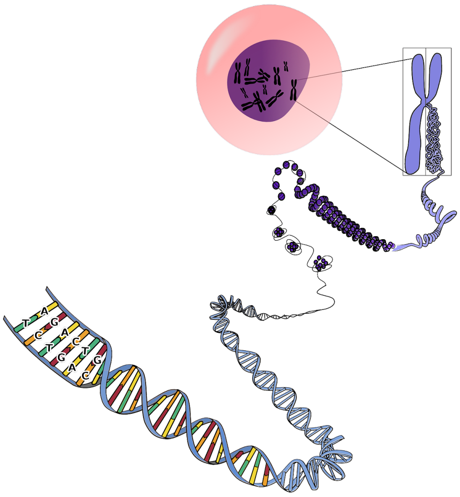


Figure 2.1.: A superficial representation of the physical positioning of genomes. Showing a double helix (bottom), a chromosome (upper rihgt) and a chell (upper center).

All living organisms, like plants and animals, are made of cells. To get a rough impression, a human body can consist of several trillion cells. A cell in itself is the smallest living organism. Most cells consist of an outer section and a core which is a called nucleus. In 2.1 the nucleus is illustrated as a purple, circlelike scheme inside a lighter circle. The nucleus contains chromosomes. Those chromosomes contain genetic information, about their organism in form of Deoxyribonucleic Acid (DNA) [5].

DNA is often seen in the form of a double helix, as shown in 2.2. A double helix consists, as the name suggests, of two single helixes [45].

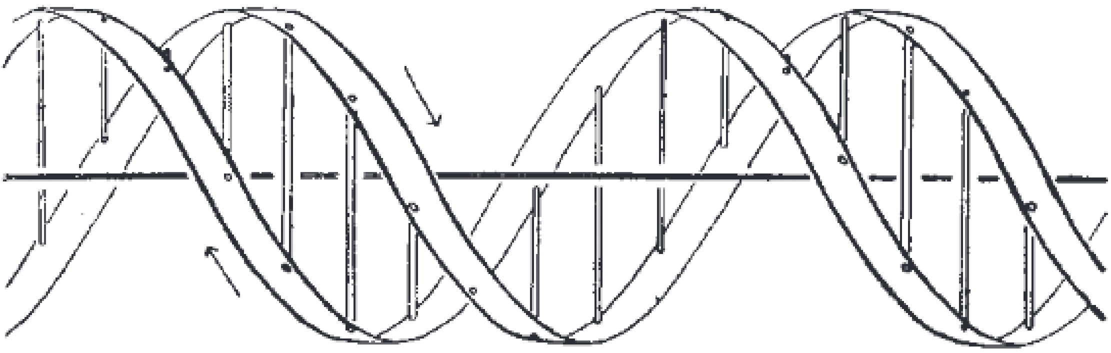


Figure 2.2.: A purely diagrammatic figure of the components DNA is made of. The smaller, inner rods symbolize nucleotide links and the outer ribbons the phosphate-sugar chains [45].

Each of them consists of two main components: the sugar phosphate backbone, which is not relevant for this work and the bases. The suggar phosphate backbones are illustrated as flat stripes, circulating aroung the horizontal line in 2.2. Pairs of bases are symbolized as vertical bars between the suggar phosphates. The arrangement of Bases represents the information, stored in the DNA. Whar is here described as base is a organic molecule, which is also called nucleotide [45].

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 a 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 need to be stored physically to save the information of the whole DNA. 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 initial. So, the example would change to `AGA` in one helix, `TCT` in the other.

This representation is 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 amount 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) [3] table is a character set registered in 1975, and to this day it is still in use to encode texts digitally. To solve communication issues, larger character sets replaced ASCII some fields. It is still used in situations where storage is short [12].

The buildingblocks of DNA require a minimum of four letters, so at least two bits are needed. Storing a single *A* with ASCII encoding, requires 8 bit. Since there are at least 2^8 or 128 displayable symbols with ASCII encoding, this leaves a great overhead of unused combinations.

In most tools, more than four symbols are used. This is due to the complexity in sequencing DNA. This process is not 100% preceice, so additional symbols are used to mark nucleotides that could not or could only partly get determined. Furthermore a so called quality score is used to indicate the certainty of correct sequencing for each nucleotide [6], [15].

More common everyday-usage text encodings like unicode require 16 bits per letter. So, settling with ASCII has improvement capabilities and 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 noticeable ones would be [15]:

- Sequence variation
- Aligned data
- Sequenced reads

The categories are listed in descending order, based on their complexity, considering their usecase and data structure. Starting with sequence variation, called haplotype which describes formats storing graph based structures that focus on analyzing variations in different genomes [7], [22]. Sequenced reads focus on storing continuous nucleotide chains from a sequenced genome [15]. Aligned data is somewhat 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 [7], [15]. The focus of this work is the compression of sequenced data but not the likelihood of how accurate the data might be. Therefore, only formats that are able to store sequenced reads will be worked with. Note that some algigned data formats are also able to store aligned reads, since latter is just a less informative representation of the first [7], [15].

Several people and groups have developed different file formats to store genomes. Unfortunately, the only standard for storing genomic data is fairly new [2], [4]. 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 the following criteria:

- The format has reputation. This can be indicated through:
  - A scientific paper, that proved its superiority to other relevant tools.
  - A broad usage of the format determined by its use on ftp servers, which focus on supporting scientific research.
- The format should not specialize on only one type of DNA or target a specific technology.
- The format stores nucleotide sequences and does not necessarily include International Union of Pure and Applied Chemistry (IUPAC) codes besides A, C, G and T [19].
- The format is open source. Otherwise, optimizations cannot be tested without buying the software and/or requesting permission to disassemble and reverse engineer the software or parts of it.

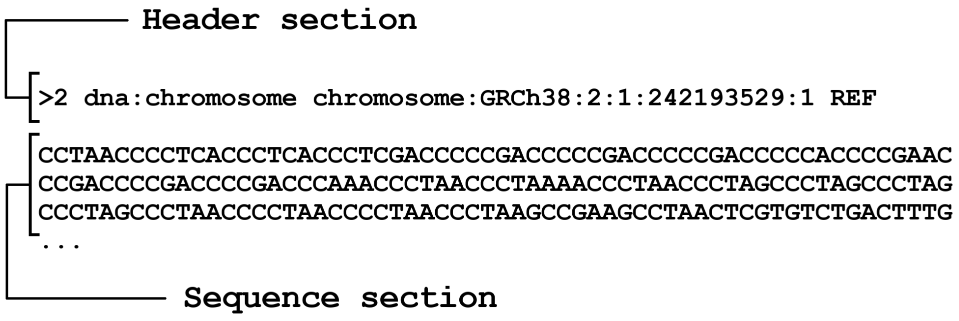
Information on available formats was gathered through various Internet platforms [42], [43] and scientific papers [6], [7], [15]. Some common file formats are:

- File Format for Storing Genomic Data (FASTA)/Multi-FASTA
- File Format Based on FASTA (FASTq) [6]
- Sequence Alignment Map (SAM)/Binary Alignment Map (BAM) [7], [39]
- Compressed Reference-oriented Alignment Map (CRAM) [7], [39]

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 an unoficial standard for sequenced reads [7], [13], [15], [21], [33].
Considering the first criteria, by searching through anonymously accessible FTP servers, only two formats are used commonly: FASTA or its extension FASTq and the BAM Format [11], [18], [29].

2.2.1. FASTA and FASTq

The rather simple FASTA format, is widely used when it comes to storing sequenced reads, without a quality score [7], [15]. Since it is an uncompressed format, FASTA files are often transmitted compressed with an external tool like gzip [11], [29].



```
>2 dna:chromosome chromosome:GRCh38:2:1:242193529:1 REF
CCTAACCCCTCACCCTCACCCTCGACCCCCGACCCCCGACCCCCGACCCCCACCCCGAAC
CCGACCCCGACCCCGACCCAAACCCTAACCCTAAAACCCTAACCCTAGCCCTAGCCCTAG
CCCTAGCCCTAACCCCTAACCCCTAACCCTAAGCCGAAGCCTAACTCGTGTCTGACTTTG
...
```

Figure 2.3.: Edited example of a FASTA file. The original was received from the ensemble server [11].

The 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 [6], [30]. 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 the character `A, C, G or T`. There are 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 [19].

The second section can have multiple lines of sequences. A similar format is the Multi-FASTA file format, it consists of concatenated FASTA files.[15].

In addition to its predecessor, FASTq files contain a quality score. The file content consists of four sections, where 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 [6]:

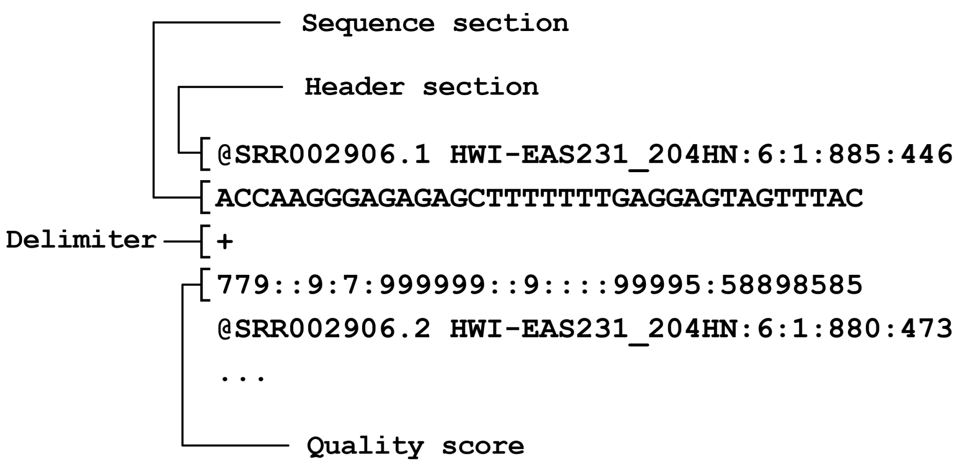
- Line 1: Sequence identifier aka. Title, starting with an @ and an optional description.
- Line 2: The sequence consisting of nucleoids, symbolized by A, T, G and C.
- Line 3: A '+' that functions as a separator or delimitier. Optionally followed by the content of line 1.
- Line 4: Quality line(s). 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 [6].

In 2.3 the described structure is illustrated. The sequence and the delmitier section were altered, to illustrate the stucture of this format better. In the header section, `SRR002906.1` is the sequence identifier; the text that follows is a description. In the delimiter line, the header section without the leading @ could be written again. The last line shows the header for the second sequence.

2.2.2. Sequence Alignment Map

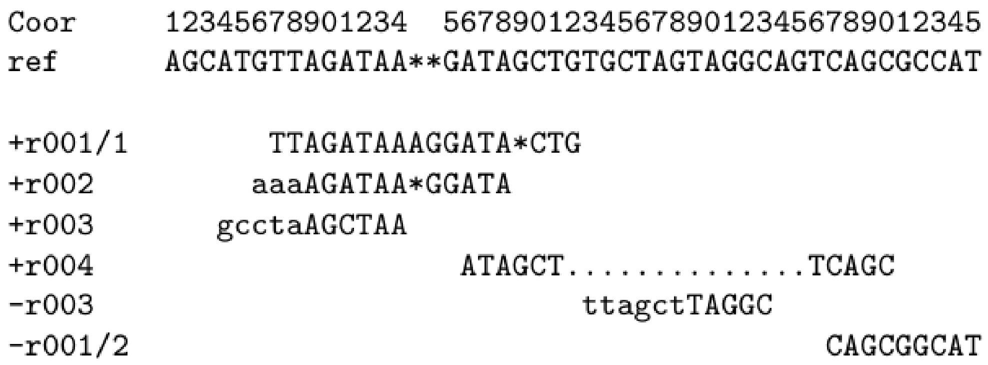
SAM often seen in its compressed, binary representation BAM with the fileextension `.bam`, is part of the SAMtools package, a utility tool for processing SAM/BAM



```
@SRR002906.1 HWI-EAS231_204HN:6:1:885:446
ACCAAGGGAGAGAGCTTTTTTTGAGGAGTAGTTTAC
+
779::9:7:999999::9::::99995:58898585
@SRR002906.2 HWI-EAS231_204HN:6:1:880:473
...
```

Figure 2.4.: Altered example of a FASTq file. The original was received from ncbi server [29].

and CRAM files. The SAM/BAM file is a text based format delimited by the whitespace character called tabulation or **tab** for short [7]. It uses 7-bit US-ASCII; to be precise charset ANSI X3.4-1968 [41]. 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 2.5.: SAM file structure example [39].

Compared to FASTA SAM and further compression forms, store more information. As displayed in 2.5 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 [15]. 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 [7] or at [39].

Samtools provide the feature to convert a FASTA file into SAM format. Since there is no way to calculate the mentioned additional information from the information stored in FASTA, the converted files only store two lines. The first line stores meta-data 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 [24].

In many cases, like in gene compressing, the compression is idealy lossless. This means, it is possible to receive the full information that was available in the origin data by decompressing any kind of compressed data. Lossy compression on the other hand might exclude parts of data in the compression process, in order to increase the compression rate. The excluded parts are typically not necessary to transmit the original information. This works with certain audio and picture files, or with network protocols like Universal Datagram Protocol (UDP) which are used to transmit video/audio streams live [32], [35].

For storing 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 exact position are needed for the sequence to be complete and useful.

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 bit can store up to 2^{11} combinations of bit and a 1 GB drive can store no more than 1 GB of data. Information on the other hand is limited by the way it is stored. What exactly defines information, depends on multiple factors. The context in which information is transmitted and the source and destination of the information. This can be in form of a signal, transferred from one entity to another, or information that is persisted, so it can be obtained at a later point in time.

For the scope of this work, information will be seen as the type and position of nucleotides, sequenced from DNA. To be even more precise, it is a chain of characters from an alphabet of `A, C, G, and T`, since this is the *de facto* standard for digital persistence of DNA [2]. When it comes to storing capabilities, the boundaries of information, can be illustrated by using the example mentioned above. A drive with the capacity of 1 GB 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 [1]. 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 losing any information.

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 acquired reputation, are described in detail below [26], [27], [30], [38].

2.3.1. Dictionary Coding

Dictionary coding, as the name suggest, uses a dictionary to eliminate redundant occurrences 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 separate dictionary entries. Which way is more efficient depends on the text that should get compressed. The dictionary should only store strings that occur in the input data. Also storing a dictionary in addition to the (compressed) input data, would be a waste of resources. Therefore the dictionary is part of the text. Each first occurrence is left uncompressed. Each occurrence of a string, after the first one, points either to to its first occurrence or to the last replacement of its occurrence. Which method is used depends on the algorithm.

2.6 illustrates how this process is executed. The bar on top of the figure, which extends over the full width, symbolizes any text. The squares inside the text are repeating occurrences of text segments. In the dictionary coding process, the square annotated as `first occ.` is added to the dictionary. `Second` and `third occ.` get replaced by a structure `<pos, len>` consisting of a pointer to the position of the first occurrence `pos` and the length of that occurrence `len`. The bar at the bottom of the figure shows how the compressed text for this example would be structured. The dotted lines would only consist of two bytes, storing position and lenght, pointing to `first occ.`. Decompressing this text would only require parsing the text from left to right and to replace every `<pos, len>` with the already parsed word from the dictionary. This means jumping back to the parsed position stored in the replacement, reading for as long as the length dictates, copying the read section, jumping back and pasting the section.

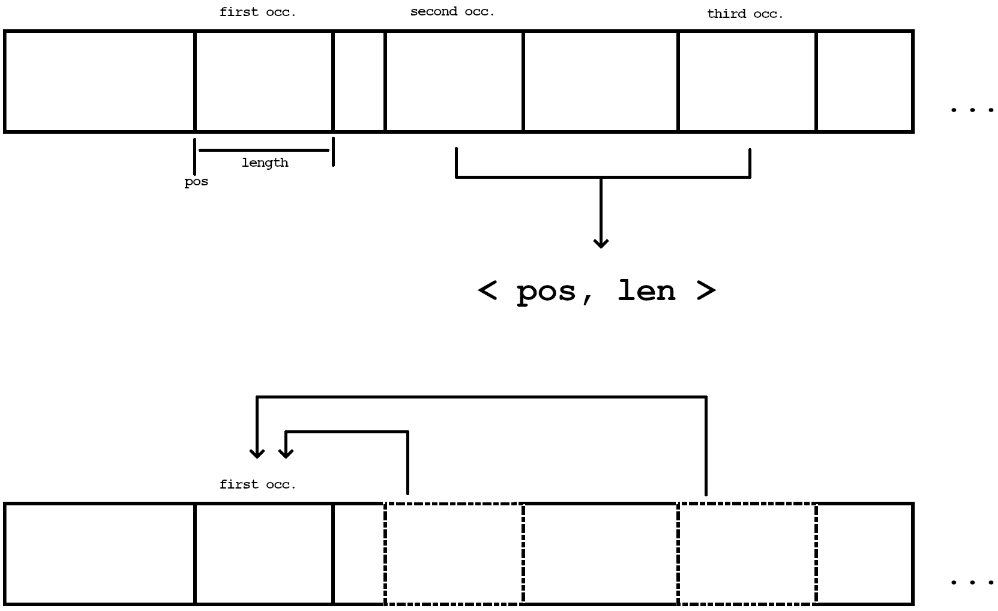


Figure 2.6.: Schematic sketch, illustrating the replacement of multiple occurrences done in dictionary coding.

The LZ Family

The computer scientist Abraham Lempel and the electrical engineer Jacob Ziv created multiple algorithms that are based on dictionary coding. They can be recognized by the substring `LZ` in their name; like `LZ77 and LZ78` which are short for Lempel Ziv 1977 and 1978 [47]. The number at the end indicates when the algorithm was published. Today, members of the LZ family are widely used in compression implementations like rar, zip, gzip and bz2 [10]. Some of those are also used to compress DNA.

LZ77 basically works, by removing all repetitions of a string or substring and replacing them with information where to find the first occurrence and how long it is. The distance between the first occurrence and a replacement is limited, because each pointer has a static amount of storage available. A pointer, length pair is typically stored in two bytes. One bit is reseverd to indicate that the next 15 bit are a position, lenght pair. More than 8 bit are available to store the pointer and the rest is reserved for storing the length. Exact amounts depend on the implementation [9], [47].

Unfortunally, 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 strengthen the general understanding of compression algortihms and because it is a part of hybrid coding implementations, this section remains.

2.3.2. Shannons Entropy

The founder of information theory Claude Elwood Shannon described entropy and published his work in 1948 [40]. Here, he focused on transmitting information. His theorem is applicable to almost any form of communication signal. His findings are not only useful for forms of information transmission.

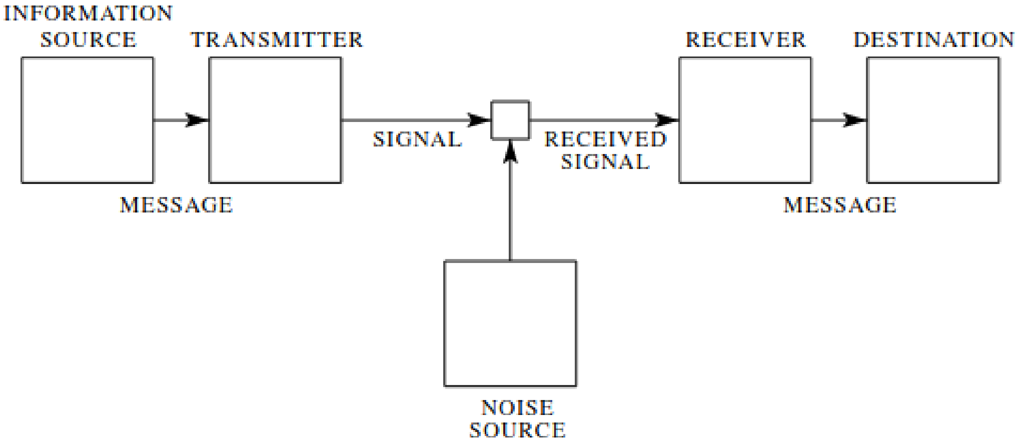


Figure 2.7.: Schematic diagram of a general communication system by Shannons definition. [40]

Altering 2.7 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, it is possible that both are represented by the same physical actor.
Transmitter and receiver would be changed to compression/encoding and decompression/decoding. Inbetween those two, there is no signal but instead any period of time [40].

Shannon's Entropy provides a formula 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 a probability experiment over X. Every state that actually occurs, while executing the experiment generates information which is measured in binary digits *bits* for short with the part of the equation displayed in 2.2 [8], [40]:

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

2.3.3. Arithmetic coding

This coding method is an approach to solve the problem of wasting memory due to the overhead which is created by encoding certain lengths of alphabets in binary [27], [37]. For example: Encoding a three-letter alphabet requires at least two bit per letter. Since there are four possilbe combinations with two bit, 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 was a possibility to encode more than one letter in two bit.
Dr. Jorma Rissanen described arithmetic coding in a publication in 1976 [37]. 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 [37].

Before getting into the arithmetic coding algorithm, the following section will go over some details on how digital fractions are handled by computers. This knowledge will be helpful in understanding how arithmetic coding works.

In computers, arithmetic operations on floating point numbers are processed with integer representations [17]. The number 0.4 for example would be represented by $4 \cdot 10^{-1}$.

An interval would be represented by natural numbers between 0 and 100 and $... \cdot 10^{-}x$. x starts with the value 2 and grows as the integers grow in length; meaning only if a uneven number is divided. For example: Dividing an 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 negative 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 [27], [46].

Binary fractions are limited in form of representing decimal fractions. This is due to the fact that every other digit adds zero or half of the value before. In other terms: $b \cdot 2^{-n}$ determines the value of $b \in 0, 1$ at position n behind the decimal point.

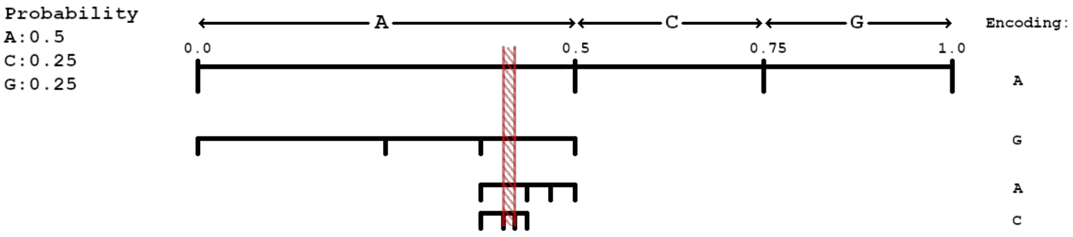


Figure 2.8.: Illustrative example of arithmetic coding.

The encoding of the input text, or a sequence is possible by projecting it on a binary encoded fraction between 0 and 1. To get there, each character in the alphabet is represented by an interval between two fractions, in the space between 0.0 and 1.0. In 2.8 this space is illustrated by the line in the upper center, with a scaling from 0.0 on the left, to 1.0 on the right side. The interval for each symbol is determined by its distribution, in the input text (interval start) and the start of the next character (interval end). The sum of all intervals will result in one [27].

In order to remain in a presentable range, the example in 2.8 uses an alphabet of only three characters: `A, C and G`. For the sequence `AGAC` a probability distribution as shown in the upper left corner and listed in 2.3.3 was calculated. The intervals resulting from these probabilities are visualized by the three sections marked by outwards pointing arrows at the top. The interval for A extends from 0.0 until the start of C at 0.5, which extends to the start of G at 0.75 and so on.

Table 2.1.: Probabilities for A,C and G as shown in example 2.8

Symbol	Probability	Interval
A	$\frac{2}{4} = 0.11$	$[0.0, 0.5)$
C	$\frac{1}{4} = 0.71$	$[0.5, 0.75)$
G	$\frac{1}{4} = 0.13$	$[0.75, 1.0)$

In the encoding process, the first symbol read from the sequence determines a interval that its symbol is associated with. Every following symbol determines a subinterval, which is formed by subdividing the previous interval into sections proportional to the probabilities from 2.3.3. Starting with A, the most left interval in 2.8 is subdivided into intervals visualized below. Leaving an available space of $[0.0, 0.5)$. From there, the interval representing G is subdivided, and so on until the last symbol C is processed. This leaves an interval of $[0.40625, 0.421275)$. This is marked in 2.8 with a red line. Since the interval is comparably small, in the illustration it seems like a point in the interval is marked. This is not the case, the red line shows the position of the last mentioned interval.
To store the encoding result in as few bits as possible, only a single number between the upper and the lower end of the last interval will be stored. To encode in binary, the binary floating point representation of any number inside the interval for the last character is calculated.
In this example, the number 0.41484375 in decimal, or 0.0110101 in binary, would be calculated.
To summarize the encoding process in short [27], [46]:

- 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 chosen.
- This process is repeated until an interval for the last character is determined.

- A binary floating point number is determined wich lays in between the interval that represents the last symbol.

For the decoding process to work, the End of File (EOF) symbol must be present as the last symbol in the text. The compressed file will store the probabilities of each alphabet symbol as well as the floatingpoint number. The decoding process executes in a similar procedure as the encoding. The stored probabilities 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 probabilities associated with the intervals onto the alphabet, the origin text can be read [27], [37], [46].

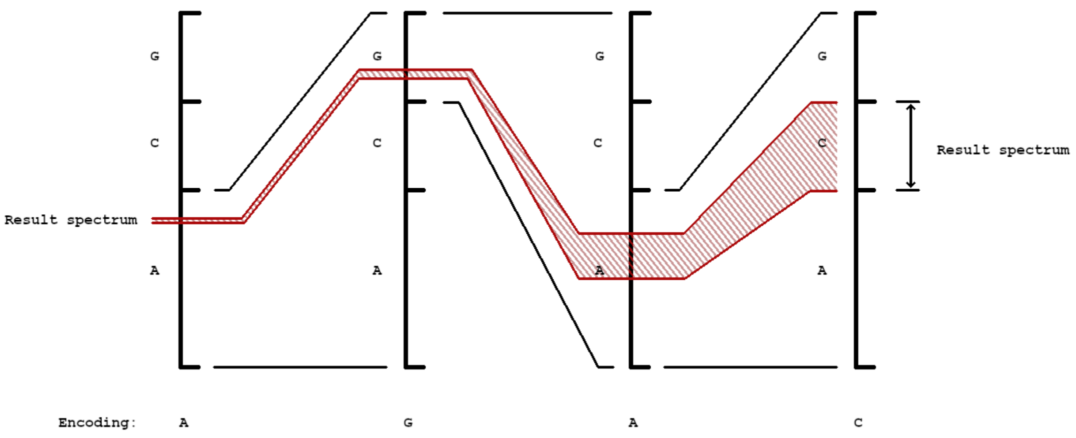


Figure 2.9.: Illustrative rescaling in arithmetic coding process.

The described coding is only feasible on machines with infinite precision [46]. As soon as finite precision comes into play, the algorithm must be extended, so that a certain length in the resulting number will not be exceeded. This is due to the fact that digital datatypes are limited in their capacity for example, the unsigned 64-bit integers which can store up to $2^64 - 1$ bit or any number between 0 and 18,446,744,073,709,551,615. That might seem like a great amount at first, but considering a unfavorable alphabet that extends the results lenght by one on each symbol that is read, only sequences with the length of 63 can be encoded (62 if EOF is exclued) [27]. For the compression with finite percission, rescaling is used. This method works by scaling up the intervals which result from subdividing. The up-scaling process is illustrated in 2.9. The vertical lines illustrate the interval of each step. The smaller, black lines between them indicate which previous section was scaled up. The red lines indicate the final interval and the letters at the bottom indicate which symbol gets encoded in this step.

2.3.4. Huffman Encoding

D. A. Huffman's work focused on finding a method to encode messages with a minimum of redundance. He referenced a coding procedure developed by Shannon and Fano, named after its developers, which worked similar. The Shannon-Fano coding is not used today due to the superiority of Huffman's algorithm in both efficiency and effectivity [27].

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 [10].

Compression with the Huffman algorithm also provides a solution to the problem, described at the beginning of 2.3.2; of waste through unused bit 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 bit used per symbol in a message. This is possible by setting individual bit lengths for symbols used in the text that should get compressed [16]. 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 each individual symbols will be encoded. The binary tree will be constructed after following guidelines [30]:

- 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 weigh. The weight is defined by the frequency the symbol occurs 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 is the probability, that this node is read next in the symbol sequence.

- Pairs of the lowest weighting nodes are formed. This pair will from there on be represented by a node which weight is equal to the sum of the weight of its child nodes.
- Higher weighting nodes are positioned left, lower ones right.

An often-mentioned difference between Shannon-Fano and Huffman coding is that the first is working top down while the latter is working bottom up. Meaning the first Shannon-Fano is starting with the highest probabilities while Huffman starts with the lowest [26], [30].

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` [16].

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

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 tuple 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 [16], [26].

For this example a four letter alphabet, containing `A, C, G and T` will be used. For this alphabet, the binary representation encoded in ASCII is listed in the second column of 2.3.4. 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 bit.

Table 2.2.: ASCII Codes and probabilities for `A,C,G and T`

Symbol	ASCII Code	Probability	Occurences
A	0100 0001	$\frac{11}{100} = 0.11$	11
C	0100 0011	$\frac{71}{100} = 0.71$	71
G	0101 0100	$\frac{13}{100} = 0.13$	13
T	0000 1010	$\frac{5}{100} = 0.05$	5

The exact input text is not relevant, since only the resulting probabilities are needed. To make this example more illustrative, possible occurrences are listed in the most right column of 2.3.4. The probability for each symbol is calculated by dividing the message length by the times the symbol occured. This and the resulting probabilities on a scale between 0.0 and 1.0, for this example are shown in 2.3.4 [16]. Creating a tree is done bottom-up. In the first step, for each symbol from the alphabet, a node without any connection is formed .

```
<A>, <T>, <C>, <G>
```

Starting with the two lowest weightened symbols, a node is added to connect both. With the added, blank node the count of available nodes got reduces 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>`.

```
<A, T>, <C>, <G>
```

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.

```
<G, <A, T> >, <C>
```

The `<G, <A, T> >` has a probability of 0.29. Adding the last, highest weightened node `C` results in a root node with the probability of 1.0. For a better understanding of this example, and to help further explanations, the resulting tree is illustrated in 2.10.

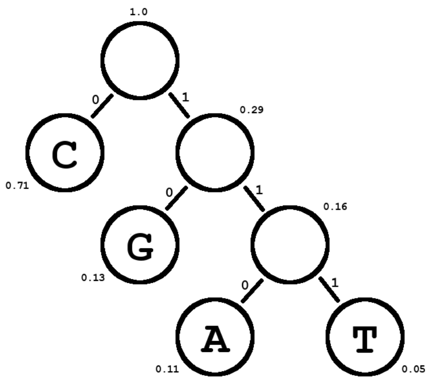


Figure 2.10.: Final version of the Huffman tree for described example.

As illustrated in 2.10 the 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 can be seen in the second column of 2.3.4.

Table 2.3.: Huffman codes for `A,C,G and T`

Symbol	Huffman Code	Occurences
A	100	11
C	0	71
G	11	13
T	101	5

Since high weightened and therefore often occuring leafs are positioned to the left, short paths lead to them and so only few bit 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 bit per encoded symbol. In this example the text would require over one bit less storage for every second symbol [16].

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 [6], [7].

2.4. Implementations in Relevant Tools

This section should give the reader a overview, how a small variety of compression tools implement described compression algorithms. It is written with the goal to compensate a problem that ocurs in scientific papers, and sometimes in technical specifications for programs. They often lack information on the implementation, in a satisfying dimension [7], [33], [39].
The information on the following pages was received through static code analysis. Meaning the comprehension of a programs behaviour or its interactions due to the analysis of its source code. This is possible because the analysed tools are openly published and licenced under GNU General Public License (GPL) v3 [33] and Massachusetts Institute of Technology (MIT)/Expat [39], which permits the free use for scientific purposes [14], [25].

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 [36]. 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 numbers from natural to real.
Custom operations happens to include general purpose functions too, with the dif-

ference that they were written, altered or extended by GeCos developer. The last category cosists of several C Files, containing implementations of two arithmetic coding algorithms: **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. 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 [36].

While following function calls in the `compressor` section of `geco.c`, to find the locations where `arith.c` gets executed, no sign of multithreading could be identified. This fact leaves additional optimization possibilities.

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 bit. 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 bit 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 [9].
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 bit 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 is achieved 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 occur 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 by a 0. To maintain prefix-free codes, the two remaining codes are not allowed to contain a leading 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 [9]. 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 [13] called CRAM, also features a block structure [39]. The whole file can be separated into four sections, stored in ascending order: File definition, a CRAM Header Container, multiple Data Container and a final CRAM EOF Container.
The complete structure is displayed in 2.11. The following paragrph will give a brief description to the high level view of a CRAM fiel, illustrated as the most upper bar. Followed by a closer look at the data container, which components are listed in the bar, at the center of 2.11. The most in deph explanation will be given to the bottom bar, which shows the structure of so called slices.

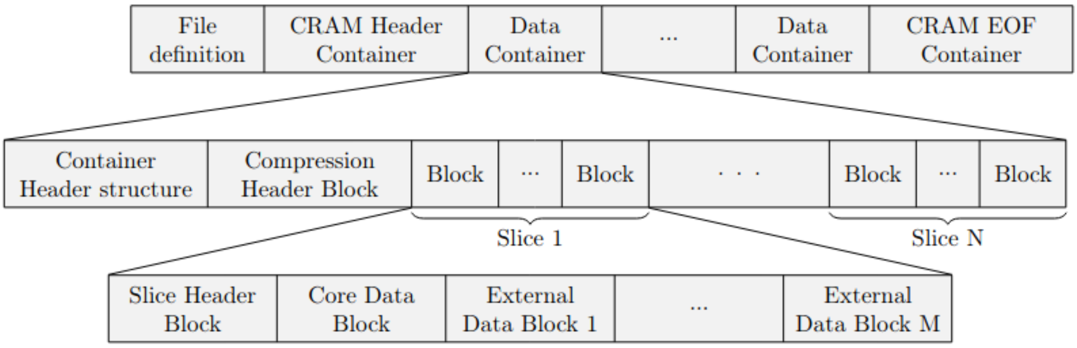


Figure 2.11.: File Format Structure of Samtools CRAM [39].

The File definition, illustrated on the left side of the first bar in 2.11, 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 [39]. 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 2.11 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 [39]. 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 [39]. 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 [7].

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. Server 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

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

- Total/Data Width: 64 bits
- Size: 4 GB
- 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, non-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 amount.

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 mentioned 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`

was picked [11]. This sample includes 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, without sufficient permission. 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 [11], [18], [29] offer several thousand files, stored in varying, 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.

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 tables B.2 and B.1 contain raw measurement values for the two goals, described in this 3 earlyer section. The table B.1 lists how long each compression procedure took, in milliseconds. B.2 contains file sizes in bytes. In these tables, as well as in the other ones associated with tests, a consistent naming scheme is used, to improve readability. The filenames were replaced by `File` followed by two numbers separated by a point. For the first test set, the number prefix `1.` was used, the second set is marked with a `2.`. For example, the fourth file of each test, in tables are named like this `File 1.4` and `File 2.4`. The name of the associated source file for the first set is:

`Homo_sapiens.GRCh38.dna.chromosome.4.fa`

Since the source files of the second set are not named as consistent as in the first one, a third column in 4.2 was added, which is mapping table identificator (ID.) and source file name.
The files contained in each test set, as well as their size can be found in the tables 4.1 and 4.2. The first test set contained a total of 2.8 GB unevenly spread over 21 files, while the second test set contained 7 GB in total, with a quantity of seven files.

Table 4.1.: Files contained in the First Test Set and their Sizes in MB

ID.	Size in MB
File 1.1	241.38
File 1.2	234.823
File 1.3	192.261
File 1.4	184.426
File 1.5	176.014
File 1.6	165.608

File 1.7	154.497
File 1.8	140.722
File 1.9	134.183
File 1.10	129.726
File 1.11	130.976
File 1.12	129.22
File 1.13	110.884
File 1.14	103.786
File 1.15	98.888
File 1.16	87.589
File 1.17	80.724
File 1.18	77.927
File 1.19	56.834
File 1.20	62.483
File 1.21	45.289

Table 4.2.: Files contained in the Second Test Set, their Sizes in MB and Source File Names

ID.	Size in MB	Source File Name
File 2.1	1188.976	SRR002905.recal.fastq
File 2.2	1203.314	SRR002906.recal.fastq
File 2.3	627.467	SRR002815.recal.fastq
File 2.4	676.0	SRR002816.recal.fastq
File 2.5	1066.431	SRR002817.recal.fastq
File 2.6	1071.095	SRR002818.recal.fastq
File 2.7	1240.564	SRR002819.recal.fastq

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 4.3 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 4.4 were converted into seconds and have been rounded to two decimal places. Also a line was added to the bottom of each table, after which the average percentage of runtime for each process is displayed.

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

ID.	GeCo %	Samtools BAM%	Samtools CRAM %
File 1.1	18.32	24.51	22.03
File 1.2	20.28	26.56	23.57
File 1.3	20.4	26.58	23.66
File 1.4	20.3	26.61	23.56
File 1.5	20.12	26.46	23.65
File 1.6	20.36	26.61	23.6
File 1.7	19.64	26.15	23.71
File 1.8	20.4	26.5	23.67
File 1.9	17.01	23.25	20.94
File 1.10	20.15	26.36	23.7
File 1.11	19.96	26.14	23.69
File 1.12	20.1	26.26	23.74
File 1.13	17.8	22.76	20.27
File 1.14	17.16	22.31	20.11
File 1.15	16.21	21.69	19.76
File 1.16	17.43	23.48	21.66
File 1.17	18.76	25.16	23.84
File 1.18	20.0	25.31	23.63
File 1.19	17.6	24.53	23.91
File 1.20	19.96	25.6	23.67
File 1.21	16.64	22.06	20.44
Total	18.98	24.99	22.71

Overall, Samtools BAM resulted in a little over 75% size reduction, the CRAM methode improved this by rughly 2%. GeCo provided the greatest reduction with over 80%. This difference of about 6% comes with a comparatively great sacrifice in time.

Table 4.4.: Compression duration in seconds

ID.	GeCo	Samtools BAM	Samtools CRAM
File 1.1	23.5	3.786	16.926
File 1.2	24.65	3.784	17.043
File 1.3	2.016	3.123	13.999
File 1.4	19.408	3.011	13.445
File 1.5	18.387	2.862	12.802
File 1.6	17.364	2.685	12.015
File 1.7	15.999	2.503	11.198
File 1.8	14.828	2.286	10.244
File 1.9	12.304	2.078	9.21
File 1.10	13.493	2.127	9.461
File 1.11	13.629	2.132	9.508

File 1.12	13.493	2.115	9.456
File 1.13	99.902	1.695	7.533
File 1.14	92.475	1.592	7.011
File 1.15	85.255	1.507	6.598
File 1.16	82.765	1.39	6.089
File 1.17	82.081	1.306	5.791
File 1.18	79.842	1.277	5.603
File 1.19	58.605	0.96	4.106
File 1.20	64.588	1.026	4.507
File 1.21	41.198	0.721	3.096
Total	42.57	2.09	9.32

As 4.4 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 source file into BAM and further into CRAM, in itself it is a format without 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 is able to contain more information about a sequence 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. Reviewing the second test set in 4.6 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 4.5.: File sizes in different compression formats in **percent**

ID.	GeCo%	Samtools BAM%	Samtools CRAM%
File 2.1	1.00	6.28	5.38
File 2.2	0.98	6.41	5.52
File 2.3	1.21	8.09	7.17
File 2.4	1.20	7.70	6.85
File 2.5	1.08	7.58	6.72
File 2.6	1.09	7.85	6.93
File 2.7	0.96	5.83	4.63
Total	1.07	7.11	6.17

Table 4.6.: Compression duration in seconds

ID.	GeCo	Samtools BAM	Samtools CRAM
File 2.1	1m58.427	16.248	23.016
File 2.2	1m57.905	15.770	22.892
File 2.3	1m09.725	07.732	12.858
File 2.4	1m13.694	08.291	13.649
File 2.5	1m51.001	14.754	23.713
File 2.6	1m51.315	15.142	24.358
File 2.7	2m02.065	16.379	23.484
Total	1m43.447	13.474	20.567

In both tables 4.6 and 4.5 the already identified pattern can be observed. Samtools compression is faster but less effective. Looking at the compression ratio in 4.5 a maximum compression of 99.04% was reached with GeCo. In this set of test files, `File 2.7` were the one with the greatest size (1̃.3 GB). Closely folled by file one and two (1̃.2 GB).

4.2. View on Possible Improvements

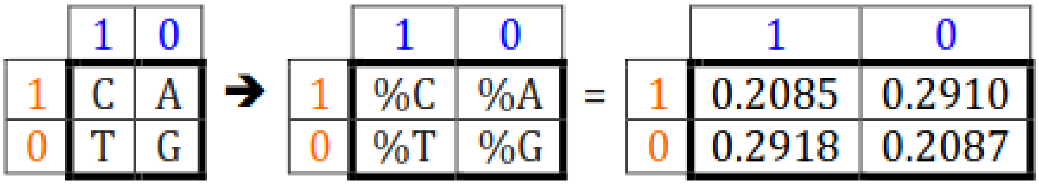
So far, this work went over formats for storing genomes, algorithms that are used to compress those genomes and through tests that compared efficiency and effectivity of mentioned algorithms. The test results show that GeCo provides a better compression ratio than Samtools but takes more time to run 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 format called CRAM. Comparing this against the results in [15], supports this statement. This study used sets of FASTA/Multi-FASTA files from 71 MB to 166 MB per file 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 [15]. Since this study focused on another goal and therefore used different test variables and environments, the results can not be compared directly. 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 prove of such an optimization, the planing of the implementation and the development of a proof of concept, will be a rewarding but time and ressource comsuming project. In order to widen the foundation for this tasks, 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 prepublished findings, which are under ongoing research, about the distribution of nucleotides in [31]. There he found that the probabilities of nucleotides in certain chromosomes align. Also, the probability of one nucleotide reveals estimations about the direct neighbours of each occurrence of the nucleotide. This can be illustrated by the following formula in which the probability of `C` is known and `N` is a placeholder for any of the four nucleotides [31]:

$$\texttt{\% C} \approx \sum\texttt{\%CN} \approx \sum\texttt{\%NC} \approx \sum\texttt{\%CNN} \approx \sum\texttt{\%NCN} \approx \sum\texttt{\%NNC} \approx \sum\texttt{\%CNNN} \approx \sum\texttt{\%NCNN} \approx \sum\texttt{\%NNCN} \approx \sum\texttt{\%NNNC} \ \ldots$$

With the probability of `C`, the probabilities for sets (n-plets) of `N` as a placeholder for any nucleotide of `A, C, G or T`, and including at least one `C` might be determinable without counting them [31]. So, $\sum$`%CN` consists of `%CC + %CA + %CG + %CT`. The elements in each sum get more, with a increasing n in the n-plet. To be precise $4^{|N|}$ describes the growing of combinations.



	1	0
1	C	A
0	T	G

➔

	1	0
1	%C	%A
0	%T	%G

=

	1	0
1	0.2085	0.2910
0	0.2918	0.2087

Figure 4.1.: Probabilities for `A, C, G and T` in `Homo sapiens chromosome 1, GRCh38.p14 Primary Assembly` [29], [31].

The exemplaric probabilities Petoukhov displayed are reprinted in 4.1. Noteable are the similarities in the distirbution of %A and %T as well as in %G and %C. They align until the third digit after the decimal point. According to him, this regularity is found in the genome of humans, some anmials, plants, bacteria and more [31]. Considering this and the measured 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 more current tool it is

possible that even greater improvements could be achived.
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 would be parsing the whole sequence from start to end and increasing a counter for each nucleotide that got parsed. With Petoukhov assumptions 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 for that could lay in determining the probabilities of all nucleotides from one by a calculation rather than counting each one. This approach throws a few questions that need to be answered in order to plan a implementation [31]:

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

The first question must be answered before considering the others. Since counting one instead of four elements is not a significant difference, the possibility that the change would be to little to be relevant, must be taken in consideration. Additionally, since in the static codeanalysis in 2.4 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 assumptions. 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 [31], [34]. The question for how many probabilities must be determined only gets answered by a theoretical prove. It could happen in form of a mathematical equation, which proves that counting all occurrences of one nucleotide type can be used to determin probabilities of the other nucleotides. 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. 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 and an alphabet length of four, would be decreseased from O($n \cdot 4$) to O($n \cdot 1$) or simply O(n) [23]. Which is clearly an improvement in complexety and therfore might result in a decreade 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.

Should Petoukhovs rules, and the regularity shown in 4.1 happen to be universal, three approaches could be used to determine the probability of a genome:

- No counting of any nucleotide and using a fixed set of probabilities.
- Counting only one nucleotide and determining the others by calculation.
- Counting either A and T or G and C and determining the other two by mirroring the results.

The calculation mentioned in the second bulletpoint, would look like the following, if for example the probability for `C` got determined by parsing the whole sequence:

$$
\begin{aligned}
&\%G \approx \%C \\
&\%A + \%T \approx \%100 - (\%G + \%C) \\
&\%A \approx \tfrac{\%100-(\%G+\%C)}{2} \\
&\%T \approx \%A
\end{aligned}
$$

The mapping, mentioned in the last point would consist of the first and last line of the example above.
Working with probabilities, in fractions as well as in percent, would probabily mean rounding values. To increase the accuracity, the actual value resulting form the counted symbol could be used. For this to work, the amount of overall symbols had to be determined, for $\%A + \%T$ to be calculateable. Since counting all symbols during the process of the one needed nucleotide, could have an impact on the runtime, the full length could also be calculated. With s beeing the size of the parsed sequence in bytes and c beeing the bytes per character $\frac{s}{c}$ would result in the amount of symbols in the sequence.

The described implementations could even work with a multithreaded parsing process. Improvements would not be impacted 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 simply n. The results can either be sumed up for global probabilities or get used individually on the associated subsequence a thread worked on. Either way, the presented improvement approach should be appliable to both parsing methods.
After that, some problems are left which needs to be regarded in the approach of developing an improvement. The last bulletpoint referes to the possibility that Petoukhovs findings will show that the simliarities in the probability distribution is univeral. Entropy codings work with probabilities, how would a similarity affect the coding mechanism? With an 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 less usefull. Arithmetic coding on the other hand is able to result in compression even with eaqual probabilites. But even though the whole chromosome might show a

certain pattern, its subsequences mostly do not. For example `File 1.10`, which contains this subsequence:

```
AACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTAACCCTTAACCC
```

Without determining probabilities, one can see that the amount of `C`s outnumbers `T`s and `A`s. With the whole 133258320 symbols 130 MB, the probability distribution will align more. The following values have been roundet down: `A` $\approx$ `0.291723`, `C` $\approx$ `0.207406`, `G` $\approx$ `0.208009`, `T` $\approx$ `0.2928609`. The pattern described by S. Petoukhov is recognizable. 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.

There are some rules that apply to any whole chromosom sequence as well as to subsequences referenced by `S`. With the knowledge about lenght `|S|` and the frequency and position of one symbol e.g. `C` represented as `|C|`, rules about the enveloping sequence can be derived. The arithmetic operations on symbols $\cdot$ for consecutive repetitions and $+$ for the concatination are used. For x and y as the ammount of nucleotides before the first and after the last `C` applies:

- $\frac{|S|}{x/y-1} \cdot (|C| - 1)$ determines the ammount of $(x \cdot N) + C$ and $C + (y \cdot N)$ sequences $\in S$.
- The longest chain starting with `C` is $C + N \cdot (|S| - x - 1)$.
- The longest chain ending with `C` is $(|S| - y - 1) \cdot N + C$.
- There are $(|C| - 1)$ occurrences of $(x + 1) \cdot N + C$ and an equal ammount of $C + N \cdot (y + 1)$.

Those statements might seem trivial to some, but possibly help other to clarify the boundaries on Petoukhov's rules. Also, they represent the end of the thought process of this works last section.

Before resulting in a final conclusion, a quick summary of important points:

- Coding algorithms did not change drastically, in the last deccades.
- Improvements are achived by additions to existing algorithms and the combination of several algorithms for specific tasks.

- Tests and comparings shown that arithmetic coding lacks in efficiency.

The goal for this new optimization approach is clearly defined. Also a possible test environment and measurement techniques that would indicate a success have been tested, in this work as well as in cited works [15]. Considering how other improvements were implemented in the past shows that the way an approach like described above is feasible [27]. This, combined with the last point leads to assumption that there is a realistic chance to optimize entropy coding, specifically the arithmetic coding algorithm.
This assumption will consolidate by viewing best- and worst-case szenarios that could result from further research. Two important future events are taken into consideration. One would be the theoretical prove of an working optimization approach and the other if Petoukhov's findings develop favorable: The best case would be described as optimization through exact determination of the whole probability distribution is possible and Petoukhov's findings prove that his rules are universal for genomes between living organisms. This would result in a faster compression in entropy coding. Depending on the dimension, either a tool that is implementing entropy coding only or a hybrid tool, with improved efficiency in its entropy coding algorithms would define a new `state of the art`.
In a worst case szenario, the exact determination of probability distributions would not be possible. This would mean more research should be done in approximating probability distibutions. Additionally, how the use of $A \approx T \approx 0.2914$ and $G \approx C \approx 0.2086$ could provide efficiency improvements in reference-free compression of whole chromosomes and general improvements in the compression of a reference genome in reference-based compression solutions [15].
Also in this szenario Petoukov would be wrong about the universality of the defined rules, considering the exemplary caculation of probability determination of `File 1.10` a concern that his rules do not apply to any genomes or that he had a miscalculation is out of the way. This would limit the range of the impact an improvement would create. The combination of which genomes follow Petoukov's rules and a list of tools that specialize on the compression of those would set the new goal for an optimization approach.

So, how favorable research turns out does not determine if there will be an impact but just how far it will reach.

List of Abbreviations

ANSI	American National Standard Insitute
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
GB	Gigabyte
MB	Megabyte
GeCo	Genome Compressor
GPL	GNU General Public License
IUPAC	International Union of Pure and Applied Chemistry
LZ77	Lempel Ziv 1977
MIT	Massachusetts Institute of Technology
RAM	Random Access Memory
SAM	Sequence Alignment Map
UDP	Universal Datagram Protocol
UTF	Unicode Transformation Format

List of Tables

List of Figures

Bibliography

[1] I. 10646:2020, “Information technology — universal coded character set UCS”, International Organization for Standardization, Geneva, Switzerland., Tech. Rep., Dec. 2020. [Online]. Available: https://www.iso.org/standard/76835.html.

[2] I. 23092-1:2020/CD, “Information technology — genomic information representation — part 1: Transport and storage of genomic information — amendment 1: Support for part 6”, International Organization for Standardization, Geneva, Switzerland., Standard, Oct. 2020. [Online]. Available: https://www.iso.org/standard/23092.html.

[3] I. 8859-1:1998, “Information technology — 8-bit single-byte coded graphic character sets — part 1: Latin alphabet no. 1”, International Organization for Standardization, Geneva, Switzerland., Standard, Dec. 2020. [Online]. Available: https://www.iso.org/standard/28245.html.

[4] C. Albert, T. Paridaens, J. Voges, *et al.*, “An introduction to MPEG-g, the new ISO standard for genomic information representation”, Sep. 2018. DOI: 10.1101/426353.

[5] E. Bianconi, A. Piovesan, F. 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.

[6] P. J. A. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. 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.

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

[8] H. Delfs and H. Knebl, *Introduction to cryptography* (Information Security and Cryptography), en. Berlin, Germany: Springer, Mar. 2007, ISBN: 978-3-540-49243-6.

[9] L. P. 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.

[10] L. P. Deutsch, J.-L. Gailly, M. Adler, L. P. Deutsch, and G. Randers-Pehrson, “GZIP file format specification version 4.3”, RFC 1952, May 1996. DOI: 10.17487/rfc1952. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1952.

[12] R. Fielding, J. Gettys, J. Mogul, *et al.*, “Hypertext transfer protocol – HTTP/1.1”, Tech. Rep., Jun. 1999. DOI: 10.17487/rfc2616. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2616.

[13] M. H.-Y. Fritz, R. Leinonen, G. Cochrane, and E. Birney, “Efficient storage of high throughput DNA sequencing data using reference-based compression”, *Genome Research*, vol. 21, no. 5, pp. 734–740, Jan. 2011. DOI: 10.1101/gr.114819.110.

[15] M. Hosseini, D. Pratas, and A. Pinho, “A survey on data compression methods for biological sequences”, *Information*, vol. 7, no. 4, p. 56, Oct. 2016. DOI: 10.3390/info7040056.

[16] D. Huffman, “A method for the construction of minimum-redundancy codes”, *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952. DOI: 10.1109/jrproc.1952.273898.

[17] “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.

[19] A. 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.

[20] P. Juliana, R. P. Singh, J. Poland, *et al.*, “Elucidating the genetics of grain yield and stress-resilience in bread wheat using a large-scale genome-wide association mapping study with 55,568 lines”, *Scientific Reports*, vol. 11, Mar. 2021. DOI: 10.1038/s41598-021-84308-4.

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

[22] W. Y. Low, R. Tearle, R. Liu, *et al.*, “Haplotype-resolved genomes provide insights into structural variation and gene content in angus and brahman cattle”, *Nature Communications*, vol. 11, no. 1, Apr. 2020. DOI: 10.1038/s41467-020-15848-y.

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

[24] C. McIntosh, *Cambridge International Dictionary of English*. Cambridge University Press, 2013, p. 1856, ISBN: 9781107035157.

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

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

[28] A. G. Motulsky, “Impact of genetic manipulation on society and medicine”, *Science*, vol. 219, pp. 135–140, Jan. 1983. DOI: 10.1126/science.6336852.

[30] A. Al-Okaily, B. Almarri, S. A. Yami, and C.-H. 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.

[31] S. 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.

[32] J. Postel, “User datagram protocol”, RFC Editor, Tech. Rep. 768, Aug. 28, 1980, 3 pp. DOI: 10.17487/RFC0768. [Online]. Available: https://www.rfc-editor.org/info/rfc768.

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

[34] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003, ISBN: 0071232656.

[35] M. RajShivare, Y. P. S. Maravi, and S. 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.

[37] 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.

[38] K. Sailunaz, M. Kotwal, and M. 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.

[40] 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.

[41] K. Simonsen, “Character mnemonics and character sets”, RFC 1345, Jun. 1992. DOI: 10.17487/rfc1345. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1345.

[44] S.-W. Wang, C. Gao, Y.-M. Zheng, *et al.*, “Current applications and future perspective of CRISPR/cas9 gene editing in cancer”, *Molecular Cancer*, vol. 21, no. 1, Feb. 2022. DOI: 10.1186/s12943-022-01518-8.

[45] J. Watson and F. 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.

[46] I. H. Witten, R. M. Neal, and J. 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.

[47] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression”, *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977. DOI: 10.1109/TIT.1977.1055714.

Online Sources

[11] “Ensembl FTP-Server”. (), [Online]. Available: https://ftp.ensembl.org (visited on 10/15/2022).

[14] “GPL - GNU Public License description”. (), [Online]. Available: http://www.gnu.org/licenses/gpl-3.0.html (visited on 11/20/2022).

[18] “IGSR – The International Genome Sample Resource”. (), [Online]. Available: https://ftp.1000genomes.ebi.ac.uk (visited on 11/10/2022).

[25] “MIT license description”. (), [Online]. Available: https://spdx.org/licenses/MIT.html (visited on 11/23/2022).

[29] “NCBI – National Center for Biotechnology Information”. (), [Online]. Available: https://ftp.ncbi.nlm.nih.gov/genomes/ (visited on 11/01/2022).

[36] “Repositories for the three versions of GeCo”. (), [Online]. Available: https://github.com/cobilab (visited on 11/19/2022).

[39] “Sequence Alignment/Map Format Specification”. (), [Online]. Available: https://github.com/samtools/hts-specs (visited on 09/12/2022).

[42] “The Ensembl Project”. (), [Online]. Available: http://www.ensembl.org/ (visited on 10/24/2022).

[43] “UCSC University of California Santa Cruz - Genome Browser”. (), [Online]. Available: https://genome.ucsc.edu/ (visited on 10/28/2022).

Appendix A

Test Environment and Server Specification

CPU specification. Due to redundance, the information is limited to the last core, beginning at: processor : 7

```
\label{a5:cpu}
  cat /proc/cpuinfo
```

```
processor : 0
...


processor : 7
vendor_id : GenuineIntel
cpu family : 6
model : 58
model name : Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
stepping : 9
microcode : 0x15
cpu MHz : 2412.891
cache size : 8192 KB
physical id : 0
siblings : 8
core id : 3
cpu cores : 4
apicid : 7
initial apicid : 7
fpu : yes
fpu_exception : yes
cpuid level : 13
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc
arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni
```

pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm cpuid_fault epb pti tpr_shadow vnmi flexpriority ept vpid fsgsbase smep erms xsaveopt dtherm ida arat pln pts
vmx flags : vnmi preemption_timer invvpid ept_x_only flexpriority tsc_offset vtpr mtf vapic ept vpid unrestricted_guest
bugs : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs itlb_multihit srbds mmio_unknown
bogomips : 6784.56
clflush size : 64
cache_alignment : 64
address sizes : 36 bits physical, 48 bits virtual
power management:

manually installed packages:

- autoconf
- automake
- bzip2
- cmake
- gcc
- git
- htop
- libbz2-dev
- libcurl4-gnutls-dev
- libhts-dev
- libhtscodecs2
- liblzma-dev

- libncurses5-dev
- libomp-dev
- libssl-dev
- zlib1g-dev
- openssh-client
- perl
- rsync
- screen
- sudo
- ufw
- vim
- wget

Appendix B

Raw Test Results Structured in Tables

Table B.1.: Compression duration of various tools, measured in milliseconds

ID.	GeCo	Samtools BAM	Samtools CRAM
File 1.1	235005	3786	16926
File 1.2	246503	3784	17043
File 1.3	20169	3123	13999
File 1.4	194081	3011	13445
File 1.5	183878	2862	12802
File 1.6	173646	2685	12015
File 1.7	159999	2503	11198
File 1.8	148288	2286	10244
File 1.9	12304	2078	9210
File 1.10	134937	2127	9461
File 1.11	136299	2132	9508
File 1.12	134932	2115	9456
File 1.13	999022	1695	7533
File 1.14	924753	1592	7011
File 1.15	852555	1507	6598
File 1.16	827651	1390	6089
File 1.17	820814	1306	5791
File 1.18	798429	1277	5603
File 1.19	586058	960	4106
File 1.20	645884	1026	4507
File 1.21	411984	721	3096
File 2.1	58427	16248	23016
File 2.2	57905	15770	22892
File 2.3	9725	7732	12858
File 2.4	13694	8291	13649
File 2.5	51001	14754	23713
File 2.6	51315	15142	24358
File 2.7	2065	16379	23484

Table B.2.: File sizes for different formats in byte

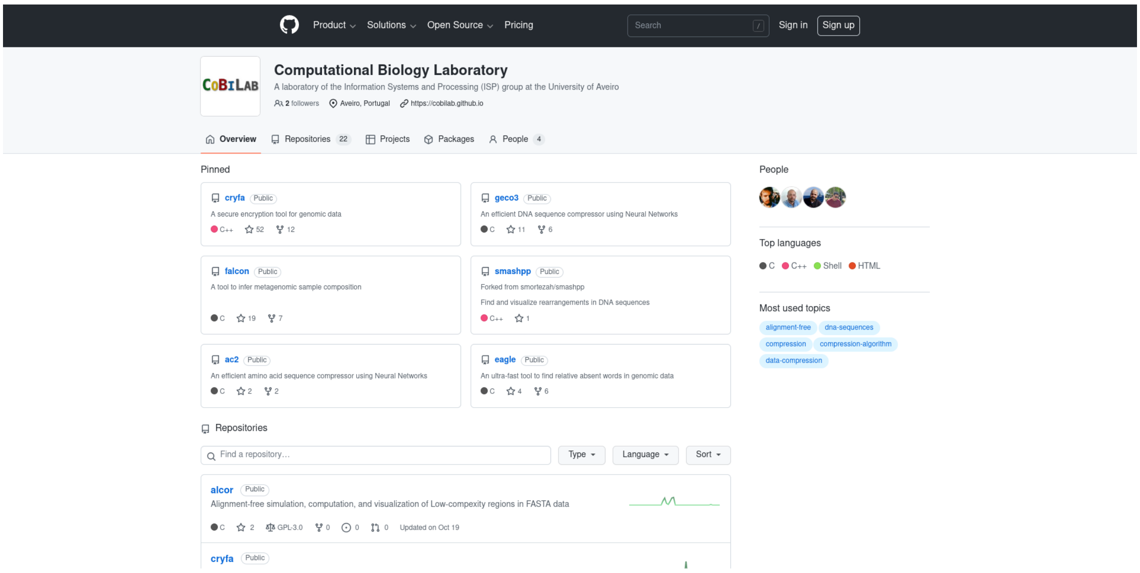
ID.	Uncompressed Source File	GeCo	Samtools BAM	Samtools CRAM
File 1.1	253105752	46364770	62048289	55769827
File 1.2	246230144	49938168	65391181	58026123
File 1.3	201600541	41117340	53586949	47707954

File 1.4	193384854	39248276	51457814	45564837
File 1.5	184563953	37133480	48838053	43655371
File 1.6	173652802	35355184	46216304	40980906
File 1.7	162001796	31813760	42371043	38417108
File 1.8	147557670	30104816	39107538	34926945
File 1.9	140701352	23932541	32708272	29459829
File 1.10	136027438	27411806	35855955	32238052
File 1.11	137338124	27408185	35894133	32529673
File 1.12	135496623	27231126	35580843	32166751
File 1.13	116270459	20696778	26467775	23568321
File 1.14	108827838	18676723	24284901	21887811
File 1.15	103691101	16804782	22486646	20493276
File 1.16	91844042	16005173	21568790	19895937
File 1.17	84645123	15877526	21294270	20177456
File 1.18	81712897	16344067	20684650	19310998
File 1.19	59594634	10488207	14616042	14251243
File 1.20	65518294	13074402	16769658	15510100
File 1.21	47488540	7900773	10477999	9708258
File 2.1	1246731616	12414797	78260121	67130756
File 2.2	1261766002	12363734	80895953	69649632
File 2.3	657946854	7966180	53201724	47175349
File 2.4	708837816	8499132	54569686	48521201
File 2.5	1118234394	12088239	84764250	75118457
File 2.6	1123124224	12265535	88147227	77826446
File 2.7	1300825946	12450651	75860986	60239362

Appendix C

Visual Persistence of Used Online Sources

The following images provide insight into the online sources referenced in this work:



Computational Biology Laboratory

A laboratory of the Information Systems and Processing (ISP) group at the University of Aveiro

2 followers · Aveiro, Portugal · https://cobilab.github.io

cryfa Public

A secure encryption tool for genomic data

● C++ ☆ 52 12

geco3 Public

An efficient DNA sequence compressor using Neural Networks

● C ☆ 11 6

falcon Public

A tool to infer metagenomic sample composition

● C ☆ 19 7

smashpp Public

Forked from smortezah/smashpp

Find and visualize rearrangements in DNA sequences

● C++ ☆ 1

ac2 Public

An efficient amino acid sequence compressor using Neural Networks

● C ☆ 2 2

eagle Public

An ultra-fast tool to find relative absent words in genomic data

● C ☆ 4 6

Repositories

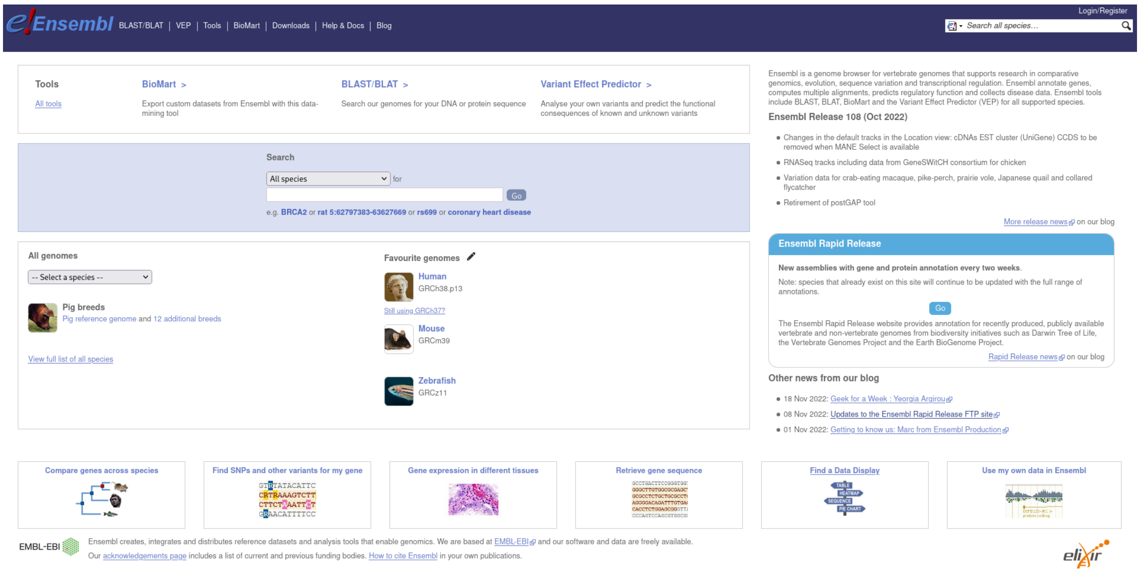
alcor Public

Alignment-free simulation, computation, and visualization of Low-compexity regions in FASTA data

● C ☆ 2 GPL-3.0 0 0 0 Updated on Oct 19

cryfa Public

Figure C.1.: View of Cobilabs - creators of GeCo - github page [36]



Tools

All tools

BioMart >

Export custom datasets from Ensembl with this data-mining tool

BLAST/BLAT >

Search our genomes for your DNA or protein sequence

Variant Effect Predictor >

Analyse your own variants and predict the functional consequences of known and unknown variants

Search

All species for

e.g. BRCA2 or rat 5:62797383-63627669 or rs699 or coronary heart disease

All genomes

-- Select a species --

Pig breeds
Pig reference genome and 12 additional breeds

View full list of all species

Favourite genomes

Human
GRCh38.p13

Still using GRCh37?

Mouse
GRCm39

Zebrafish
GRCz11

Ensembl is a genome browser for vertebrate genomes that supports research in comparative genomics, evolution, sequence variation and transcriptional regulation. Ensembl annotate genes, computes multiple alignments, predicts regulatory function and collects disease data. Ensembl tools include BLAST, BLAT, BioMart and the Variant Effect Predictor (VEP) for all supported species.

Ensembl Release 108 (Oct 2022)

- Changes in the default tracks in the Location view: cDNAs EST cluster (UniGene) CCDS to be removed when MANE Select is available
- RNASeq tracks including data from GeneSWitCH consortium for chicken
- Variation data for crab-eating macaque, pike-perch, prairie vole, Japanese quail and collared flycatcher
- Retirement of postGAP tool

More release news on our blog

Ensembl Rapid Release

New assemblies with gene and protein annotation every two weeks.

Note: species that already exist on this site will continue to be updated with the full range of annotations.

The Ensembl Rapid Release website provides annotation for recently produced, publicly available vertebrate and non-vertebrate genomes from biodiversity initiatives such as Darwin Tree of Life, the Vertebrate Genomes Project and the Earth BioGenome Project.

Rapid Release news on our blog

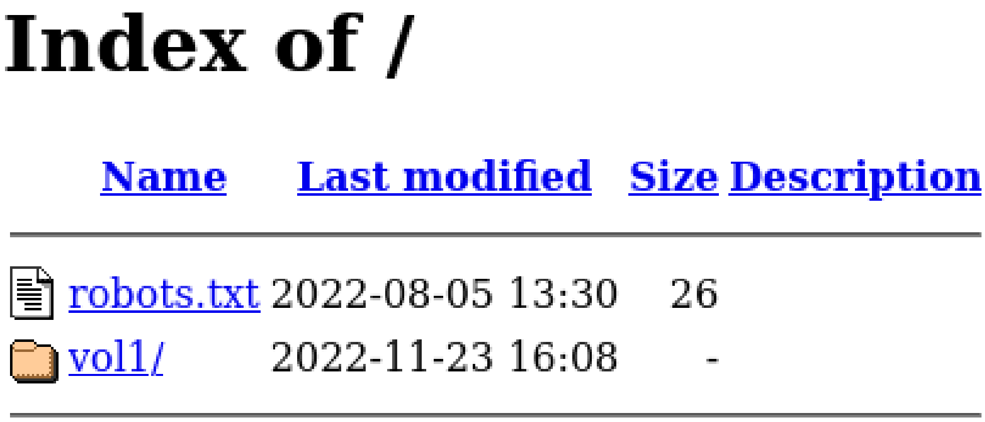
Other news from our blog

- 18 Nov 2022: Geek for a Week : Yeorgia Argirou
- 08 Nov 2022: Updates to the Ensembl Rapid Release FTP site
- 01 Nov 2022: Getting to know us: Marc from Ensembl Production

Compare genes across species | Find SNPs and other variants for my gene | Gene expression in different tissues | Retrieve gene sequence | Find a Data Display | Use my own data in Ensembl

Ensembl creates, integrates and distributes reference datasets and analysis tools that enable genomics. We are based at EMBL-EBI and our software and data are freely available.
Our acknowledgements page includes a list of current and previous funding bodies. How to cite Ensembl in your own publications.

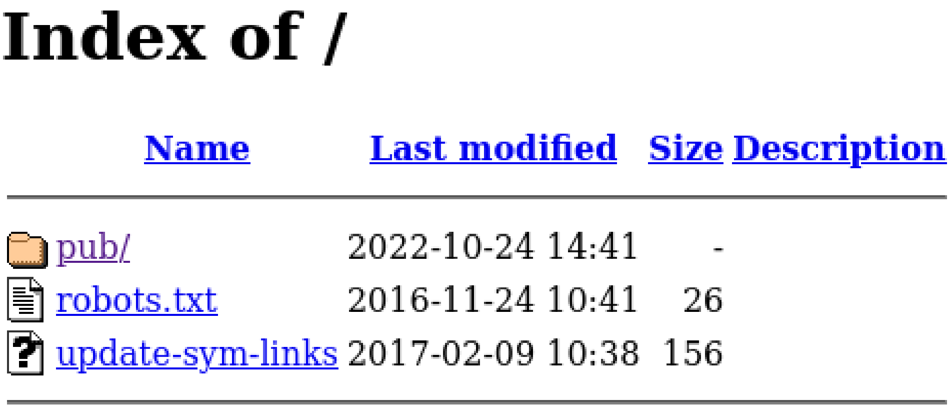
Figure C.2.: View of ensembl genome browser splash page [42].



Index of /

Name	Last modified	Size	Description
robots.txt	2022-08-05 13:30	26	
vol1/	2022-11-23 16:08	-	

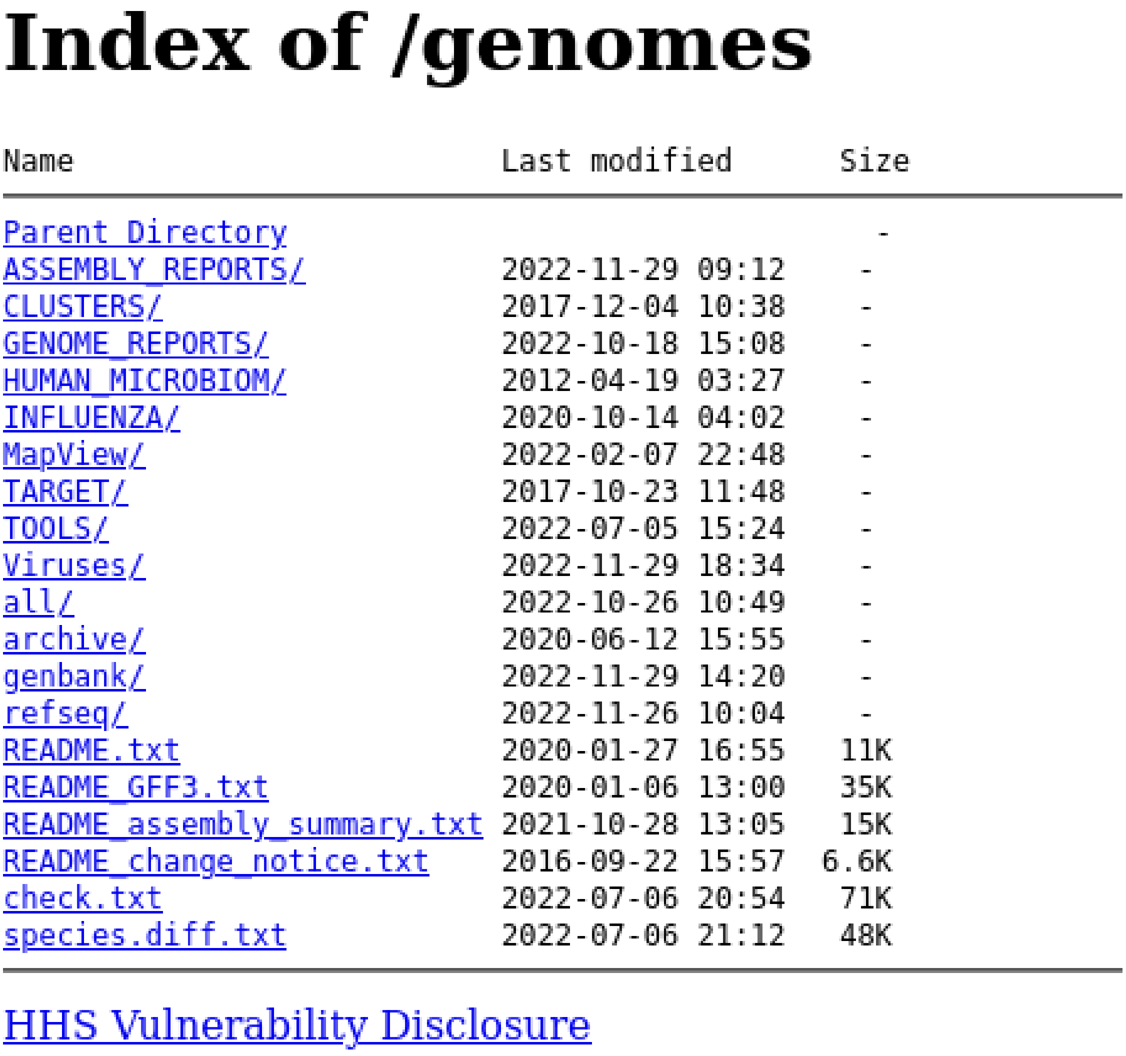
Figure C.3.: View of the browser view for IGSRs FTP-Server [18].



Index of /

Name	Last modified	Size	Description
pub/	2022-10-24 14:41	-	
robots.txt	2016-11-24 10:41	26	
update-sym-links	2017-02-09 10:38	156	

Figure C.4.: View of the browser view for Ensembls FTP-Server [11].

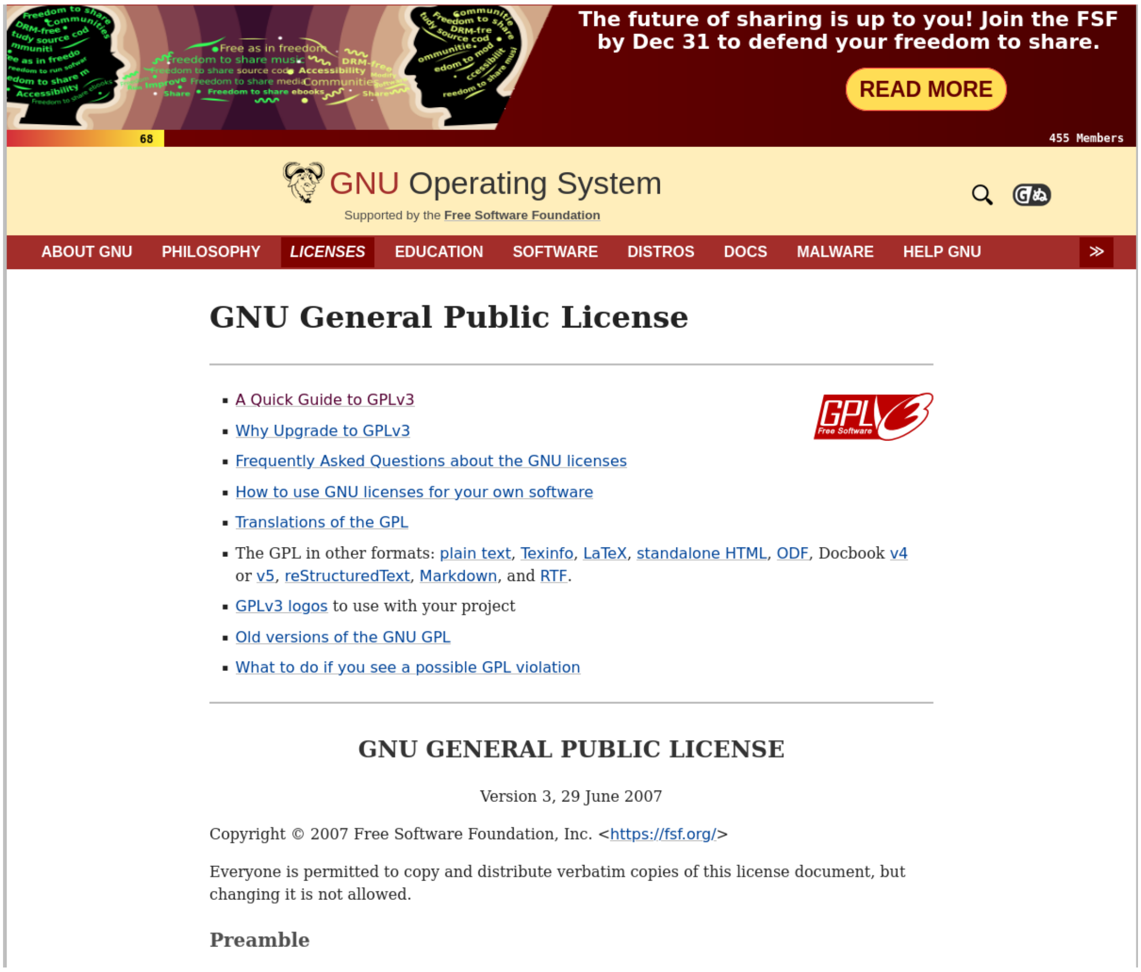


Index of /genomes

```
Name                          Last modified      Size
---------------------------------------------------------
Parent Directory                                   -
ASSEMBLY_REPORTS/             2022-11-29 09:12    -
CLUSTERS/                     2017-12-04 10:38    -
GENOME_REPORTS/               2022-10-18 15:08    -
HUMAN_MICROBIOM/              2012-04-19 03:27    -
INFLUENZA/                    2020-10-14 04:02    -
MapView/                      2022-02-07 22:48    -
TARGET/                       2017-10-23 11:48    -
TOOLS/                        2022-07-05 15:24    -
Viruses/                      2022-11-29 18:34    -
all/                          2022-10-26 10:49    -
archive/                      2020-06-12 15:55    -
genbank/                      2022-11-29 14:20    -
refseq/                       2022-11-26 10:04    -
README.txt                    2020-01-27 16:55   11K
README_GFF3.txt               2020-01-06 13:00   35K
README_assembly_summary.txt   2021-10-28 13:05   15K
README_change_notice.txt      2016-09-22 15:57  6.6K
check.txt                     2022-07-06 20:54   71K
species.diff.txt              2022-07-06 21:12   48K
---------------------------------------------------------
```

HHS Vulnerability Disclosure

Figure C.5.: View of the browser view for NCBIs FTP-Server [29].


The future of sharing is up to you! Join the FSF by Dec 31 to defend your freedom to share.
READ MORE
68
455 Members
GNU Operating System
Supported by the Free Software Foundation
ABOUT GNU
PHILOSOPHY
LICENSES
EDUCATION
SOFTWARE
DISTROS
DOCS
MALWARE
HELP GNU
GPLv3 Free Software

GNU General Public License

- A Quick Guide to GPLv3
- Why Upgrade to GPLv3
- Frequently Asked Questions about the GNU licenses
- How to use GNU licenses for your own software
- Translations of the GPL
- The GPL in other formats: plain text, Texinfo, LaTeX, standalone HTML, ODF, Docbook v4 or v5, reStructuredText, Markdown, and RTF.
- GPLv3 logos to use with your project
- Old versions of the GNU GPL
- What to do if you see a possible GPL violation

GNU GENERAL PUBLIC LICENSE

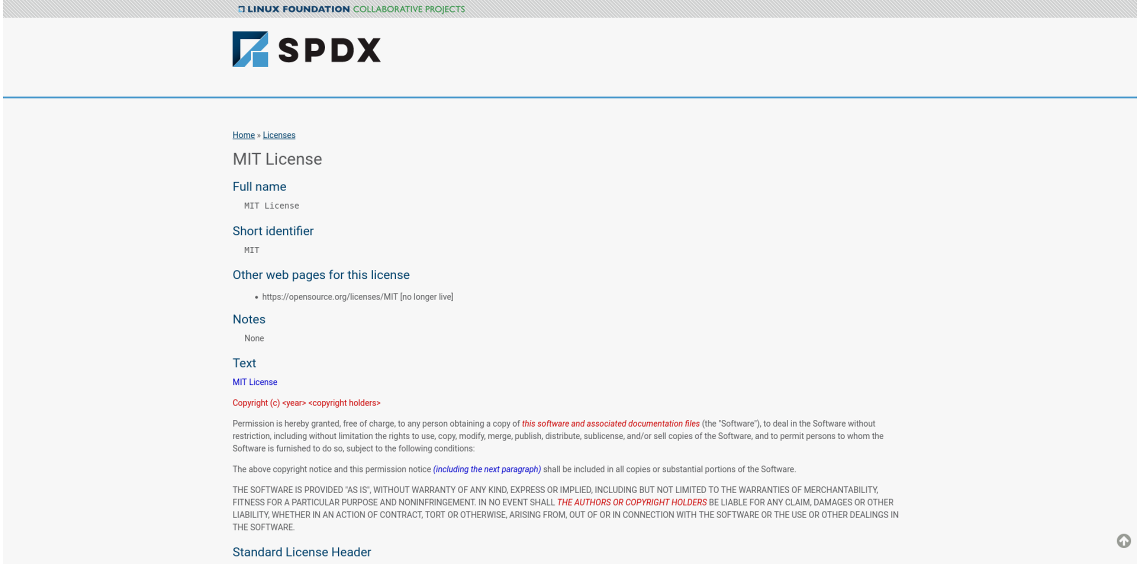
Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <https://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

Figure C.6.: View of the GNU Public License page [14].



MIT License

Full name

MIT License

Short identifier

MIT

Other web pages for this license

- https://opensource.org/licenses/MIT [no longer live]

Notes

None

Text

MIT License

Copyright (c) <year> <copyright holders>

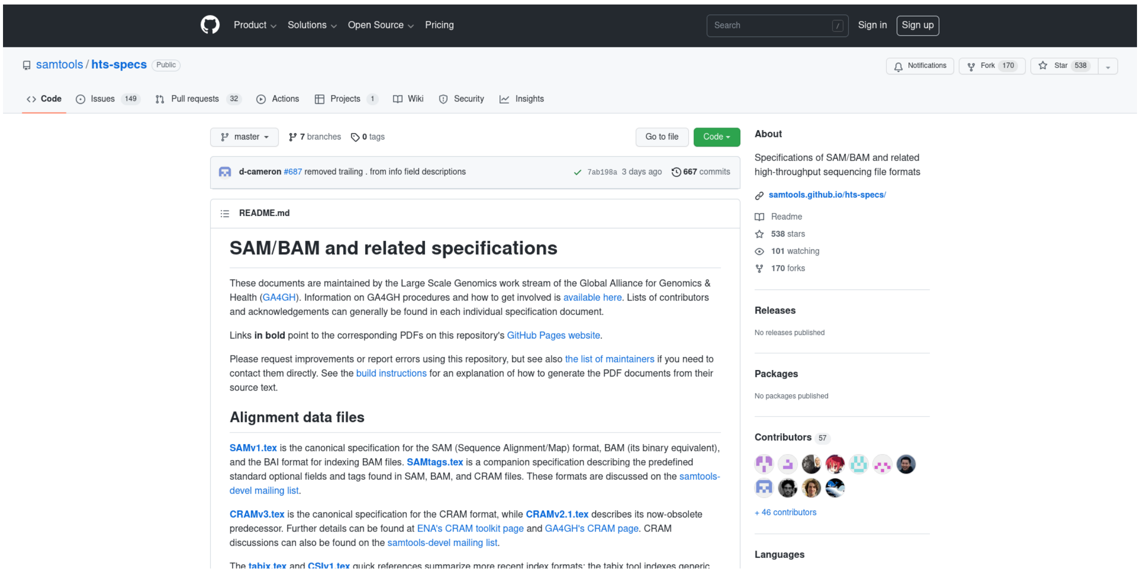
Permission is hereby granted, free of charge, to any person obtaining a copy of *this software and associated documentation files* (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice *(including the next paragraph)* shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL *THE AUTHORS OR COPYRIGHT HOLDERS* BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Standard License Header

Figure C.7.: View of MIT license description [25].



d-cameron #687 removed trailing . from info field descriptions ✓ 7ab198a 3 days ago 667 commits

README.md

SAM/BAM and related specifications

These documents are maintained by the Large Scale Genomics work stream of the Global Alliance for Genomics & Health (GA4GH). Information on GA4GH procedures and how to get involved is available here. Lists of contributors and acknowledgements can generally be found in each individual specification document.

Links **in bold** point to the corresponding PDFs on this repository's GitHub Pages website.

Please request improvements or report errors using this repository, but see also the list of maintainers if you need to contact them directly. See the build instructions for an explanation of how to generate the PDF documents from their source text.

Alignment data files

SAMv1.tex is the canonical specification for the SAM (Sequence Alignment/Map) format, BAM (its binary equivalent), and the BAI format for indexing BAM files. **SAMtags.tex** is a companion specification describing the predefined standard optional fields and tags found in SAM, BAM, and CRAM files. These formats are discussed on the samtools-devel mailing list.

CRAMv3.tex is the canonical specification for the CRAM format, while **CRAMv2.1.tex** describes its now-obsolete predecessor. Further details can be found at ENA's CRAM toolkit page and GA4GH's CRAM page. CRAM discussions can also be found on the samtools-devel mailing list.

The **tabix.tex** and **CSIv1.tex** quick references summarize more recent index formats; the tabix tool indexes generic

About

Specifications of SAM/BAM and related high-throughput sequencing file formats

samtools.github.io/hts-specs/

Readme
538 stars
101 watching
170 forks

Releases

No releases published

Packages

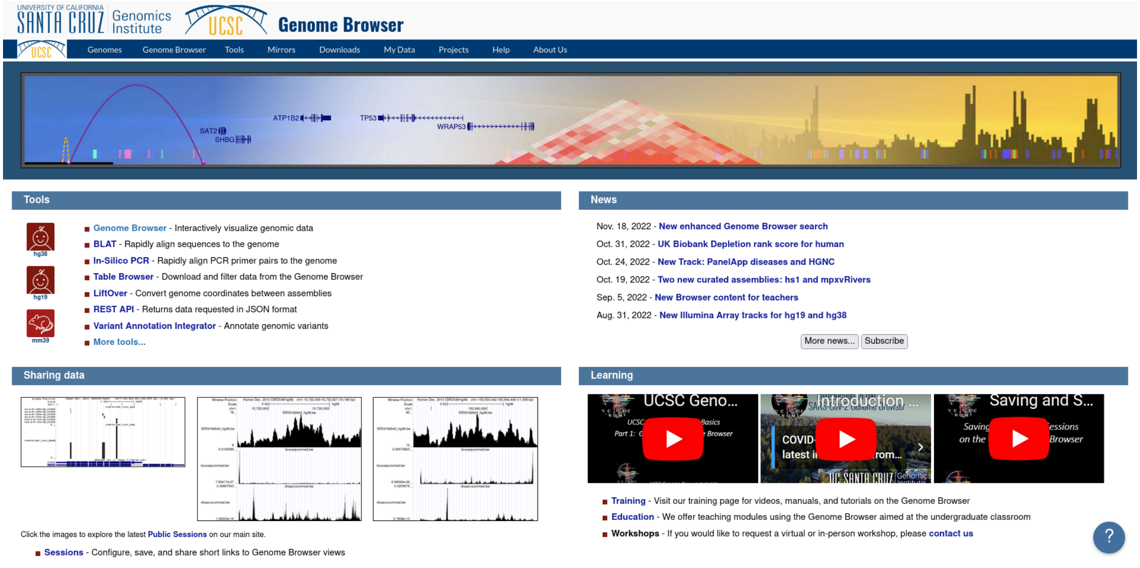
No packages published

Contributors 57

+ 46 contributors

Languages

Figure C.8.: View of Samtools sourcefiles and file format description [39].



Tools

- **Genome Browser** - Interactively visualize genomic data
- **BLAT** - Rapidly align sequences to the genome
- **In-Silico PCR** - Rapidly align PCR primer pairs to the genome
- **Table Browser** - Download and filter data from the Genome Browser
- **LiftOver** - Convert genome coordinates between assemblies
- **REST API** - Returns data requested in JSON format
- **Variant Annotation Integrator** - Annotate genomic variants
- **More tools...**

News

Nov. 18, 2022 - **New enhanced Genome Browser search**

Oct. 31, 2022 - **UK Biobank Depletion rank score for human**

Oct. 24, 2022 - **New Track: PanelApp diseases and HGNC**

Oct. 19, 2022 - **Two new curated assemblies: hs1 and mpxvRivers**

Sep. 5, 2022 - **New Browser content for teachers**

Aug. 31, 2022 - **New Illumina Array tracks for hg19 and hg38**

Sharing data

Click the images to explore the latest **Public Sessions** on our main site.

- **Sessions** - Configure, save, and share short links to Genome Browser views

Learning

- **Training** - Visit our training page for videos, manuals, and tutorials on the Genome Browser
- **Education** - We offer teaching modules using the Genome Browser aimed at the undergraduate classroom
- **Workshops** - If you would like to request a virtual or in-person workshop, please **contact us**

Figure C.9.: View of UCSC splash page [43].

:wq