Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Mapping between Genomes

Bachelor Thesis

2014

Petra Kubincová

Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Mapping between Genomes

Bachelor Thesis

Study programme:	Informatics
Study field:	2508 Informatics
Department:	Department of Computer Science
Supervisor:	Mgr. Bronislava Brejová, PhD.

Bratislava, 2014

Petra Kubincová

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Petra Kubincová
Study programme:	Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	9.2.1. Computer Science, Informatics
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Mapping Between Genomes

Aim: In recent years, availability of genomic sequences from many different organisms lead to demand for computational tools capable of efficiently processing such data. The goal of this thesis is to implement a practical tool for efficiently mapping genomic regions from one genome to another based on inferred evolutionary relationships. Such relationships between genomic regions can be computed by existing tools, and the goal is to develop file formats and data structures for storing and efficiently accessing such data.

Supervisor:	Mgr. Bronislava Brejová, PhD.
Department:	FMFI.KI - Department of Computer Science
Head of department:	doc. RNDr. Daniel Olejár, PhD.
Assigned:	24.10.2013

Approved: 24.10.2013 doc. RNDr. Daniel Olejár, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Petra Kubincová
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Mapping Between Genomes
Mapovanie medzi genómami

Cieľ: V súčasnosti máme k dispozícii genomické sekvencie mnohých organizmov, vďaka čomu rastie dopyt po softvérových nástrojoch na ich analýzu. Cieľom tejto práce je vytvoriť praktický nástroj na mapovanie genomických oblastí z jedného genómu do druhého na základe ich predpovedaných evolučných vzťahov. Evolučné vzťahy medzi regiónmi je možné spočítať existujúcimi nástrojmi, pričom cieľom tejto práce je vyvinúť formáty súborov a dátové štruktúry na ukladanie a efektívny prístup k týmto dátam.

Vedúci: Mgr. Bronislava Brejová, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Dátum zadania: 24.10.2013

Dátum schválenia: 24.10.2013 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Acknowledgement

Most of all, I would like to thank my supervisor Mgr. Bronislava Brejová, PhD. for her patience, guidance and a lot of helpful advice.

Special thanks to my family and friends for their support.

Abstrakt

V tejto práci sa venujeme problému mapovania oblastí medzi genómami. Cieľom je návrh a implementácia praktického nástroja na efektívne mapovanie genomických oblastí z jedného genómu do druhého, založeného na zarovnaniach vypočítaných existujúcimi programami. Navrhujeme reprezentáciu zarovnaní a formáty súborov podporujúce efektívny prístup k dátam, a popisujeme teoretické problémy súvisiace s mapovaním. V práci taktiež opisujeme podobné, už existujúce nástroje a porovnávame ich s naším nástrojom.

Kľúčové slová: zarovnanie, celogenómové zarovnanie, mapovanie oblastí, rank, select, range minimum query problém

Abstract

In this thesis we focus on the problem of mapping regions between genomes. The goal is to design and implement a practical tool for efficiently mapping genomic regions from one genome to another, based on genomic alignments computed by existing programs. We propose representation of the alignment data and file formats allowing efficient data access, and discuss theoretical problems related to mapping. We also describe similar existing tools and compare them with our tool.

Keywords: alignment, whole-genome alignment, region mapping, rank, select, range minimum query problem

Contents

Introduction **1**

1 Problem statement **2**
1.1 Biological point of view 2
1.2 Computer science abstraction 5
1.3 What is a correct mapping? 6

2 Related work **11**
2.1 MAF 11
2.2 BED 12
2.3 liftOver 13
2.4 Libmultialn 15
2.5 HAL 16

3 Maptool **19**
3.1 Overall design 19
3.2 Representation of alignments 20
3.3 Preprocessing 23
3.4 Mapping 25
3.4.1 Loading the alignment data 26
3.4.2 Mapping a region 26
3.4.3 Time complexity 27
3.5 Comparison of HAL, liftOver and Maptool 28
3.5.1 Preprocessing 28
3.5.2 Mapping 30
3.5.3 Summary 33

4 Theoretical aspects **34**
4.1 Problem formulation 34
4.2 Survey of RMQ data structures 36

4.2.1 Naïve solution 36
4.2.2 Sparse Table algorithm 36
4.2.3 Segment tree 37
4.2.4 Even better solutions 39

Conclusion **40**

A – Implementation **44**

B – Credits for the used files **45**

List of Figures

1.1 Sample pairwise alignment 4
1.2 Sample alignment of three sequences 4
1.3 Possible alignments 7
1.4 Sample alignment consisting of three blocks 8

2.1 Example of a an alignment block from a MAF file 12
2.2 Example of a BED line 13
2.3 Example of a chain 14
2.4 Example of a phylogenetic tree 17
2.5 Example of a HAL graph 18

3.1 Example of the first considered representation 20
3.2 Example of the second considered representation 21
3.3 Example of the third considered representation 22
3.4 Comparison of running time of Maptool and HAL preprocessing 29
3.5 Comparison of memory used by Maptool and HAL preprocessing 29
3.6 Comparison of running time of Maptool, HAL and liftOver for dataset 1 31
3.7 Comparison of running time of Maptool, HAL and liftOver for dataset 2 31
3.8 Comparison of memory used by Maptool, HAL and liftOver for dataset 1 32
3.9 Comparison of memory used by Maptool, HAL and liftOver for dataset 2 32

4.1 Example of the modified representation of the alignment 35

List of Tables

3.1 Testing data for preprocessing 28
3.2 Outputs of preprocessing 30
3.3 Testing data for mapping (1) 30
3.4 Testing data for mapping (2) 31

Introduction

Thanks to the technological progress over recent decades, we known genomic sequences of many species. To compare genomes of several related species, we can construct alignments, identifying corresponding parts of these sequences. The alignments usually consist of one reference sequence to which a number of informant sequences are aligned. Several programs constructing such alignments are known.

The corresponding parts of an alignment are likely to be evolutionary related and often have similar functions. Therefore, given a region in one sequence, it is desirable to find the corresponding region in another sequence according to the alignment. Since the alignments can be very large, such mapping between two genomes needs to be performed effieciently, in terms of both time and memory.

In this thesis, we focus on the problem of mapping regions between genomes. This includes multiple subproblems: designing a representation of the alignment data, data structures and file formats supporting fast mapping, and the mapping itself.

We propose and implement a solution to this problem, which firstly preprocesses the alignment data, stores them in two files and than efficiently maps regions from the reference genome to any of the informant genomes in the alignment, using only the preprocessed files.

We start with overview of the biological background along with a computer science abstraction. Next, we introduce a set of rules of biological origin that lead to a definition of a correct mapping. Afterwards, we describe existing tools and file formats related to the problem of mapping between genomes. Then we explain our solution of the problem and compare it with known tools for mapping. Finally, we analyse a theoretical problem related to mapping, reduce it to the well-known Range Minimum Query problem and present a survey of its solutions.

Chapter 1

Problem statement

In this section we explain biological origin of the problem and its computer science abstraction.

1.1 Biological point of view

Genetic information of living organisms is stored in deoxyribonucleic acid (DNA), a double-stranded helix consisting of two biopolymeres made of smaller units, nucleotides. In the DNA, there are four types of nucelotides, which differ by nucleobase they contain – guanine, adenine, thymine and cytosine. It is natural to represent the information contained in a DNA strand as a sequence of letters G, A, T and C, each corresponding to one nucleobase. This representation of a DNA strand is called a (DNA) *sequence*. Since the technologies used to extract the sequence information from DNA sometimes do not offer a precise information about which base is at some position in the DNA, we will represent a base of unknown type with letter N.

Note that sequences of the two strands, of which the DNA consists, are not independent. When there is a guanine on one of the strands, there can only be cytosine at the corresponding position on the other strand, and vice versa. Analogously thymine and adenine are always paired together. Two bases at the corresponding positions in the two strands of a DNA molecule are called a base pair.

Since the nucleotides are asymmetric, the strands are oriented (so a strand with DNA sequence `ACC` is not the same as a strand with DNA sequence `CCA`), and because of the molecular structure, the complementary strands have opposite orientation. For example, when there is DNA sequence `ACC` on one of the strands, the DNA sequence of the other strand is `GGT`. Because of this complementarity of the DNA double helix, there is no need to store sequences of both strands of a helix – one of them can be computed from the other. In the following, we will denote the strands as the + strand

and − strand, and we store the sequence of the + strand.

In living organisms, genetic information typically consist of a number of DNA molecules, each packaged with histone proteins into a structure known as *chromosome*. All genetic material of an organism is called its *genome*.

Since the genetic information is hereditary, genomes of related organisms are similar. This applies also to whole species – when two species are evolutionary related (meaning that they had a common ancestor in the recent past), their DNA will be similar. Conversely, if genomes of two organisms (species) are similar, then they are likely to be related.

Two DNA sequences being similar means that they both can be constructed from one DNA sequence by a small number of mutations. This corresponds to the evolutionary model: two related organisms, A and B, had a common ancestor, whose DNA went through a number of mutations before becoming the A's and B's DNA. There is a number of different mutation types [Ree+10]:

- Large-Scale Mutations:
  - deletion – loss of a chromosomal fragment
  - duplication – addition of an already present chromosomal fragment
  - inversed duplication – the duplicated fragment is inserted in the complementary strand
  - translocation – loss and consecutive addition of the same chromosomal fragment on a different position
- Small-Scale Mutations:
  - nucleotide-pair substitution – change of one nucleotide pair from one base pair to another
  - deletion – loss of several nucleotide pairs
  - insertion – addition of several nucleotide pairs

Similarities between sequences can be highlighted in the form of an *alignment*. An alignment of two sequences is a pair of strings created by inserting hyphen characters to the original sequences so that when these strings are written in consecutive lines, corresponding letters are above each other, as in figure 1.1.

Aligned parts of similar sequences often carry information serving similar purposes, e.g. if there are genes from two organisms aligned to each other in a way that corresponding letters are largely the same, we can expect that the function of these genes

```
TACGAGGATGTT-A
T---ACGATG--CA
```

Figure 1.1: Sample pairwise alignment

is similar. Hence, sequence alignments can help us to understand evolutionary history of a particular region of DNA as well as its function.

Alignments are therefore constructed not only for small parts of sequences, but also for whole genomes. Because of large-scale mutations, the chromosome fragments of one organism can be rearranged when compared to corresponding fragments of the other organism, meaning that we cannot satisfactorily align a whole chromosome of one organism to its counterpart from another organism in a single alignment. To solve this problem, a whole-genome alignment is constructed as a set of *alignment blocks*, where each alignment block is an alignment of a part of some chromosome to its counterpart in the other organism.

In addition, alignments are often constructed for more than two organisms, as in figure 1.2. We will call this type of alignment a *multiple alignment*.

```
TACGAGGATGTT-A
T---ACGATG--CA
---GACTAT-TTCG
```

Figure 1.2: Sample alignment of three sequences

In this thesis, we will consider multiple whole-genome alignments in which one genome is selected as a *reference*, the other genomes as *informants*. For the reference genome it is guaranteed that each base occurs in the whole-genome alignment at most once, while parts of the informant sequences can occur multiple times in different blocks.

There already exist several computer programs constructing multiple alignments [Bla+04], but having such an alignment a new problem arises: according to the alignment, which part of the DNA sequence from one organism corresponds to a given part of the DNA sequence from another organism? The results of such mapping between genomes of two organisms is used for multiple purposes, e.g. to study of evolution [Kos+08].

An example of frequently mapped DNA sequences are *genes*: parts of the DNA sequence of various length (ranging from dozens to tens of thousands bases) coding proteins. The process of protein synthesis based on the genetic information is, however, complicated and often only information from selected parts of the gene, called *exons*,

is used to synthesize the resulting proteins [Ree+10]. Thus, when mapping a gene, it could be useful to map also its exons.

In the following text we will focus on the problem of mapping between genomes and we will describe the design and implementation of a program which can efficiently map a given part of the reference genome to a part of a given informant genome.

1.2 Computer science abstraction

DNA sequences can be fairly long, and because of that also the alignments can require much space. (Only human genome sequence on its own has more than 3 billion nucleotides.) This is the reason why straightforward approaches to the mapping problem are not sufficient, and we need to desing an efficient algorithm finding the answers.

Alignment data used for mapping consists of a number of alignment blocks. Each block contains one reference sequence, multiple informant sequences (but not neccessarily the same number of informants in each block) and the following information about the sequences:

- name of the chromosome to which the sequence belongs
- strand of the chromosome to which the sequence belongs
- position on the chromosome at which the sequence starts.

The strand of the reference sequence is typically the + strand, but for informant sequences it may be either + or −, because the translocations and inversed duplications could change the strand on which is a fragment of a DNA sequence located.

Our program will first preprocess the input alignment data to a form suitable for fast mapping. We assume that blocks in the input alignment are sorted according to the position reference, i.e. the chromosome position of the reference sequence in one block equals always at least the chromosome position of the reference sequence in the previous block plus the number of the bases (letters) in the reference sequence in the previous block.

In the second phase, our program maps *regions* between two genomes. A region is the tuple $(chr, s, [p_1, p_2])$ where chr is the name of a chromosome and $s \in \{+, -\}$ is an identifier of strand on which is the closed interval of positions $[p_1, p_2]$ located.

Positions in the chromosomes are zero-based and relative to the start of a strand. For example, if a chromosome $chr0$ had DNA sequence `ATGCCTT` on the + strand and

DNA sequence `AAGGCAT` on the − strand, the region $(chr0, +, [1, 3])$ would denote `TGC` and the region $(chr0, -, [1, 3])$ would denote `AGG`.

Our program uses precomputed data structures to answer *mapping queries*. Each mapping query is a pair (I_R, inf) where I_R is a region in the reference sequence and inf is a name of an informant. The desired output is the region I_I in the sequence of the specified informant corresponding to the region I_R.

We will call the input (reference) sequence the *query sequence* and the sequence of the specified informant organism the *target sequence*. Thus the goal is to map a region from the query sequence to the target sequence.

1.3 What is a correct mapping?

As we will show next, in many cases it is not clear how to define the mapping of a particular query. We will introduce a set of rules leading to a formal definition of mapping which is also intuitive from the biological point of view.

Before we present the rules, let us define a *primitive mapping*. A primitive mapping of an region from the reference sequence is the corresponding part of the informant sequence in their alignment. For example, primitive mapping of the `GAGGATGTT` interval from the first (reference) to the second (informant) sequence in the alignment showed in figure 1.1 is the `-ACGATG--` interval.

The first two rules consider mapping of a region which falls into a single alignment block. The third and the fourth rule consider multiple blocks.

Rule R1: Region in the query genome is mapped to an non-empty region in the target genome. Note that a primitive mapping may start or end with the gap character and these characters do not represent any base present in the DNA.

So how do we map regions in the query sequence starting or ending with characters corresponding to a gap in the target sequence? There are two possibilities: to shrink or to enlarge the primitive mapping. In the example above it means that the mapping would be `ACGATG` (shrinked primitive mapping) or `T---ACGATG--C` (enlarged primitive mapping).

Naturally a question rises: which of these two possibilities is the correct one? Unfortunately, there is no generally applicable answer – we may prefer one or the other depending on the application. This means that a tool constructing the mappings should be able to construct both types of mappings: the shrinking type (called *inner*) and also the enlarging type (called *outer*).

Note that sometimes is the inner mapping empty – for example the primitive map-

ping of the ACG interval from the reference to the informant sequence in 1.1 is the --- interval, and therefore the inner mapping is an empty interval. In a case like this we say that the inner mapping does not exist.

Rule R2: Nucleotides starting and ending the target region should be aligned to nucleotides in the query sequence (not to gaps).

To apply this rule, if the desired mapping is inner, the target region from the first rule is shrinked base by base until its ending points are aligned to bases in the query sequence. If the desired mapping is outer, the region is similarly enlarged until the ending points are aligned to bases in the query sequence.

For the inner mapping in the example above, this rule changes nothing, but the outer mapping will be different. Instead of T---ACGATG--C, it will be T---ACGATG--CA, because the last C in the original outer mapping is not aligned to a letter in the query sequence.

This rule originates from the biological point of view: if a region of an alignment contains bases from one sequence mapped to gaps in the other sequence and vice versa, there is no way to tell which letters are supposed to be in the alignment first. For example, figure 1.3 shows three alignments of sequences TGTTA and TGCA which all have the same pairs of bases aligned to each other and differ only in relative placement of columns with gaps.

TGTT-A	TGT-TA	TG-TTA
TG--CA	TG-C-A	TGC--A

Figure 1.3: Possible alignments

Without applying the second rule for outer mapping of query interval TGTT, the target interval might be TGC (in the first alignment) or TGCA (in the second alignment), and when constructing an inner mapping, the target interval might be TG or TGC. With the second rule applied, the inner mapping will be the target interval TG and the outer mapping the target interval TGCA.

Rule R3: When constructing a mapping based on two or more alignment blocks, the target interval mapped to the query interval must consist of *linearly consecutive* parts of the target sequence. We say that two regions $I_1 = (chr_1, s_1, [p_{1,1}, p_{1,2}]), I_2 = (chr_2, s_2, [p_{2,1}, p_{2,2}])$ are linearly consecutive if:

- $chr_1 = chr_2$ (i.e. I_1 and I_2 come from the same chromosome)
- $s_1 = s_2$ (i.e. I_1 and I_2 come from the same strand)

- $p_{1,2} < p_{2,1}$. (i.e. I_2 starts after I_1 ends)

Note that region I_2 does not have to start exactly where region I_1 ended: a missing part of the informant sequence is the same as if it was present and aligned to gaps in the reference sequence.

If the query region spreads over $k \geq 2$ different alignment blocks, $B_1, \ldots, B_k$, ordered according to the position in the reference sequence, and if some B_i contains sequence from the queried informant and $j > 1$ is the smallest possible index such that B_j also contains sequence from the queried informant, then the queried informant sequences from the blocks B_i and B_j should be linearly consecutive. If this is not the case for some B_i and B_j, we say that the mapping does not exist.

For example, a mapping of query $((chr1, +, [102, 112]), informant1)$ based on the alignment showed in figure 1.4 does not exist because the first part of the query interval ($[102, 108]$) is aligned to the third chromosome ($chr3$) of $informant1$, but the second part of the query interval ($[109, 112]$) is aligned to the tenth chromosome ($chr10$) of $informant1$. If the sequence of $informant1$ in the second block came from its third chromosome, the mapping would be $(chr3, +, [27, 42])$.

Organism	Chromosome name	Chromosome position	Strand	Sequence
reference	chr1	100	+	GGATGGTGC-
informant1	chr3	25	+	GGATGG-GCA
informant2	chr1	48	-	GCACGGT-C-
informant3	chr1	115	+	GG--GGTGCA
reference	chr1	109	+	AGATCG-CGT
informant1	chr10	40	+	AG-TCGTCAT
informant3	chr1	132	+	AG-TCG-CGT
reference	chr1	118	+	CCGAGGT
informant2	chr1	56	-	CC-A--T

Figure 1.4: Sample alignment consisting of three blocks

This rule guarantees that if a mapping of a region based on several alignment blocks exist, it will be a group of linearly consecutive regions $(chr, s, [p_{1,1}, p_{1,2}]), \ldots, (chr, s, [p_{k,1}, p_{k,2}])$. Then the target region can be constructed as the minimal interval containing all the intervals from this group: $(chr, s, [p_{1,1}, p_{k,2}])$.

From the biological point of view, this rule only means that a region will be mapped to a region, but since the partition of an informant sequence into alignment blocks depends not only on the queried informant, but also on other informants, the rule is neccessary.

If the rule R3 is fulfilled and if we focus only on the query and target sequences, we can imagine a *virtual block* – a concatenation of the query and target data from all alignment blocks containing the queried region. If the target sequence is missing in some block, we insert in its place in the virtual block an appropriate number of gaps. If some part of the target region is not covered by any alignment block, we insert in the virtual block an appropriate number of 'N's in the target sequence and an appropriate number of gaps in the query sequence.

We can then apply the rules R1 and R2 to the virtual block and get a mapping even for long query regions which are spread over multiple alignment blocks.

Rule R4: There should not be subintervals of the query region or the target region aligned to gap symbols in the other sequence which are longer than g.

More precisely, let $A = \{(q_i, t_i)\}_{i=1}^{k}$ be the sequence of all pairs of aligned bases between the query and the target regions (this means that q_i-th base in the query region is aligned to the t_i-th base in the target region). Then for each consecutive pair of elements $(q_j, t_j), (q_{j+1}, t_{j+1}) \in A$ we should have $q_{j+1} - q_j - 1 \leq g$ and $t_{j+1} - t_j - 1 \leq g$. If this is not the case, we say that the mapping does not exist.

For example, the mapping of query $((chr1, +, [106, 112]), informant3)$ based on the alignment in figure 1.4 is $(chr1, +, [119, 135])$ where the subinterval $[123, 131]$ of the target sequence is entirely missing in the alignment and the base at the position 122 is aligned to a gap in the query sequence, meaning that there is a 10-bases long sequence in $informant3$ that does not correspond to the same part of the reference sequence as its neighborhood – a 10-bases long gap. We have $A = (106, 119), (107, 120), (108, 121), (109, 132), (110, 133), (112, 134)$, thus the biggest gap in the informant is $132 - 121 - 1 = 10$-bases long and the biggest gap in the reference is $112 - 110 - 1 = 1$-base long.

A similar situation would occur for query $((chr1, +, [102, 119]), informant2)$ which maps to $(chr2, -, [48, 57])$ where there is no target sequence aligned to the subinterval $[109, 117]$ of the query sequence, meaning that there is an 9-bases long gap in the alignment.

Similarly to the Rule R1, it is not possible to generally set one threshold value g on the longest permissible gap, because the answer depends on the type of the query region and the goal of mapping. This means that the tool constructing the mappings should be able to check the target and query regions for gaps of an arbitrary size set by the user.

As we have showed, a correct mapping of a region does not always exist, but if it does, we can construct it by the following method:

1. Find the query region in the alignment.
2. If the query region is contained in more alignment blocks, check if the rules R3 and R4 are fulfilled. If not, the mapping does not exist.
3. Concatenate the query and target data from all alignment blocks containing the query region to a virtual block.
4. Find the primitive mapping of the query region in the virtual block.
5. Adjust the mapping according to the rules R1 and R2.

While this method guarantees to find a correct mapping if it exist, in practice it is inefficient because of the third step. Therefore, our implementation will be slightly different, as we will show in chapter 3.

Chapter 2

Related work

In this chapter we introduce known tools for mapping positions and intervals between genomes, and related file formats.

2.1 MAF

Multiple Alignment Format (MAF) is a common file format used to store multiple alignments of various length and some additional information (e.g. the score assigned to the alignment at its construction by the alignment program, indicating how accurate the alignment is).

A MAF file typically contains several header lines with comments and meta-data about the alignment (version of the MAF, name of the scoring scheme used for the alignments, etc.), which are irrelevant for mapping. The rest of the file consists of a number of multiple alignments separated by blank lines.

The first line of each alignment in a MAF file always starts with the letter `a` and can contain values of several variables (e.g. the score of the alignment in the block). The rest of the lines in the alignment can be of various types distinguishable by the first letter. When mapping, only the lines starting with `s` are relevant, since they contain all the information of an alignment block as defined in chapter 1. An example of one alignment block is showed in figure 2.1.

The structure of the lines starting with `s` is as follows: the first entry after the `s` character is the ID of the chromosome to which the sequence from the line belongs, the second entry is the zero-based starting position of the sequence on the source chromosome, the third entry is the number of bases in the sequence, the fourth entry is the orientation of the source strand, the fifth entry is the length of the source chromosome and the last entry is the aligned sequence itself. The ID of the chromosome consists of an abbreviation of the species name, the version number of the sequence, dot and a

chromosome name. In the example alignment showed in figure 2.1, sacCer, sacPar and sacKud are abbreviations for yeast species *Saccharomyces cerevisiae*, *Saccharomyces paradoxus* and *Saccharomyces kudriavzevii*, respectively.

```
a score=235.0
s sacCer3.chrX      204671 21 + 745751 GTCGAGAAAATA-TTT------T-CAAAT
s sacPar.contig_342    140 22 -  52145 A---ATACAATG-TTT-AAT--TCCTAAT
s sacKud.Contig1941  17931 29 +  19631 AAAGACAGAATAATTTAAATGATCCTAGG
```

Figure 2.1: Example of a an alignment block from a MAF file

In the first line of each alignment block in the MAF file is the sequence from the reference organism, and the alignment blocks in MAF file are usually ordered according to its chromosomal position.

Note that MAF files are redundant with respect to mapping: to construct an alignment it is neccessary to know what letters the sequences consist of, but this information is not needed to map between already aligned sequences. To carry out the mapping correctly, the only information neccessary is whether a character in an alignment is representing a gap, or a nucleotide (or, similarly, for each nucleotide in a sequence, whether it is aligned to a character representing a gap or a nucleotide).

2.2 BED

Browser extensible data (BED) is a file format for storing regions (as defined in chapter 1) and some additional data about them (e.g. data relevant for visualization using the UCSC genome browser [Ken+02]). Each line of a BED file contains the information about one region in multiple fields delimited by tab character. The meaning of the field values is determined by their position.

Each BED line must contain the following fields in this order:

1. The name of the region's source chromosome.
2. The position on the chromosome on which the region begins.
3. The position on the chromosome on which the region ends.

The positions in BED are zero-based and the intervals are half-opened, meaning that the interval from the region $(chr1, +, [10, 20])$ as defined in chapter 1 1 is in BED format represented by starting position 10 and ending position 21. Besides, the positions are relative to the start of the + strand, meaning that the interval from the region $(chr1, -, [10, 20])$ in a 25-bases long $chr1$ is in BED format represented by starting

position 4 (corresponding to position 20 in our representation) and ending position 15 (corresponding to position 9 in our representation).

The remaining fields are optional, however, if the i-th field is present, each field with index lower than i must be present too. Also, the number of fields should be the same for each line in a BED file. The optional fields are as follows:

4. Name of the region.
5. Score of the region. Meaning of this value varies depending on the type of the region.
6. Source strand of the region. (Since the interpretation of the positions is independent of the strand, for some uses the information about the strand can be irrelevant.)
7. Position on which the thick visualization of the region starts
8. Position on which the thick visualization of the region ends.
9. Color in format R, G, B (where $R, G, B \in \{0, \ldots, 255\}$) used for visualization of the region.
10. Count of subregions of the region. When the region represents a gene, these subregions often represent the gene's exons.
11. Sizes of the subregions separated by commas.
12. Starting positions of the subregions separated by commas.

An example of a BED line representing the `AATACAATG` region of *Saccharomyces paradoxus* showed in the alignment in figure 2.1 is showed in figure 2.2. Our representation of the same region, as defined in chapter 1, is $(contig\_342, -, [140, 148])$.

```
contig_342    51996    52005    shortRegion    0    -
```

Figure 2.2: Example of a BED line

2.3 liftOver

Since occasionaly new corrected versions of sequences are published for already sequenced species, the UCSC Genome Bioinformatics Group developed liftOver [TJ05],

a utility designed to map intervals and positions from one version of a sequence to another, so already computed annotations (notes associated with regions in the genome) do not have to be recomputed from scratch.

The liftOver utility is a part of the UCSC genome browser [Ken+02], an online multipurpose tool, but it can be also downloaded as a standalone program. Aside from mapping the annotations between two versions of the same sequence, it can be used for mapping intervals between genomes of different species as well. The input format of the intervals to map is BED.

To map between genomes of two species, liftOver needs a file with their (pairwise) alignment in the chain format [Ken+03], where the alignment is represented by a set of *chains*, each consisting of a header line and a list of *aligned blocks*. An aligned block represents a part of the alignment without gaps in any of the two sequences. Two aligned blocks are in the alignment separated by gaps, which may be simultaneously in both genomes.

The order of the blocks in a chain must be the same as the order of the corresponding parts of DNA in the genome sequences of both species; hence the blocks in a chain cannot represent a rearrangement of the DNA segments in an alignment (caused e.g. by a large-scale mutation). This problem is solved by skipping all the inversed, translocated and duplicated segments when constructing the chain.

A block in a chain consists of three numbers: the size of the corresponding part of the alignment and the number of nucleotides between the end of this block and the start of the next block in both sequences. The last block consists only of the first mentioned number. An example of a chain corresponding to the alignment of *Saccharomyces cerevisiae* and *Saccharomyces paradoxus* from the figure 2.1 is showed in figure 2.3.

The structure of the header line is as follows: the first entry is the keyword `chain`, the second entry represents the score of the alignment, the last entry is the chain ID. The rest of the header entries represents information about the aligned regions (chromosome ID and length, strand, starting and ending positions).

```
chain 235.0 chrX 745751 + 204671 204692 contig_342 52145 - 140 162 1
1 3 0
11 0 3
1 0 1
5
```

Figure 2.3: Example of a chain.

To create the chain file, each block of the pairwise alignment is converted into

a chain. When mapping, the chains are read into the memory and indexed using a hierarchial binning scheme. For each chromosome in the first genome a hierarchial structure of its subintervals represented by bins is created: one top-level bin representing the whole chromosome, which is divided into eight second-level bins divided into third-level bins etc. Each bin stores a linked list of chains fully contained in the bin's interval, and each chain is stored in the smallest bin which covers it.

When mapping a region, liftOver linearly finds the chains containing the query region endpoints in the bin corresponding to the query region and then walks through the chains to find the positions in the target sequence corresponding to the query region endpoints. Thus the time complexity is in the worst case linear, but in practice bins contain only a small number of chains and chains contain only a small number of chain blocks, so in the average case the performance of liftOver is sufficient.

liftOver supports several mapping settings such as allowing the output to contain multiple regions for one query region. It also maps not only the region itself, but also its subregions when present (the eleventh and twelfth BED field).

The liftOver mapping is, however, not a correct mapping as defined in chapter 1. It fullfils rules R1 and R2, since blocks in chains represent ungapped alignments, and thus the query region is mapped to a region in the target organism and the target region endpoints correspond to nucleotides in the query sequence. However, it does not fullfil rules R3 and R4.

Instead, liftOver uses a different definition of correct mapping: it does not check the length of the gaps in the query or target regions, but the user can set the minimum fraction of bases that must be mapped.

2.4 Libmultialn

In 2012, Michal Petrucha developed a C++ library Libmultialn for mapping between genomes based on the MAF representation of the alignment [Pet12]. To perform the mapping, it reads the alignment in a MAF format, builds data structures in memory and then quickly maps multiple regions using operations *rank* and *select*.

Rank and select are operations on a binary vector $B = (b_1, ..., b_n)$ defined as follows:

- $rank(i, B) = \sum_{j=1}^{i} b_j$
- $select(i, B) = \min_{rank(j,B)=i} j$

The Libmultialn library extracts from the redundant MAF file only the information relevant to mapping: the meta-information about the sequences (chromosome names,

lengths of sequences etc.) and the alignments themselves, which it converts into binary sequences by changing the letters (bases) to ones and dashes (gaps) to zeros.

The mapping of the query region is then constructed as the interval between the mappings of the the query region endpoints. The endpoints are mapped using rank and select operations: if B_T is the binary sequence for the target sequence and B_Q for the query sequence, Libmultialn maps positions in query to position $rank(select(i, B_Q), B_T)$ in the target. The use of rank and select and the representation of alignments is described in details in chapter 3.

For rank and select operations on binary vectors there exist succint data structures, meaning that the space needed for additional indexing information is asymptotically smaller than the space taken up by the original vector. Yet even this small additional space is sufficient to achieve constatnt operations [Gon+05], [RRR02]. Therefore, also the running time of position mapping (and thus interval mapping) for the Libmultialn, which uses such implementations of rank and select, is constant.

However, after including reading the input the running time is linear in the size of the input alignment file, which is impractical when there are only few query intervals to be mapped.

Libmultialn also does not produce correct mappings in the terms of rules R1-R4 defined in chapter 1. Although it fullfils rules R1 and R3 (Libmultialn requiers the target subregions to be colinear to the query subregions, condition analogous to our linearly consecutive target subregions), it does not check if the rules R2 and R4 are met.

2.5 HAL

The recently published Hierarchial Alignment format (HAL) and tool set [Hic+13] are designed to store multiple whole-genome alignments with index on all contained genomes, providing access to each DNA segment in constant time. HAL represents the alignment as a graph.

To create a HAL representation of a multiple alignment, the alignment data and a tree with vertices representing the organisms from the alignment are needed. The HAL representation is most beneficial when a *phylogenetic tree* is provided – a tree representing the evolutionary relationships between species. For this representation, organisms are used as vertices, and a child-parent graph relationship represents a descendant-ancestor evolutionary relationship. An example of a phylogenetic tree is showed in figure 2.4. In this tree, species denoted by S_1, S_2 and S_3 are living species, S_4 is the common ancestor of S_1 and S_2, S_5 is the common ancestor of S_3 and S_4.

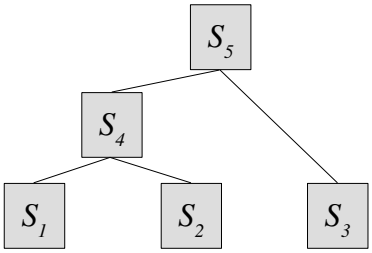

Figure 2.4: Example of a phylogenetic tree

The alignment data can be converted to a HAL graph from two different file types:

- MAF file. In this case, HAL can use an arbitrary tree with vertices representing the species from the alignment. If the user does not specify any tree, HAL uses a star-like tree with the reference organism in the root and informant organisms in the leaves.
- Cactus graph file [Pat+11], which also includes sequences of ancestral species inferred from the multiple alignment. In this case, a phylogenetic tree containing the organisms from the alignment and their ancestors is imported from the file and used.

The multiple alignment is decomposed into pairwise alignments according to the given tree – each alignment corresponding to a branch from the tree, aligning sequences of the species at the endpoints of the tree edge. The pairwise alignments are represented as subgraphs of the HAL graph with segments of original sequences (without gaps) as vertices and edges between corresponding segments. An example of the HAL graph corresponding to the phylogenetic tree showed in figure 2.4 is showed in figure 2.5. It represents the same alignment of S_1 = *Saccharomyces cerevisiae*, S_2 = *Saccharomyces paradoxus* and S_3 = *Saccharomyces kudriavzevii* as in figure 2.1, with added ancestral sequences S_4 and S_5.

In the HAL graph representation, there are at most three arrays stored for each organism: array with its whole genome sequence (in figure 2.5 the sequences in grey boxes), array with its vertices from the pairwise alignment graph with its parent from the tree (in figure 2.5 represented by green braces above the sequences) and array with its vertices from the pairwise alignment graphs with its children from the tree (in figure 2.5 represented by blue braces below the sequences). Since the vertices in the third array are the same for each child, the amount of segmentation of the parental sequence in the third array is determined by the number of mutations in all pairwise alignments with its children.

The HAL graphs are currently saved in a format designed to store large matrix

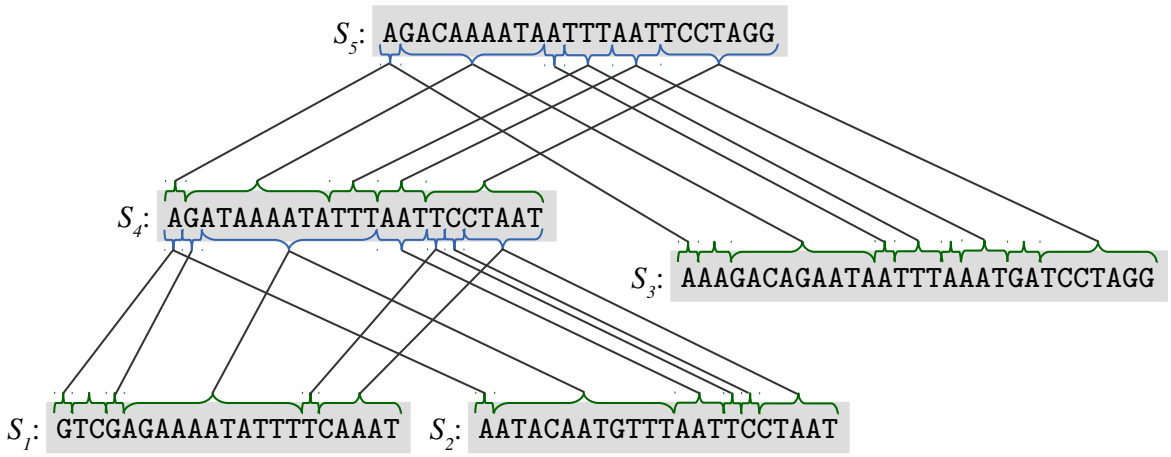

Figure 2.5: Example of a HAL graph

data, the Hierarchical Data Format (HDF5). It is optimized for compression and allows also random access to a specific portion of the file[Gro00]. HAL comes with a comprehensive API in C++ designed to create and query HAL files, and a set of command line utilities including tools to import alignments from MAF or Cactus files, and halLiftover, a mapping utility. This utility supports mapping between arbitrary query and target organisms via mapping the interval from organism to organism along the path from the query organism to the target organism in the tree. Although the input regions for halLiftover are in the BED format, it does not support mapping of neither the thick subregion determined by seventh and eighth BED field nor subregions from the elevent and twelfth BED field.

halLiftover usually does not output a single target region, but mutiple linearly consecutive subregions of the target region, thus fulfilling our rule R3. Rule R1 is also fulfilled by halLiftover mapping (since alignments in HAL graphs contain sequences without gaps), but rules R2 and R4 are not enforced (halLiftover does not check whether the target interval endpoints correspond to bases in the query sequence, and it also does not check the size of the gaps between two subregions neither in the target nor in the query sequence).

Chapter 3

Maptool

In this section we describe design and implementation of our program, Maptool, and its comparison with liftOver and HAL tools.

3.1 Overall design

Since the alignment data are growing bigger every day, we designed our application so that the input alignment file (consisting of multiple alignment blocks as described in chapter 1) has to be read only once. This phase is called preprocessing. Upon preprocessing, two new files are created:

- a compressed file containing data about how bases correspond to each other (i.e. the only data from alignment sequences relevant to mapping),
- a *header file* containing:
  - information neccessary to locate the data about a specific alignment in the compressed file,
  - sequence *source data* – relevant information about sources of the sequences (the list of the organisms, their chromosomes and the sizes of those chromosomes).

The second phase, mapping, uses only the header file and the compressed file. Also, typically only some parts of the compressed file are read and used, which significantly speeds up the mapping.

The preprocessing is written in Python and the mapping in C++. The format of the input alignments is MAF, the format of the input query and output target regions is BED (our program supports also mapping of subregions from the optional fields).

3.2 Representation of alignments

Since the alignment data are redundant, we extract and store only the relevant information. When designing the mapping process, we considered three different representations of the alignments. Note that while their memory and time complexities are asymptotically the same, there are some differencies in practice due to the size of the alignments.

For an alignment block, we will denote by k the number of informants, by n the number of bases in the reference sequence, by $m_1, \ldots, m_k$ the numbers of bases in the informat sequences, by m the maximum of $\{m_1, \ldots, m_k\}$, and by s the length of the aligned sequence (which is the same for each aligned sequence in one alignment block). In practice for a typical alignment block containing up to six species, the following holds: $s \leq \frac{3}{2} \cdot n$ and $s \leq \frac{3}{2} \cdot m$, meaning also that $n \leq \frac{3}{2} \cdot m$ and $m \leq \frac{3}{2} \cdot n$.

Since our program uses rank and select to perform the mapping, to achieve constant time complexity, additional data structures are needed for each aligned sequence. These structures take up $\frac{s}{r}$ additional space, where r is a fixed constant, for each aligned sequence in the block. Our implementation creates these additional structures when reading the aligned sequence data from the compressed file.

1. **Binary representation of reference-informant pairs of the original (unaligned) sequences.**

   For each informant in each alignment block create two binary vectors $R = (r_1, \ldots, r_n), I = (i_1, \ldots, i_m)$, representing pairwise alignment of these two sequences. The binary vectors indicate for each base in the original sequences whether it is aligned to a base in the other sequence, or to gap. In particular, r_j (resp. i_j) equals `1` iff the j-th base in the reference (resp. specified informant) is aligned to a base and `0` otherwise. An example is showed in figure 3.1.

`TACGA-GGTT-A`		R_1, I_1		R_2, I_2
`T---ACGG--CA`	$\mapsto$	`1000111001`	,	`0001110111`
`---GACT-TTCG`		`1101101`		`11011101`

Figure 3.1: Example of the first considered representation

   Using this representation, the mapping of the j-th base to a base in the i-th informant can be constructed as $select(rank(j, R_i), I_i)$. First, rank returns k, the count of bases which are in the original reference sequence on position j or before it and are aligned to a base in the i-th informant. Then select gives us the

position of the base in informant sequence, on which lies the k-th base aligned to a base in the reference.

Complexities for one alignment block are as follows:

- Space $s_1(s, n, m, k) = k \cdot n + \sum_{i=1}^{k} m_i = O(k \cdot (n + m)) = O(k \cdot s)$.
- Loading of the block from the file: $s_1(s, n, m, k) + \frac{s_1(s,n,m,k)}{r} = O(k \cdot s)$.
- Mapping: $O(1)$ using rank and select.

2. **Binary representation of the aligned sequences.**

   For each aligned sequence A (for the informant sequences as well as for the reference sequence) create binary vector $B = (b_1, \ldots, b_s)$, where b_j equals `1` iff j-th character in A is a base, i.e. the binary vector indicates for each character in A whether it represents a base, or a gap. An example is showed in figure 3.2. Representation similar to this one was used also by library Libmultialn, described in section 2.4.

```
TACGA-GGTT-A        111110111101
T---ACGG--CA   ↦    100011110011
---GACT-TTCG        000111101111
```

Figure 3.2: Example of the second considered representation

   Using this representation the mapping of the j-th base to a base in the i-th informant can be constructed as $rank(select(j, B_R), B_{I_i})$, where B_R, B_{I_i} are the representations of the aligned sequences of the reference and i-th informant, respectively. First, select returns the position of the j-th base in B_R. Then rank gives us the position of the base in original informant sequence, which is aligned to the character in the reference sequence on the position returned by rank.

   Complexities for one alignment block are as follows:

   - Space $s_2(s, n, m, k) = (k + 1) \cdot s = O(k \cdot s)$.
   - Loading of the block from the file: $s_2(s, n, m, k) + \frac{s_2(s,n,m,k)}{r} = O(k \cdot s)$.
   - Mapping: $O(1)$ using rank and select.

3. **Numeric representation of the aligned sequences.**

   For each aligned sequence A create a list of pairs in the following way: for each maximal substring of bases in A add to the list a pair (p, l), where p is the position

of the substring in A and l is its length. The number of gaps between two maximal substrings of bases represented by pairs $(p_1, l_1), (p_2, l_2)$ is then $p_2 - (p_1 + l_1)$. An example of this representation, similar to the chain format described in section 2.3, is showed in figure 3.3.

```
TACGA-GGTT-A       (0, 5), (6, 4), (11, 1)
T---ACGG--CA   ↦   (0, 1), (4, 4), (10, 2)
---GACT-TTCG           (3, 4), (8, 4)
```

Figure 3.3: Example of the third considered representation

Note that this representation is only a slight modification of the second representation, hence, if we have available adequate additional data structures similar to the data structures needed by rank and select, the mapping using this representation can be constructed in a way similar to the construction using the second representation.

Complexities for one alignment block are as follows:

- Space $s_3(s, n, m, k) = (k + 1) \cdot c \cdot s = O(k \cdot s)$, where c is the coefficient identifying the ratio of maximal bases-substrings in the alignment.
- Loading of the block from the file: $s_3(s, n, m, k) + \frac{s_3(s,n,m,k)}{r} = O(k \cdot s)$.
- Mapping: $O(1)$ using the additional data structures.

After careful consideration, we chose the second representation because of its significant space complexity advantage over the first representation when working on multiple-genome alignment: for $k > 2$ is $s_2(s, n, m, k) = (k+1) \cdot s \leq (k+1) \cdot (\frac{3}{2} \cdot \frac{1}{2} \cdot (n + m)) \leq k \cdot (m + n) = s_1(s, n, m, k)$ – the first representation creates k binary vectors to represent the reference sequence, while the second representation needs only one vector for the reference sequence. (The length of each such vector in the first representation is equal to the number of bases in the reference sequence, while in the second representation is its length equal to the length of the aligned reference sequence. However, for a bigger number of informants is the space needed by additional vectors bigger than the space needed to represent gaps.) Compared to the third representation, the space complexity of the second representation is stable regardless of how long is the maximal substring of bases on average.

The chosen representation decreased the amount of space needed to store the input data approximately by 85% (when stored in a binary file) in comparison with the size of the MAF file. Even more space was spared by compression of the alignment data.

3.3 Preprocessing

The main goal of preprocessing is to create a compressed file and a header file containing the relevant information from the MAF file in the selected representation. Our program does not convert each alignment block from the input alignment file into its binary represention separately. Instead, if possible, it combines several alignment blocks into a *compound block*, which it then converts to its binary representation. A compound block consists of one long reference sequence, its source data, and a list of informant structures (one structure for each informant species). The informant organism includes several sequences, each covering part of the reference sequence, and for each sequence its source data.

Source data for both reference and informant sequences include chromosome ID, chromosome position, strand (i.e. + or −) and the count of bases in the sequence. For an informant sequence we also store the position relative to the reference sequence start, at which the alignment of this informant sequence starts.

Since the alignment blocks are often only a dozens of bases long, compound blocks ensure that instead of a great number of small alignments our program works with a smaller number of bigger alignments, thus reducing the time needed to find the block containing a given position without increasing the time needed to find the corresponding bases (which is still $O(1)$ using rank and select). However, the bigger the block is, the bigger are the data that have to be read from the compressed file in the mapping phase regardless of the size of the region to be mapped. Therefore we have to set an upper bound on how big the compound block can be. Empirically we chose this bound to be 64 kB.

Adding an alignment block to a compound block proceeds as follows:

1. Check if the size of the compound block with added alignment block is less than the upper bound for the size of the compund block. If no, and the compound block is not empty, do not proceed. (The alignment block will be added to the next compound block.)

2. Check if at least one of these conditions is fulfilled:

   - The compound block is empty (i.e. the lengths of the compound block's reference sequence and the informant structure list are both 0).
   - The compound block is not empty, the reference sequences from the compound and the alignment block form two adjacent regions on the same chromosome and they are linearly consecutive.

   If neither of the conditions holds, do not proceed.

3. Append the reference sequence from the alignment block to the compound block's reference sequence.

4. For each informant sequence in the alignment block:

   - Check whether this informant is already contained in the informant structures list. If not, add an informant structure representing this informant species to the list.
   - Add the informant sequence and its source data to the informant structure.
   - Add to the informant structure the position in the compound block's reference sequence at which the alignment of the informant sequence starts (this position is equal to the length of the compound block's reference sequence before appending the new part of the reference sequence in step 3).

The compound blocks are stored in the compressed file so that each compound block can be read without having to read any other blocks. The header file stores a sorted list of tuples (p, n, c), each tuple representing one compound block, where p is the reference sequence chromosome position, n is its length and c is the position of the compound block in the compressed file.

Given an input alignment file (consisting of alignment blocks ordered according to the chromosome position of the reference sequence), the whole preprocessing is carried out in this way:

1. Create a new empty compound block. We will call this block the current block.

2. Check if the end of the input alignment file is reached:

   - If yes, end the preprocessing, but if the current block si not empty, first add its binary representation to the compressed file and add the corresponding tuple (p, n, c) to the header file.
   - If not, read an alignment block from the input alignment file.

3. Try to add the read alignment block in the current compound block:

   - If the addition suceeds, continue with step 2.
   - If not, add the current block's binary representation to the compressed file and add the corresponding tuple (p, n, c) to the header file. Create a new empty compound block and make it the current block. Then repeat step 3.

Note that while it seems that the preprocessing may loop infinitely by repeating the third step, thanks to the conditions of adding an alignment block to a compound

block there will never be more than two repeats of the third step: when the current compound block is empty, addition of any alignment block succeeds.

The time complexity of the preprocessing phase is linear in the size of the input alignment file.

For the compression format we chose BGZF, block compression format build on top of the standard gzip file format [Li+09]. This format joins multiple compressed blocks to one file and allows fast access to any of the compressed blocks it contains.

Each compound block is compressed when added to the compressed file, becoming one of the compressed blocks. To access the blocks in the compressed file we used two different libraries – Biopython [Coc+09] for the preprocessing phase and ocaml-bgzf [Lin11] for the mapping phase.

The size of our preprocessed compressed file in BGZF format is typically about 2-5% of the size of the original MAF file (depending on how many lines irrelevant to mapping the MAF file contains).

3.4 Mapping

Having preprocessed the alignment to the compressed file and the header file, we can start mapping. In this section, by *sequence* we mean the binary representation of an aligned sequence.

For the input BED file with query regions, our program first reads the entire header file into memory. The mapping of each BED line then consists of 4 steps:

1. Load the compound blocks containing the queried region from the compressed file.

2. Create an empty output BED line.

3. For each region r from the input BED line (the query region, the thick subregion determined by seventh and eighth field and each subregion determined by eleventh and twelfth fields) do the following:

   (a) Map the region r in a way fulfilling the rules R1 and R2 by mapping its endpoints. If it is not possible, do not proceed.

   (b) Check whether the target region fulfills also rules R3 and R4. If not, do not proceed.

   (c) Fill the corresponding fields in the output BED line. If it is not possible, do not proceed.

4. Output the BED line filled in the previous step.

In the following paragraphs, we will explain the first and the third step in detail.

3.4.1 Loading the alignment data

Since loading the compound blocks is computationally expensive, our program stores the last ten read blocks in memory, indexed by their position in the compressed file.

Upon request to load compound blocks containing the queried region, our program first finds the corresponding tuples in the header file data using binary search and then checks if the wanted blocks are already in memory. If yes, it simply returns them. If not, the blocks have to be loaded from the compressed file.

After reading a block from the compressed file, we construct an additional array A_S for each sequence S in this block. This array stores data used to efficiently compute rank (for target sequence) or select (for query sequence). Values in A_S are computed as follows:

- For rank, $A_S[i]$ stores the value $rank(i \cdot 32, S)$.
- For select, $A_S[i]$ stores the value $select(i \cdot 32, S)$.

These arrays are a simplified version of the succint data structures for rank and select (mentioned in section 2.4) and are later used in step 3a.

3.4.2 Mapping a region

To compute $rank(i, S)$ or $select(i, S)$ for sequence S and integer i in constant time, the array A_S is used. First, let $S' = S[i-(i \mod 32)+1 : i]$ (i.e. the part of S starting at $(i-(i \mod 32)+1)$-th index and ending at i-th index). Then $rank(i, S) = A_S[\lfloor \frac{i}{32} \rfloor] + rank(i \mod 32, S')$, and $select(i, S) = A_S[\lfloor \frac{i}{32} \rfloor] + select(i \mod 32, S')$. Note that since the size of S' is at most 32, $rank(j, S')$ resp. $select(j, S')$ can be for arbitrary j computed in constant time by iterationg through at most 32 elements of S'. The value $A_S[\lfloor \frac{i}{32} \rfloor]$ is also accessed in constant time, making the overall complexity of rank and select constant.

The mapping of each region r from the BED line (step 3a) is constructed as the region between the mappings of r's endpoints. For each endpoint p, we compute the position $i = select(p, S_R)$ in the reference sequence S_R using the precomputed array A_{S_R}. Then, using binary search in the list of target sequences, we find the sequence S_I that contains bit aligned to i-th bit in S_R. Then we compute $p' = rank(i - s, S_I)$, where s is the position in S_R on which the alignment of S_R and S_I starts.

After conducting multiple tests, we found that although in theory the time complexity of computing $p' = rank(i - s, S_I)$ using A_{S_I} is $O(1)$, in practice it is faster to find p' by a linear-time algorithm counting the ones in S_I, since the target sequences are on average fairly short and computing the array A_{S_I} in step 1 causes significant a overhead.

The resultig position p' is then in linear time incremented or decremented until it fulfills the rule R2 and fits the type of mapping (inner or outer). Finally, position p' in the target sequence is the mapping of position p from the query sequence.

The checking of the mapped region (step 3b) is conducted by linearly walking through all the target sequences overlapping the target region which correspond to the query region. Both rules R3 and R4 are verified in the same traversal. Although in theory we need to check the size of the gaps both inside the alignment blocks and between them, in practice we assume that the gaps inside blocks are small and therefore our program checks only the sizes of the gaps between original alignment blocks in both query and target sequences (note that several alignment blocks are now concatenated to one compound block).

In the step 3c, the fields of output BED line are simply filled in with the target region data. But if the target regions consists of several subregions, we check whether all subregions are from the same chromosome and strand as the target region, and for the subregions from the eleventh and twelfth field we also check if they are linearly consecutive.

3.4.3 Time complexity

Our program maps the regions in a way fulfilling rules R1-R4, thus creating correct mappings as defined in chapter 1.

For q query regions of maximal length n and a compressed file containing b compound blocks are the time complexities of loading, mapping and checking the fulfillment of R3 and R4 – $l(n, b), m(n)$ and $c(n)$, respectively, as follows:

- Since the minimal size of the block to be loaded from the compressed file regardless of the query region size is 64kB = B, and in the case that the query region overlaps two blocks it is $2B$, $l(n, b) = O(\log_2 b + max(n, 2B))$.
- The time complexities of mapping of the endpoints and checking the fulfillment of R3 and R4 are both also in the worst case linear:
  - $m(n) = O(n)$ if the whole query region is covered by one target sequence,

– $c(n) = O(\frac{n}{s})$ where the query region is covered by $\frac{n}{s}$ target sequences of length s.

Although the worst case time complexity of the mapping phase is $O(h+q\cdot(l(n,b)+m(n)+c(n))) = O(h+q\cdot(n+\log_2 b+B))$ where h is the size of the header file, in practice the loading from compressed file is unneccessary for many queries, the average size of the query region is usually less than 5000 and the target sequences are also fairly short. Therefore the performance of our program is in average case sufficient and as we will show in the following section, it is also better than the performance of the existing tools.

3.5 Comparison of HAL, liftOver and Maptool

In this section we measure the running of our program, Maptool, and compare it with the running time of published tools halLiftover and liftOver.

The tests were performed on a machine running 64-bit Linux 2.6.32 with 8 Intel Xeon CPU cores running at 2.26 GHz, 8 MB CPU cache and 23 GB of RAM. We used the GCC 4.8 C++ compiler with -O2 optimizations.

All tests were executed multiple times.

3.5.1 Preprocessing

To test the speed of preprocessing, we have used four data sets, each consisting of one human chromosome [Con01] aligned to 4 vertebrates – chimpanzee [The05], mouse [Mou+02], rat [Gib+04], [Hav+04] and chicken [Hil+04]. The alignments used are from the UCSC Genome Browser website [UCS].[1]

The sizes of the used input files and the lengths of the chromosomes are showed in the table 3.1.

Chromosome	Chromosome length	Size of file
Chromosome 2	243,615,958 nucleotides	741.2 MB
Chromosome 13	113,042,980 nucleotides	287.5 MB
Chromosome 22	49,396,972 nucleotides	99.6 MB
Chromosome Y	50,286,555 nucleotides	29.2 MB

Table 3.1: Testing data for preprocessing

As can be seen, the size of the file does not depend only on the length of the

[1] For credits, see Appendix B.

chromosome which it covers – while chromosome Y is longer than the chromosome 22, its alignment is sparser, thus its alignment file is much smaller.

For every input alignment, we measured its preprocessing time and memory usage for both Maptool and HAL tool. The measurements are showed in figures 3.4 and 3.5. Comparison of the resulting files (BGZF compressed file and a header file created by Maptool and the HAL graph) is showed in table 3.2.

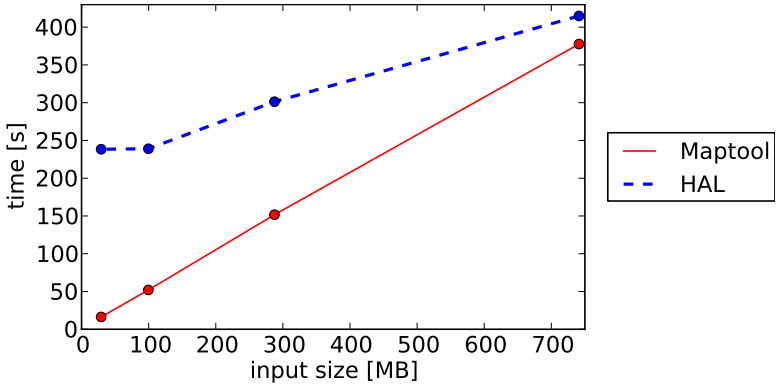

Figure 3.4: Comparison of running time of Maptool and HAL preprocessing

The running times of Maptool and HAL preprocessing are getting more alike with growing size of the alignment. Both are linear in the size of the input file.

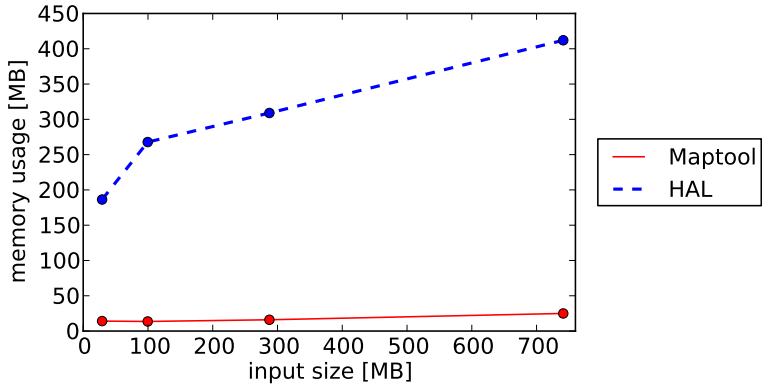

Figure 3.5: Comparison of memory used by Maptool and HAL preprocessing

However, Maptool has a much better memory usage, since it stores in the memory at most one whole compound block at a time and for the other compound blocks it stores only several numbers.

Since HAL graphs store more information about the alignment than our representation – not only the data needed to map, but also the DNA sequences – we expected that the size of HAL graphs will be greater than the size of BGZF compressed files created by Maptool. Nevertheless, we did not expect that the difference will be so big: the size of the preprocessed files created by Maptool is on average less then 6% of the size of the files created by HAL preprocessing.

	Chromosome	Alignment size	Compressed file size	Header size	Overall file size
HAL	2	741.2 MB	491,630 KB	-	491,630 KB
Maptool			28,836 KB	489.5 KB	29,325.5 KB
HAL	13	287.5 MB	195,050.5 KB	-	195,050.5 KB
Maptool			10,794.5 KB	185.1 KB	10,979.6 KB
HAL	22	99.6 MB	91,051.6 KB	-	91,051.6 KB
Maptool			4,229.2 KB	113.5 KB	4,342.7 KB
HAL	Y	29.2 MB	58,861.4 KB	-	58,861.4 KB
Maptool			1,353.1 KB	137.5 KB	1,490.6 KB

Table 3.2: Outputs of preprocessing

3.5.2 Mapping

To compare mapping speeds, we have constructed two sets of BED files. The first set contains one set of regions for each of the four chromosomes tested in the previous section. The second set contains files for chromosome 2 which differ by the number of the regions in them. Each region in the BED file identifies one exon of some gene. The data used were obtained via UCSC Table Browser [Kar+04], [Con01] from the table containing known genes for hg16 (version 16 of the human genome). We selected from them subsets with sizes of multiples of thousand by cropping the original files after the 1000th, 3000th, ..., 19000th region, thus obtaining exons from the beginning of the chromosomes.

The sizes of the used input files and the numbers of the queries are showed in tables 3.3 and 3.4.

Set 1		
Chromosome	Number of queries	Query file size
Chromosome 1	7,000	415.5 KB
Chromosome 13	5,000	308.3 KB
Chromosome 22	3,000	184.5 KB
Chromosome Y	1,000	57.9 KB

Table 3.3: Testing data for mapping (1)

In this test, the queries were mapped from the human genome to the chimpanzee genome by Maptool, halLiftover and liftOver. To map, both Maptool and halLiftover used the files created in the preprocessing phase. liftOver used four chain files containing alignment of human and chimpanzee for human chromosomes 1, 13, 21 and Y of sizes 4.84 MB, 1.91 MB, 0.95 MB and 1.9 MB, respectively. The source of these files is also the UCSC Genome Browser website [UCS], [Con01], [The05].[2]

[2]For credits, see Appendix B.

Set 2		
Chromosome	Number of queries	Query file size
Chromosome 2	7,000	415.5 KB
Chromosome 2	9,000	534.7 KB
Chromosome 2	11,000	654 KB
Chromosome 2	13,000	778 KB
Chromosome 2	15,000	903.2 KB
Chromosome 2	17,000	1,027.8 KB
Chromosome 2	19,000	1,152.5 KB

Table 3.4: Testing data for mapping (2)

In this subsection, we will refer to all these files as the alignment data.

The results of the time and memory usage measurements for dataset 1 are showed in figures 3.6 and 3.8 and for dataset 2 in figures 3.7 and 3.9.

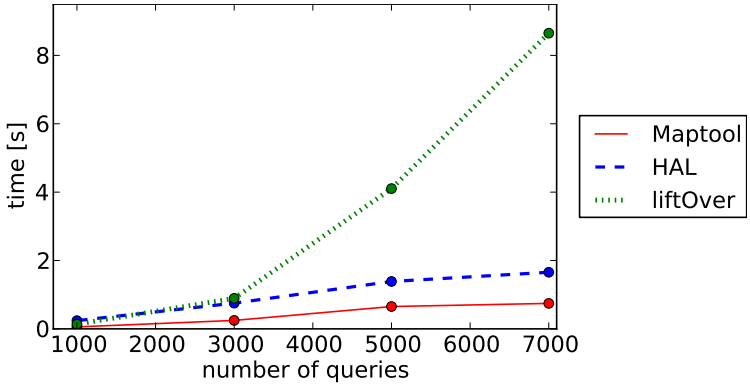

Figure 3.6: Comparison of running time of Maptool, HAL and liftOver for dataset 1

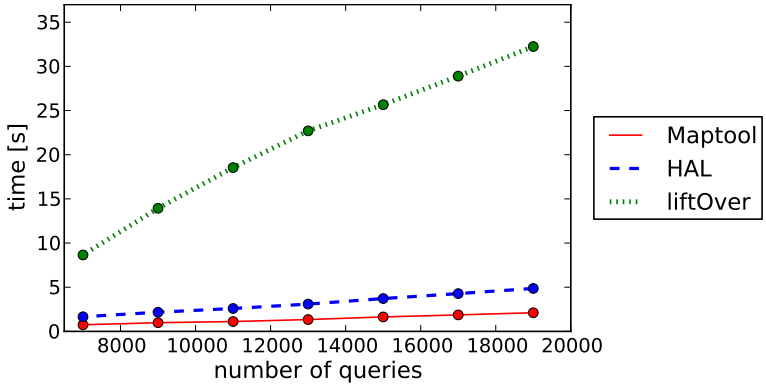

Figure 3.7: Comparison of running time of Maptool, HAL and liftOver for dataset 2

Since the representation of the alignment data that liftOver uses contains sequence only for two species which are in this test closely related, liftOver needs to read only a small file. This helps it to keep up with HAL and Maptool for smaller sets of regions, where reading the alignment data takes a significant portion of the running time.

However, for a bigger number of query regions, liftOver gets slower. Moreover, running time of liftOver grows with increasing number of queries faster than running time of Maptool, which is for larger numbers of queries up to 2-times faster than halLiftover and up to 15-times faster than liftOver. The reason is most likely that Maptool needs more time to load the alignment data than liftOver and halLiftover, but it executes the mapping itself faster.

Thanks to the slimmer representation of the alignment data, liftOver has also the lowest memory usage. The most memory is required by halLiftover, Maptool needs only slighlty more memory than liftOver. Also, the memory usage of both halLiftover and liftOver depends mostly on the size of the alignment data, while the Maptool's memory usage depends observably also on the number of queries.

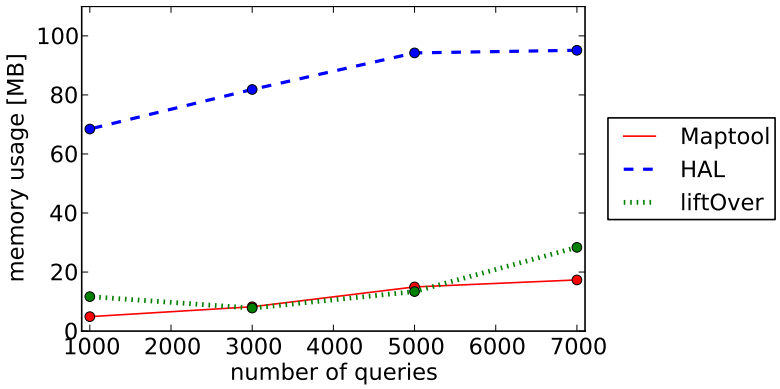

Figure 3.8: Comparison of memory used by Maptool, HAL and liftOver for dataset 1

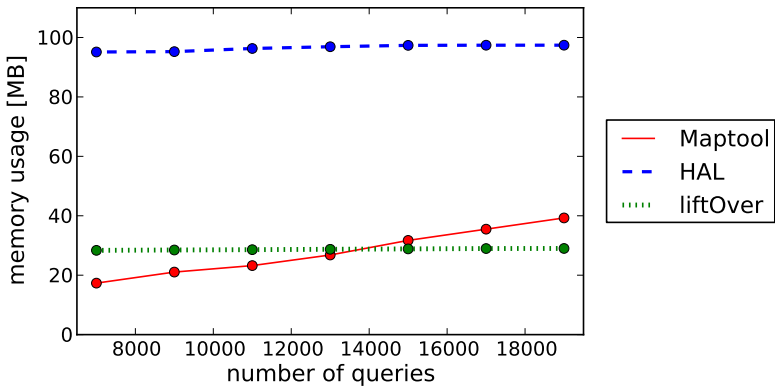

Figure 3.9: Comparison of memory used by Maptool, HAL and liftOver for dataset 2

Finally, we compared the mapped regions, but first the halLiftover outputs were processed so that each set of output regions for the same input was joined into one output region.

For Maptool and liftOver, less than 0.8% of the mapped regions from chromosomes 2, 13 and 22 were different. On the other side, nearly 27% of the mapped regions from

the chromosome Y differed. However, Maptool mapped only 33.4% of the regions located on chromosome Y, while for other chromosomes it mapped 77-87% of the regions. liftOver mapped for each chromosome except Y 91-96% regions and for chromosome Y 76.5%.

There are two possible reasons for different success of liftOver and Maptool in mapping: one is that the chain files are different from the MAF alignments, and the other is that liftOver uses different rules to determine whether a mapping is correct and it does not check the sizes of gaps. When we changed the size of maximal allowed gap for Maptool from 10 to 50, it successfully mapped 82-92% query regions from the chromosomes 2, 13 and 22.

The regions mapped by halLiftover differed from the the outputs of both Maptool and HAL in a slighlty bigger number of cases, 3-7% for chromosomes 2, 13 and 22. However, these differencies occur predominantly when the target region lies on the $-$ strand and for the target regions on $+$ strand most mappings are equal. Therefore, a possible explanation is that halLiftover makes mistakes when mapping on the reverse strand.

3.5.3 Summary

As we have shown in this section, our program offers a better solution for mapping as HAL, since it uses less memory for both preprocessing and mapping, and maps more efficiently, while the running time of preprocessing is comparable.

Also, while for small sets of queries the performance of liftOver mapping is better in terms of both time and memory usage than the performance of our implementation, for big sets of queries Maptool is faster.

However, it is important to keep in mind that these tools offer slightly different mapping possibilities: HAL can map between any two species from the alignment, Maptool can map from the reference species to multiple informant species and liftOver from the reference species to only one informant species. Of course, these differencies have impact on the amount of time and memory needed to construct a mapping.

Chapter 4

Theoretical aspects

In this chapter, we discuss theoretical aspects of the mapping problem. Althought the performance of our program is in average case sufficient, its worst case time complexity is linear in the size of the query interval. This is due to the three steps of mapping that take linear time: loading the data from the compressed file, mapping of positions and checking the target subregion's linear consecutiveness and the gaps' sizes (i.e. checking the fulfillment of rules R3 and R4, as defined in chapter 1). While the loading of the data from the compressed file is inevitable and clearly needs linear time, the theoretical complexity of the other two steps can be improved. Such improvements are meaningful if we want to execute more mappings on the same part of the alignment, or if we already have the alignment data in memory.

For the mapping, the theoretical time complexity can be improved by using the additional array constructed during the loading to perform rank on the target (informant) sequence, as described in subsection 3.4.2. Since in practice the informant sequences are fairly short, using rank does not change the running time much. To enhance the practical performance, several informant sequences can be merged into one in the same way the reference sequences are merged when adding an alignment block to a compound block, as described in section 3.3.

In the rest of this chapter we will focus on the problem of checking if rules R3 and R4 apply for a given region. First, we will reduce this problem to the well-known Range Minimum Query problem and then we will present a survey of data structures for this problem.

4.1 Problem formulation

In the following we will assume a modification of the alignment representation: bases will be represented by zeros and gaps by numbers denoting the size of each gap. An

example of this representation is showed in figure 4.1. Of course, to map in this representation of the alignment, operations rank and select have to be modified accordingly.

```
TACGA-GGTT-A       000001000010
T---ACGG--CA   ↦   033300002200
---GACT-TTCG       333000010000
```

Figure 4.1: Example of the modified representation of the alignment

We will also assume that both the query and target regions consist of one sequence. In practice, this can be achieved by merging more sequences into one by adding gap symbols (numbers greater than 0) if needed. Additionaly, we will mark each gap subsequence between two parts of the sequence which are not linearly consecutive (or, if there are no gaps between two not linearly consecutive parts of the sequence, we will add one gap symbol), and in the following we will call them *bad gaps*.

Subsequently, checking if some subregion of one of the sequences fulfills rules R3 and R4 (each two following subsequences of a sequence in the subregion should be linearly consecutive and there should be no gaps between them longer than g) is equivalent to checking if the subsequence corresponding to the given subregion does not contain any number bigger than g or a bad gap. And that is only a slight modification of the *Range Minimum Query problem*.

Range Minimum Query (RMQ) problem is the following task: to preprocess an array A of n numbers so that for queries $q(i, j)$ we can efficiently find the answer $RMQ(i, j)$ – the index of the smallest element in subarray $A[i : j]$ (i.e. the subarray of A starting with the i-th and ending with the j-th index). The modification of the RMQ problem, where the answer is not the index of the minimal element in $A[i : j]$, but the index of the maximal element, is called the *Range Maximum Query* (RMxQ) problem.

We will say that the RMQ (resp. RMxQ) problem has a solution with overall time complexity $\langle O(f(n)), O(g(n))\rangle$ (resp. $\langle O(f(n)), O(g(n))\rangle$-solution) if the preprocessing phase has complexity $O(f(n))$ and answering a query has complexity $O(g(n))$.

The problem of checking if rules R3 and R4 apply can now be solved in the following way:

1. Convert all numbers in the sequences representing bad gaps to symbols representing infinity.

2. Preprocess the sequences as if they were arrays for the RMxQ problem.

3. R3 and R4 are then fulfilled for each region $[i, j]$ in the reference sequence S_R and the informant sequence S_I for which $S_R[q(i,j)] = S_R[RMxQ(i,j)] \leq g$ and $S_I[q(i,j)] = S_I[RMxQ(i,j)] \leq g$.

The steps 1 and 2 can be carried out during the loading phase of the mapping, meaning that the time complexity of loading will be $max(O(n), O(f(n)))$ if we use a $\langle O(f(n)), O(g(n))\rangle$-solution of the RMxQ. The complexity of checking the fulfillment of rules R3 and R4 will then be $O(g(n))$.

4.2 Survey of RMQ data structures

In this section we will describe known data structures for solving the well-studied RMQ problem. We are interested in data structures that have a better time complexity of answering the queries than the trivial solution with no preprocessing phase which finds the index of the smallest element linearly, thus having overall complexity $\langle O(1), O(n)\rangle$.

4.2.1 Naïve solution

The naïve solution to this problem is to precompute answers to all possible queries. Since the precomputing can be done in $O(n^2)$ using dynamic programming, the naïve solution has overall complexity $\langle O(n^2), O(1)\rangle$.

4.2.2 Sparse Table algorithm

The naïve solution can be improved by precomputing only the answers to selected queries – the ones for which the length of the query interval $j - i + 1$ is a power of 2.

The answers can be precomputed using dynamic programing. When we want to compute $k = RMQ(i, i + 2^l - 1)$ for some i and $l > 0$, we need to have already precomputed the values $k_1 = RMQ(i, i + 2^{l-1} - 1)$ and $k_2 = RMQ(i + 2^{l-1}, i + 2^l - 1)$ (i.e. the answers for the first and the second half of the $A[i : i + 2^l - 1]$). Then k can be found in constant time as $k \in \{k_1, k_2\}$ such that $A[k] = min(A[k_1], A[k_2])$. Therefore, we start the precomputing with the answers $RMQ(i, i + 2^l - 1)$ for each i and $l = 0$, then we compute it for each i and $l = 1$, then for each i and $l = 2$ etc.

Since there are $O(n \log_2 n)$ answeres to be precomputed, the preprocessing can be done in $O(n \log_2 n)$.

Finding the answer for an arbitrary query $q(i, j)$ is then carried out as follows:

1. Compute the biggest l such that $2^l \leq j - i + 1$.

2. Find among the precomputed answers k_1, k_2 such that $k_1 = RMQ(i, i + 2^l - 1), k_2 = RMQ(j - 2^l + 1, j)$ (i.e. the answers for the two overlapping subintervals of the $[i, j]$ interval which fully cover it).

3. Then $RMQ(i, j) = k \in \{k_1, k_2\}$ such that $A[k] = min(A[k_1], A[k_2])$.

Hence, the overall complexity of this solution, called the Sparse Table algorithm [BFC00], is $\langle O(n \log_2 n), O(1) \rangle$.

4.2.3 Segment tree

Another solution with better preprocessing and worse query answering complexities uses a *segment tree* [Bra08]. The segment tree is a binary tree defined recursively as follows:

- The root stores data about the interval $[1, n]$.
- If some node stores data about the interval $[i, j]$, and $i < j$, then the left child of this node stores data about the interval $[i, \lfloor \frac{i+j}{2} \rfloor]$ and the right child data about the interval $[\lfloor \frac{i+j}{2} \rfloor + 1, j]$. If $i = j$, the node is a leaf.

For the RMQ problem, the data about an interval $[i, j]$ is the value of $RMQ(i, j)$.

The segment tree is a complete binary tree with n leaves, thus it has $2n - 1$ nodes. Since the value in each node can be computed from its children's values (or, if the node is a leaf, the value is trivially the same as the endpoints of this node's interval), the preprocessing can be done in $O(n)$.

Answering a query $q(i, j)$ then can be carried out by traversing the tree from the root to the leaves finding the nodes corresponding to subintervals that make up the query interval. We will denote by I_N the interval represented by the node N, by V_N the value stored in N and by N_L, N_R the left and the right child of N. The traversing is then done by the following recursive algorithm, starting with TRAVERSE(root), using the auxiliary function MIN(l, r).

function MIN(l, r):
  if $l \neq \infty$ **and** ($r = \infty$ **or** $A[l] < A[r]$) **then return** l
  else return r **end if**
end function

function TRAVERSE(N):
  if $I_N \subseteq [i, j]$ **then return** V_N **end if**
  $l, r \leftarrow \infty, \infty$
  if $I_{N_L} \cap [i, j] \neq \emptyset$ **then** $l \leftarrow$ TRAVERSE(N_L) **end if**

if $I_{N_R} \cap [i,j] \neq \emptyset$ **then** $r \leftarrow$ TRAVERSE(N_R) **end if**
return MIN(l, r)
end function

We will call the function call TRAVERSE(N) *k-long*, if it recursively calls itself for at least k of N's children and if k of these calls yield another call of the TRAVERSE function. For example, if TRAVERSE(N) calls both TRAVERSE(N_L) and TRAVERSE(N_R), but only the former calls TRAVERSE again and the latter simply returns V_{N_R}, the original call TRAVERSE(N) was 1-long.

Each call of the function TRAVERSE(N) will be of one of the following types:

1. At least one of the I_N's endpoints lies in $[i,j]$ and:
   (a) interval of one of the N's children is disjoint with $[i,j]$, or
   (b) interval of one of the N's children is a subinterval of $[i,j]$, or
   (c) I_N is a subinterval of $[i,j]$.
2. None of the I_N's endpoints lies in $[i,j]$.

In the first situation, TRAVERSE(N) cannot be 2-long: in the case 1a, TRAVERSE(N) will call TRAVERSE only for one of the N's children, in the case 1b one of the N's children's calls will return the child's value and in the case 1c TRAVERSE(N) will return V_N, not calling TRAVERSE for any of N's children. Therefore, TRAVERSE calls of the first type can be at most 1-long. In addition, if M is a descendant of N and call of TRAVERSE(N) was of the first type, call of TRAVERSE(M), if any, will also be of the first type.

In the second situation, clearly TRAVERSE(N) can be 2-long, but only if some part of $[i,j]$ lies in I_{N_L} and some part in I_{N_R}. Otherwise TRAVERSE(N) is at most 1-long. However, if TRAVERSE(N) is 2-long, TRAVERSE(N_L) and TRAVERSE(N_R) will be of the first type: at least one of the I_{N_L}'s and at least one of the I_{N_R}'s endpoints will lie in $[i,j]$.

How many 2-long calls of TRAVERSE are there in the whole computation of RMQ? If at the beginning of the traversal we start with a call of the first type, there will be no 2-long calls of TRAVERSE (since each descendant will have the first type as well). If at the beginning of the traversal the second situation occurs, there will be at most one 2-long call of TRAVERSE (since after the first 2-long call, only the first situation can occur).

Finally, we can estimate the total number of recursive calls. On the levels above the node in which the 2-long call occurs, there is at most one 1-long call per level. On the levels under the 2-long call, there are at most two 1-long calls per level.

Since the height of the tree is $\log_2 n + 1$, there are at most $2 \log_2 n + 1$ nodes for which 1-long or 2-long calls were done. For each such node, there could be at most one of their children called by a 0-long call; hence, together there will be at most $4 \log_2 n + 2$ nodes visited during the traversal, meaning that answering a query has time complexity $O(\log_2 n)$.

The overall complexity of this solution is therefore $\langle O(n), O(\log_2 n) \rangle$.

4.2.4 Even better solutions

There exists also a better solution for the RMQ problem with overall time complexity $\langle O(n), O(1) \rangle$ [BFC00]. The basic idea of this solution is the reduction of the RMQ problem to the problem of finding the lowest common ancestor of two nodes (u, v) in a tree (i.e. the node which is an ancestor of both u, v and lies furthest from the root), and its subsequent reduction to the $\pm$1RMQ problem – a variation of the RMQ problem where the adjacent values in A differ by $+1$ or -1.

Moreover, there exists also a succint version of the $\langle O(n), O(1) \rangle$-solution [Dav+13]. It creates a data structure with space usage $2n + o(n)$ bits and when answering the queries, it does not need the original array A. Even so, since for mapping we need to store the sequences anyway, the space usage of solution of our problem using this data structure would be $3n + o(n)$ bits.

However, the details of these data structures are beyond the scope of this thesis.

Conclusion

The objective of this thesis was to create a practical tool for efficiently mapping regions between genomes, an important task in bioinformatics. To reach this goal, we defined a set of rules characterizing a correct mapping and designed a solution based on these rules.

Our solution has two phases. In the first phase the alignmnet data, created by existing tools, are preprocessed and stored in two files, especially designed for this purpose. The second phase is fast mapping of regions based on the preprocessed data. We tested our tool thoroughly and implemented several optimizations to enhance its practical performance.

In this thesis, we also described already published tools for mapping and compared them with our tool. While for small sets of regions (containing up to 5000 regions) liftOver utility works faster, our solution is faster when mapping larger sets.

We also proposed a solution for checking the linear consecutivity and gaps sizes, a problem encountered when constructing a correct mapping. To this purpose we studied the well-known Range Minimum Query problem and various algorithms which solve it. In the future, it would be interesting to implement the solution.

Our tool maps only the reference genome to any selected informant species. Another possible direction of the future work could be the problem of mapping between two arbitrary genomes from the alignment data.

Bibliography

[BFC00] Michael A. Bender and Martin Farach-Colton. "The LCA Problem Revisited". In: *Proceedings of the 4th Latin American Symposium on Theoretical Informatics*. LATIN '00. London, UK, UK: Springer-Verlag, 2000, pp. 88–94. ISBN: 3-540-67306-7.

[Bla+04] Mathieu Blanchette et al. "Aligning multiple genomic sequences with the threaded blockset aligner". English. In: *Genome Res* 14.4 (4 Apr. 2004), pp. 708–15. DOI: 10.1101/gr.1933104.

[Bra08] Peter Brass. *Advanced Data Structures*. 1st. New York, NY, USA: Cambridge University Press, 2008. ISBN: 0521880378, 9780521880374.

[Coc+09] Peter J. Cock et al. "Biopython: freely available Python tools for computational molecular biology and bioinformatics." In: *Bioinformatics (Oxford, England)* 25.11 (June 2009), pp. 1422–1423. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btp163. URL: http://dx.doi.org/10.1093/bioinformatics/btp163.

[Con01] International Human Genome Sequencing Consortium. "Initial sequencing and analysis of the human genome". In: *Nature* 409.6822 (Feb. 2001), pp. 860–921. ISSN: 0028-0836. DOI: 10.1038/35057062. URL: http://dx.doi.org/10.1038/35057062.

[Dav+13] Pooya Davoodi et al. "Encoding Range Minimum Queries". In: *CoRR* abs/1311.4394 (2013).

[Gib+04] R. A. Gibbs et al. "Genome sequence of the Brown Norway rat yields insights into mammalian evolution". In: *Nature* 428.6982 (Apr. 2004), pp. 493–521. DOI: 10.1038/nature02426. URL: http://dx.doi.org/10.1038/nature02426.

[Gon+05] Rodrigo González et al. "Practical implementation of rank and select queries". In: *In Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA'05) (Greece*. 2005, pp. 27–38.

[Gro00] The HDF5 Group. *Hierarchial Data Format Version 5*. 2000-2010. URL: http://www.hdfgroup.org/HDF5.

[Hav+04] Paul Havlak et al. "The Atlas genome assembly system." In: *Genome research* 14.4 (Apr. 2004), pp. 721–732. ISSN: 1088-9051. DOI: 10.1101/gr.2264004. URL: http://dx.doi.org/10.1101/gr.2264004.

[Hic+13] Glenn Hickey et al. "HAL: a hierarchical format for storing and analyzing multiple genome alignments." In: *Bioinformatics* 29.10 (2013), pp. 1341–1342. URL: http://dblp.uni-trier.de/db/journals/bioinformatics/bioinformatics29.html#HickeyPEZH13.

[Hil+04] L. W. Hillier et al. "Sequence and comparative analysis of the chicken genome provide unique perspectives on vertebrate evolution." In: *Nature* 432.7018 (2004), pp. 695–716. URL: http://eutils.ncbi.nlm.nih.gov/entrez/eutils/elink.fcgi?cmd=prlinks\&dbfrom=pubmed\&retmode=ref\&id=15592404.

[Kar+04] Donna Karolchik et al. "The UCSC table browser data retrieval tool". In: *Nucleic Acids Res* 32 (2004), pp. 493–496.

[Ken+02] W. James Kent et al. "The Human Genome Browser at UCSC". In: *Genome Research* 12.6 (2002), pp. 996–1006. DOI: 10.1101/gr.229102. URL: http://dx.doi.org/10.1101/gr.229102.

[Ken+03] W. James Kent et al. "Evolution's cauldron: Duplication, deletion, and rearrangement in the mouse and human genomes". In: *Proceedings of the National Academy of Sciences* 100.20 (Sept. 2003), pp. 11484–11489. ISSN: 1091-6490. DOI: 10.1073/pnas.1932072100. URL: http://dx.doi.org/10.1073/pnas.1932072100.

[Kos+08] Carolin Kosiol et al. "Patterns of Positive Selection in Six Mammalian Genomes". In: *PLoS Genet* 4.8 (Aug. 2008), e1000144. DOI: 10.1371/journal.pgen.1000144. URL: http://dx.doi.org/10.1371%2Fjournal.pgen.1000144.

[Li+09] Heng Li et al. "The Sequence Alignment/Map format and SAMtools". In: *Bioinformatics* 25.16 (2009), pp. 2078–2079. DOI: 10.1093/bioinformatics/btp352.

[Lin11] Mike Lin. *OCaml library for BGZF*. 2011. URL: https://github.com/mlin/ocaml-bgzf.

[Pat+11] Benedict Paten et al. "Cactus Graphs for Genome Comparisons." In: *Journal of Computational Biology* 18.3 (2011), pp. 469–481. URL: http://dblp.uni-trier.de/db/journals/jcb/jcb18.html#PatenDEJMSH11.

[Pet12] Michal Petrucha. "Data Structures for Whole-Genome Alignments". B.S. Thesis. Bratislava, Slovakia, 2012.

[Ree+10] Jane B. Reece et al. *Campbell Biology*. 9th. San Francisco: Pearson Benjamin Cummings, 2010. ISBN: 0-321-55823-5.

[RRR02] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. "Succinct indexable dictionaries with applications to encoding k-ary trees and multisets". In: *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*. SODA '02. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002, pp. 233–242. ISBN: 0-89871-513-X.

[TJ05] Heather Trumbower and Jennifer Jackson. "Key Features of the UCSC Genome Site." In: *CSB Workshops*. IEEE Computer Society, Dec. 2, 2005, pp. 33–34. ISBN: 0-7695-2442-7. URL: http://dblp.uni-trier.de/db/conf/csb/csbw2005.html#TrumbowerJ05.

[UCS] UCSC. *UCSC Genome Browser*. URL: http://genome.ucsc.edu/.

[Mou+02] Mouse Genome Sequencing Consortium et al. "Initial sequencing and comparative analysis of the mouse genome." In: *Nature* 420.6915 (Dec. 2002), pp. 520–562. ISSN: 0028-0836. DOI: 10.1038/nature01262. URL: http://dx.doi.org/10.1038/nature01262.

[The05] The Chimpanzee Sequencing and Analysis Consortium. "Initial sequence of the chimpanzee genome and comparison with the human genome". In: *Nature* 437.7055 (Sept. 2005), pp. 69–87. ISSN: 0028-0836. DOI: 10.1038/nature04072. URL: http://dx.doi.org/10.1038/nature04072.

Appendix A – Implementation

This thesis includes an attached DVD, containing the source code of our tool, which is also available at `https://github.com/kpetra/maptool`.

Appendix B – Credits for the used files

To the files used in the section 3.5 of this thesis are provided with the following acknowledgments:

- The chimpanzee genome, assembly Nov. 2003 (panTro1):
  - Funding: National Human Genome Research Institute
  - Sequence assembly: teams led by Eric Lander, Ph.D. at The Broad Institute and Richard K. Wilson, Ph.D. at The Genome Institute at WUSTL
  - Alignments: LaDeana Hillier, The Genome Institute at WUSTL and The Broad Institute
- The mouse genome, assembly Feb. 2003 (mm3):
  - Project Oversight: Arthur Holden and Francis Collins
  - Funding: GlaxoSmithKline; Merck Genome Research Institute; Affymetrix, Inc.; The Wellcome Trust; The National Institutes of Health
  - Sequencing: The Broad Institute, Cambridge, MA, USA; The Genome Institute at Washington University (WUSTL); The Wellcome Trust Sanger Institute
- The rat genome, assembly Jun. 2003 (rn3):
  - Project Coordination: Baylor College of Medicine Human Genome Sequencing Center (BCM-HGSC) and Richard Gibbs, Director
  - Funding: National Human Genome Research Institute (NIH) and National Heart, Lung and Blood Institute (NIH)
  - Sequencing: BCM-HGSC; Celera Genomics Group of Applera Corporation; Genome Therapeutics Corporation; The University of Utah
  - Assembly: BCM-HGSC ATLAS group - Rui Chen, James Durbin, Paul Havlak
- The chicken genome, Feb. 2004 (galGal2):

- Sequencing: The Genome Institute at Washington University (WUSTL), St. Louis, MO, USA
- Physical Map: The Genome Institute at WUSTL
- Genetic Mapping/Linkage Analysis: The Chicken Mapping Consortium – Wageningen Map based on Wageningen broilerxbroiler mapping population ("W" linkage maps; 460 F2 animals), coordinated by Martien Groenen, Wageningen University, Wageningen, The Netherlands; East Lansing Map based on the RJFxlayer cross ("E" linkage maps; 56 BC animals), coordinated by Hans Cheng, USDA-ARS, East Lansing, MI, USA; Compton map based on layerxlayer cross ("C" linkage maps; 52 BC animals), coordinated by Nat Bumstead, Institute for Animal Health; Consensus linkage map: Martien Groenen and Richard Crooijmans, Wageningen University; RH map: coordinated by Mireille Morisson and Alain Vignal, French National Institute for Agricultural Research (INRA), Toulouse; Linkage Mapping: White Leghorn X Red Junglefowl (800 F2 animals), coordinated by Susanne Kerje, Lina Jacobsson and Leif Andersson, Uppsala University, Uppsala, Sweden; Clone/Marker Pairs: coordinated by Jerry Dodgson, Michigan State University, East Lansing, MI, USA; Marker Name/Sequence Resolution: coordinated by Hans Cheng, USDA-ARS, East Lansing, MI, USA; Chicken FPC browser: ChickFPC, Martien Groenen and Richard Crooijmans (galGal3), Jan Aerts (galGal2), Wageningen University; Additional Mapping Data Contributions: Winston Bellott, David Page Lab, Whitehead Institute for Biomedical Research (WIBR-MIT), Cambridge, MA, USA
- cDNA sequences: National Institutes of Health and The University of Manchester BBSRC ChickEST Database
- RJF finished clones: NIH Intramural Sequencing Center (NISC) (AC091708.2, AC094012.2, AC138565.3); S. Kerje and L. Andersson, Uppsala University (AY636124.1); The Genome Institute at WUSTL (all others)
- Assembly, Assembly/Map Integration, Golden Path Creation: The Genome Institute at WUSTL