Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Data Structures for Whole-Genome Alignments

Bachelor's thesis

2012 **Michal Petrucha**

COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS

DATA STRUCTURES FOR WHOLE-GENOME ALIGNMENTS

Bachelor's thesis

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

Bratislava, 2012 **Michal Petrucha**

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Michal Petrucha
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: Dátové štruktúry pre celogenómové zarovnania

Cieľ: Cieľom práce je skúmať pamäťovo a časovo efektívne dátové štruktúry na ukladanie zarovnaní dlhých DNA sekvencií.

Vedúci: Mgr. Bronislava Brejová, PhD.
Katedra: FMFI.KI - Katedra informatiky

Dátum zadania: 14.10.2011

Dátum schválenia: 20.10.2011

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

....

študent vedúci práce

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Michal Petrucha
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: Data Structures for Whole-Genome Alignments

Aim: The goal of the thesis is to explore space and time efficient data structures appropriate for storing alignments of long DNA sequences.

Supervisor:	Mgr. Bronislava Brejová, PhD.
Department:	FMFI.KI - Department of Computer Science
Assigned:	14.10.2011
Approved:	20.10.2011

doc. RNDr. Daniel Olejár, PhD.
Guarantor of Study Programme

.... Student

.... Supervisor

Abstrakt

V tejto práci navrhujeme dátové štruktúry na uchovávanie zarovnaní genómov viacerých organizmov. Cieľom je efektívne mapovať pozície a regióny z jedného genómu na iný na základe ich evolučných vzťahov. V práci opisujeme niekoľko známych dátových štruktúr, ktoré umožňujú vykonávať operácie *rank* a *select* na binárnych reťazcoch a využívame tieto operácie na navrhnutie riešenia nášho problému. Toto riešenie sme taktiež naimplementovali a odmerali jeho efektivitu na skutočných údajoch, pričom sme porovnali viaceré varianty tejto implementácie. V porovnaní s existujúcim nástrojom na mapovanie liftOver naša implementácia mapuje pozície efektívnejšie, ale potrebuje viac času na inicializáciu.

Kľúčové slová: zarovnanie, celogenómové zarovnanie, mapovanie pozícií, mapovanie regiónov, rank, select

Abstract

In this thesis we design data structures for storing alignments of genomes from multiple species. The goal is to efficiently map positions and regions from one genome to another based on their evolutionary relationships. We provide an overview of several known data structures for the *rank* and *select* operations on binary strings and use these operations to design a solution to our problem. We have then implemented this solution and measured its performance on real data, comparing several variants of our data structure. Compared to the mapping tool LiftOver, our implementation maps positions more efficiently, but requires longer initialization times.

Keywords: alignment, whole-genome alignment, position mapping, region mapping, rank, select

Acknowledgements

I would like to thank my supervisor, Broňa Brejová, for her patience and valuable guidance.

Contents

Introduction **1**

1 Problem statement and related work **3**

1.1 Biological motivation 3

1.2 Formal definitions 5

1.2.1 Pairwise alignments 5

1.2.2 Multiple-sequence alignments 7

1.2.3 Whole-genome alignments 8

1.3 Problem statement 9

1.4 Input format 11

1.5 Related work 12

1.5.1 LiftOver 13

1.5.2 SAMtools 14

1.5.3 Nested Containment Lists 15

2 Proposed data structure **16**

2.1 Rank and select 16

2.1.1 Rank 17

2.1.2 Select 21

2.2 The use of rank and select for position and region mapping . . 22

2.2.1 Single-position mapping 22

2.2.2 Region mapping 24

2.3 Implementation design 25

2.3.1 BitSequence in libcds 27

2.3.2 Class WholeGenomeAlignment 28
2.3.3 Class AlignmentBlockStorage 28
2.3.4 Class BinSearchAlignmentBlockStorage 29
2.3.5 Class RankAlignmentBlockStorage 29
2.3.6 Class AlignmentBlock 30
2.3.7 Class SequenceDetails 30
2.3.8 Memory size estimate 31

3 Performance and testing **33**
3.1 Used test data 33
3.2 Ownership of BitSequence 34
3.3 Block search algorithm 37
3.4 Implementation of BitSequence 38
3.5 Region mapping and comparison to liftOver 39
3.6 Summary and possible improvements 42

Conclusion **44**

A Implementation **47**

List of Figures

1.1 Real-world alignment example 4
1.2 Alignment visualization example 6
1.3 Multiple possible alignments of a single pair of sequences . . . 6
1.4 A multiple alignment of three sequences 7
1.5 Score of a multiple alignment 7
1.6 Sample MAF file 12
1.7 Example of a chain 13

2.1 Jacobson's rank solution 18
2.2 Compression of bit sequences used by Raman et al. 20
2.3 Diagram of the implementation 26

3.1 Block lengths within the sample alignment 34
3.2 Comparison with liftOver on inputs of different sizes 42

List of Tables

3.1 Memory usage with fake BitSequence 35
3.2 Effects of shared BitSequences 36
3.3 Effect of different block search implementations 37
3.4 Efficiency of BitSequence implementations 38
3.5 Comparison to liftOver with positions 40
3.6 Region mapping performance 41

Introduction

Aligning sequences encoding genetic information of multiple species is a common approach to finding similarities between the species. Usually an alignment consists of a reference sequence to which one or more other sequences are aligned in blocks of variable length. There are many software solutions implementing algorithms that align sequences.

Once these alignments are computed, we are facing another task – we need to store them in a data structure allowing us to efficiently find regions in other sequences corresponding to a given region on the reference sequence. Since whole-genome alignments are large, we want to minimize the amount of data kept in memory.

A working solution needs to account for the fact that throughout evolution, the genomes of all species are subject to mutations, which means that some of the aligned sequences will contain extra substrings which appeared by insertions while others will have certain regions deleted.

In this thesis, we propose and implement an approach to this problem using succinct data structures supporting the *rank* and *select* operations on binary strings. We have chosen this approach, because these data structures are time and space efficient.

We start with a detailed description of the problem along with an overview of work already done in this area. Afterward we explain known data structures for *rank* and *select* and describe how they can be used to implement mapping of positions and regions from one genome to another. Finally, we compare the performance of various *rank* and *select* solutions in a practi-

cal implementation using real-world alignment data and also compare our solution to an existing mapping tool.

Chapter 1

Problem statement and related work

In this chapter we explain why alignments of sequences representing genomes are important from the biological point of view and formally define the problem we are aiming to solve. We finish this chapter with an overview of related work.

1.1 Biological motivation

DNA molecules encoding genetic information in living organisms are organized in two facing strands which are linear chains of nucleotides. Each nucleotide contains exactly one base – adenine, cytosine, guanine or thymine and faces a nucleotide from the opposite strand. Each pair of facing nucleotides contains complementary bases, i.e. a nucleotide with an adenine base faces a nucleotide containing thymine and a cytosine nucleotide faces a guanine nucleotide and vice versa. In addition, the structures of the two strands are reverse to each other, which means the beginning of one strand faces the end of the other one.

It is natural to represent genomes as sequences of letters `A`, `C`, `G` and `T`, corresponding to the four bases. These sequences are obtained from physical

```
agaattgtactgttctgtat------cccaccag
ggaggtg-actggtctgtcccctctgccccccag
```

Figure 1.1: An example of a real-world alignment. The first sequence is taken from the first chromosome of a human and the second one is taken from the 27-th chromosome of a dog.

DNA samples via sequencing. We usually store a sequence of bases contained in a single strand; as the strands are reverse and complementary to each other, the sequence of the other strand can be easily determined from the stored one.

There are several tasks where it is convenient to align DNA sequences of different species. Simply put, it means finding parts of the sequences where they are sufficiently similar to each other. A real-world example of an alignment is shown in figure 1.1. Once this is done, we can use this information to solve other tasks requiring us to relate these sequences to each other.

One example of such a task is reconstruction of phylogenetic trees. A phylogenetic tree shows the evolutionary relationships between species. We assume all of the given species have a common evolutionary ancestor and try to trace, based on their genetic information, the most probable course of evolution that resulted in the given outcome. Usually we expect the species with the most similar sequences to have a close common ancestor while species whose genetic information differs more have a distant common ancestor. There are various tree reconstruction algorithms that solve this problem based on an alignment of sequences from several species [Dur+98].

Another problem where alignments are applied is the search for genes, i.e. regions of DNA that hold information about proteins. Genes themselves are less likely to change than other parts of the DNA. That means the regions that vary only slightly even between distantly related species are good candidates for genes.

Lastly, sometimes we know a certain gene in a given organism, but we do

not know its function. We can try to look up the corresponding (orthologous) gene in a different species where the purpose of this ortholog might be better known and use it as a point of reference.

1.2 Formal definitions

Having an intuitive idea of what an alignment is good for, let us define it more formally. We start with simple pairwise alignments and then use them to define alignments of multiple sequences and, finally, whole-genome alignments.

1.2.1 Pairwise alignments

Pairwise alignments provide the basic building block for more complex alignments. They show the similarities between a pair of sequences.

Definition 1.1 *An alignment of sequences* $S_0 = a_0 \dots a_{n-1}$ *and* $S_1 = b_0 \dots b_{m-1}$ *is a function* $f : \mathbb{Z}_n \to \mathbb{Z}_m \cup \{\perp\}$ *satisfying the following condition:*

$$(\forall i, j \in \mathbb{Z}_n)(f(i) \neq \perp \wedge f(j) \neq \perp \wedge i < j) \Rightarrow (f(i) < f(j))$$

In other words, an alignment is a strictly increasing function that maps positions in the first sequence to positions in the second sequence, but we allow some of the positions in S_0 to have no corresponding position in S_1 by mapping them to $\perp$.

An alignment thus defined is best visualized using a table. Each line contains one of the aligned sequences and each column corresponds to a pair of nucleotides. If a position in the first sequence maps to $\perp$, we represent this in the table by putting a dash instead of a nucleotide in this column for the second sequence. Similarly, if a position in the second sequence has no position from the first one associated, we write a dash above the nucleotide. We refer to successive groups of dashes as *gaps*. See figure 1.2 for an example.

```
 012 3
--AGC-T
GCA-CGT
012 345
```

i	$f(i)$
0	2
1	$\perp$
2	3
3	5

Figure 1.2: Example of a local alignment of sequences $S_0 =$ AGCT and $S_1 =$ GCACGT. Left: visualization as a table, right: alignment function.

```
GCATATATG        GCATATATG--------        GCATATATG
CATATATG-        ---------CATATATG        -CATATATG
```

(a) f_1 (b) f_2 (c) f_3

Figure 1.3: Multiple possible alignments of a single pair of sequences

Given a pair of sequences there are many possible alignments between them. Figure 1.3 shows three of the many possible alignments for a pair of sequences. Clearly, the first two alignments give practically no information about the similarity of the two sequences whereas the third one makes it apparent. Therefore we use a scoring system to distinguish useful alignments from useless ones.

Usually a scoring function assigns a negative score to each gap or pair of mismatched nucleotides while a positive score is assigned to matching nucleotides. The score of an alignment is then the sum of scores for each pair of nucleotides and each gap. This way, an alignment with many equal nucleotides aligned to each other and only few gaps and mismatches will get a high score.

For example, let us assign a score of $+1$ to each pair of equal nucleotides and -1 to each gap or unequal pair. With this scoring function, alignments f_1, f_2 and f_3 from figure 1.3 would get scores -9, -17 and 7 respectively. As we could expect, f_3 has the highest score among the three presented alignments, since it is the only one with equal nucleotides aligned to each other.

```
S0: AAA-GATGTTAAATGA---GTT
S1: -AATGATGTTAAACGA-TTATT
S2: -AA-GGTGCTCAATGAGTTGTT
```

Figure 1.4: A multiple alignment of three sequences

Aligned sequence pair	Score
S_0: `AAA-GATGTTAAATGA--GTT` S_1: `-AATGATGTTAAACGATTATT`	9
S_0: `AAAGATGTTAAATGA---GTT` S_2: `-AAGGTGCTCAATGAGTTGTT`	7
S_1: `AATGATGTTAAACGA-TTATT` S_2: `AA-GGTGCTCAATGAGTTGTT`	7

Figure 1.5: The score of the three-sequence alignment from figure 1.4 computed as the sum of scores of the individual sequence pairs yields 23. Note that we omit columns that contain only gaps from the pairwise alignments.

1.2.2 Multiple-sequence alignments

The tabular representation shown for simple alignments can be naturally extended to alignments of multiple sequences, see figure 1.4. We can extend scoring as well, for example by computing the score of alignment between each individual pair of sequences in this multiple alignment and adding them to get the score of the whole multiple-sequence alignment. Refer to figure 1.5 for an illustration.

We provide an alternative definition in which one sequence has a special meaning.

Definition 1.2 *A multiple-sequence alignment of sequences $S_0, S_1, \ldots, S_n$ is a tuple of functions $f_1, \ldots, f_n$ where f_i is an alignment of sequences S_0 and S_i. Sequence S_0 is called the reference sequence and $S_1, \ldots, S_n$ are informants.*

Using this definition it is impossible to express all the information that can be contained within a table. For instance, the alignment table presented in

figure 1.4 contains information about the alignment of S_1 and S_2 in columns 17 through 19 which are filled with a gap in S_0. This cannot be specified by a multiple-sequence alignment following definition 1.2 as functions f_i only contain information about columns where the reference sequence does not have a gap. However, for the purposes of this work this definition is sufficient.

1.2.3 Whole-genome alignments

Pairwise and multiple alignments defined above are useful for representing relationships between relatively short related sequences. We will now turn our attention to whole-genome alignments. During evolution, portions of DNA may be reversed, moved or copied to different positions, and therefore it is impossible to display an alignment of whole genomes in the form of a simple table. Instead we need to split the genomes into reasonably sized regions better suited to form simple alignments. Alignments of these short regions make up whole-genome alignments.

The following set of definitions gives a formal overview of several common terms.

Definition 1.3

1. *A* chromosome *is a pair* (S, id) *where* S *is a sequence of characters* `A`*,* `T`*,* `C`*,* `G`*, each representing a nucleotide, and* id *is an identifier of the chromosome.*

2. *A* genome *is a non-empty set of chromosomes, where each of the chromosomes has a unique id.*

3. *A* region *is a tuple* (id, i, j, s)*, where* id *identifies a chromosome,* i, j *are indices into the sequence* S *of the chromosome specifying the range's end points, and* s *is either* $+$ *or* $-$ *and specifies the strand of the range. If* s *is* $+$*, the range describes a substring of* S *starting at* i *and ending at* j*. If* s *is* $-$*, the range describes the reverse of the complement of such a substring, where* `A` *is complementary to* `T` *and* `C` *is complementary to* `G`*.*

4. A position *is a tuple* (id, i, s) *where* id *is an identifier of a chromosome,* i *is an index into the sequence* S *of the chromosome and* s *specifies the strand.*

Pair $(\texttt{CTCCGAACGTGATTG}, chr1)$ is an example of a chromosome. Region $(chr1, 5, 10, +)$ represents its substring `GAACGT`, while region $(chr1, 5, 10, -)$ represents sequence `ACGTTC`. Similarly, position $(chr1, 2, +)$ points to a `T`, while position $(chr1, 2, -)$ points to its complement, which is `A`.

Definition 1.4 *A whole-genome alignment of genomes* $G_0, G_1, \ldots, G_n$ *is a set of multiple-sequence alignments, where:*

(i) each multiple-sequence alignment aligns a region of a chromosome from G_0 *to regions of a subset of genomes* $G_1, \ldots, G_n$

(ii) each region from G_0 *lies in the* $+$ *strand*

(iii) each nucleotide of each chromosome in G_0 *is contained in at most one multiple sequence alignment*

We call G_0 *the reference genome, and* $G_1, \ldots, G_n$ *are informant genomes.*

Put differently, a whole-genome alignment is organised in blocks where each block aligns parts of a subset of informant genomes to a region of the reference genome. No part of the reference genome appears in two different blocks. If within a block an informant does not have any alignment to the reference genome with a sufficiently high score, it may be omitted from the block altogether.

1.3 Problem statement

The goal of this thesis is to study the following problem. We want to preprocess and store a given whole-genome alignment, so that we can efficiently map positions and regions from the reference genome to each informant. This allows us to find genes orthologous to known genes in the reference genome

or regions of informants aligned to other regions of interest from the reference genome.

In particular, we will support the following two queries:

a) $map_position(P, g)$ returns for a given position $P = (id, i, s)$ in the reference genome the position in the genome G_g aligned to P.

b) $map_region(R, g)$ returns for a given region $R = (id, i, j, s)$ in the reference genome a region in G_g aligned to R.

For single-position queries, in case the reference position is aligned to a gap, we want either of the two informant positions adjacent to this gap. If there is no block containing position P, or the block containing P does not contain a range from genome G_g, the mapping does not exist.

For region queries we need a set of extra constraints for the result. If the whole region fits within a single alignment block, the mapping is clear and well defined. However, if the end points of the source region reside in different blocks, there is a valid mapping only if the following conditions are met:

i) in all blocks overlapping the query region, the region of the informant belongs to the same chromosome and the same strand

ii) the positions of regions of the informant sequence contained within these blocks, ordered by the positions of the respective reference regions, form a monotonous sequence, and this sequence is increasing for the forward strand and decreasing for the reverse strand of the informant

In other words, we require the mapped region of the informant to be colinear to the source region. This means that if we order the blocks involved by their reference position, regions of the informant have to follow each other. As a result, it is possible to take such blocks and merge them into a pairwise alignment between some region of G_0 and some region of G_g.

Note that successive blocks do not need to follow one another immediately in the informant genome G_g. The sequence between successive blocks will

correspond to a gap in the reference sequence in the merged alignment. Similarly, some blocks might not contain the informant G_g at all, which would result in a gap within the informant sequence in the merged alignment.

1.4 Input format

Whole-genome alignments can be stored for example using the Multiple Alignment Format [Maf]. MAF is a simple human-readable text format which consists of blocks separated by blank lines. This format was introduced with the Threaded Blockset Aligner and MULTIZ programs [Bla+04].

Lines starting with anything but an `'s'` contain metadata which is irrelevant for our purposes and can thus be ignored. Each line starting with an `'s'` describes a single sequence region present in the block and contains the following whitespace-separated fields:

- the name of the sequence, usually consisting of the name of the species and the name of chromosome
- the position of the region start in the whole sequence of the chromosome
- the length of the region
- the strand of DNA this region resides in
- the size of the whole sequence
- the actual nucleotides of this region, interleaved with dashes representing gaps

The strand is indicated as either a `+` or a `-` sign. In case of a `+` the sequence region given is from the forwad strand and in case of a `-` the region is taken from the complementary strand with coordinates relative to the beginning of the reverse strand.

The last field describing the sequence region itself is similar to the tabular representation of alignments described in the previous section. It is a string

```
track name=euArc visibility=pack
##maf version=1 scoring=tba.v8
# tba.v8 (((human chimp) baboon) (mouse rat))

a score=23262.0
s hg18.chr7    27578828 38 + 158545518 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
s panTro1.chr6 28741140 38 + 161576975 AAA-GGGAATGTTAACCAAATGA---ATTGTCTCTTACGGTG
s baboon         116834 38 +   4622798 AAA-GGGAATGTTAACCAAATGA---GTTGTCTCTTATGGTG
s mm4.chr6     53215344 38 + 151104725 -AATGGGAATGTTAAGCAAACGA---ATTGTCTCTCAGTGTG
s rn3.chr4     81344243 40 + 187371129 -AA-GGGGATGCTAAGCCAATGAGTTGTTGTCTCTCAATGTG

a score=5062.0
s hg18.chr7    27699739 6 + 158545518 TAAAGA
s panTro1.chr6 28862317 6 + 161576975 TAAAGA
s baboon         241163 6 +   4622798 TAAAGA
s mm4.chr6     53303881 6 + 151104725 TAAAGA
s rn3.chr4     81444246 6 + 187371129 taagga

a score=6636.0
s hg18.chr7    27707221 13 + 158545518 gcagctgaaaaca
s panTro1.chr6 28869787 13 + 161576975 gcagctgaaaaca
s baboon         249182 13 +   4622798 gcagctgaaaaca
s mm4.chr6     53310102 13 + 151104725 ACAGCTGAAAATA
```

Figure 1.6: Sample MAF file taken from the MAF format specification [Maf]

consisting of letters and dashes where each letter represents a nucleotide and dashes indicate gaps. Usually the fields in a MAF file are indented with spaces to make them vertically aligned within each block. This makes MAF files more human-readable.

An example of a short MAF file is presented in figure 1.6.

1.5 Related work

In this section we give a brief overview of known solutions to the mapping problem we have defined and other related problems.

```
TGAAG---CGTACCGTT          5 0 3
CGAAGTCCCCTAG--TT          5 2 0
                           2
```

Figure 1.7: Example of an alignment (left) with its corresponding chain (right). The first number in the first line indicates that the alignment starts with 5 gapless columns. The other two numbers in the line indicate that there are no nucleotides in the first sequence and 3 nucleotides in the second one between the two leading gapless blocks. The last line only contains one number as it corresponds to the last block.

1.5.1 LiftOver

LiftOver is a utility which is a part of the Kent source utilities, used in the UCSC genome browser [Ken+02]. This tool works with pairwise whole-genome alignments and maps regions from one genome in the alignment to the other.

The pairwise alignment needs to be processed into a set of chains beforehand. A chain represents how a pair of regions of two sequences is aligned to each other. It consists of a list of blocks, where each block corresponds to a gapless set of successive columns of the alignment. (Note the difference between blocks in liftOver chains and blocks in whole-genome alignments that correspond to whole chains.) In each chain block three numbers are stored: the length of the block and for each sequence the number of nucleotides that have to be skipped to reach the following block. Figure 1.7 shows an example of an alignment along with the chain corresponding to it.

Each block of the pairwise alignment is transformed into a chain. LiftOver then reads this set of chains into memory and uses it to find region mappings. The chains are stored in memory using a technique called binning.

Binning is a method of storing intervals by putting each into a bin which is determined by the end points of this interval. This technique divides the space of all possible intervals into multiple levels of bins. There is one top-level bin, which represents the whole range of possible coordinates. The

second level contains eight smaller non-overlapping bins that divide this range into equal parts. Each level is a subdivision of the previous one in a similar manner.

Each bin can only contain intervals fully contained in the range corresponding to this bin. Conversely, each interval belongs to the smallest bin capable of accommodating it. Within bins, intervals are stored as linked lists.

LiftOver stores the chains in bins based on their position in the reference genome. When mapping a region, it searches through all bins overlapping this region to find the chain containing it and then walks through the list of chain blocks to find the end points of the queried regions and their corresponding positions in the informant, thus finding the aligned informant region. Therefore the running time is in the worst case linear, but in practice each bin contains only a small number of chains and most chains are relatively short, which means that in the average case, the performance of this tool is sufficient.

1.5.2 SAMtools

The Sequence Alignment/Map format and a set of tools accompanying it, called SAMtools [Li+09], are designed to store alignments of many short sequences to a single long contiguous sequence. This is mainly useful for DNA sequencing where the equipment is only capable of extracting short reads from DNA molecules, which are then aligned to a reference sequence.

Due to the nature of these alignments, instead of mapping positions or regions from one sequence to another, SAMtools implement slightly different types of queries, such as retrieving the number of reads aligned to a given position. Because the amount of data retrieved from modern sequencing instruments is usually orders of magnitude higher than the sizes of MAF files we are working with, the files cannot be loaded into memory.

In addition to the SAM format, which is text-based, these tools also implement support for BAM, the Binary Alignment/Map format. This is a

binary compiled version of a SAM file, which significantly reduces the storage requirements and speeds up operations on alignments stored in this format.

1.5.3 Nested Containment Lists

In 2006, Alekseyenko and Lee [AL07] proposed a data structure to store intervals, which supports efficient search for intervals overlapping a queried range. Their experiments show that this technique is more efficient than binning used in LiftOver.

Intervals may correspond to alignment blocks or other genomic features. In contrast to our problem, authors also consider overlapping intervals (our blocks are non-overlapping in the reference genome).

The NCList data structure stores intervals in a list sorted by their starting positions. If an interval is contained in another one that precedes it, instead of being stored in the same list, it is stored in a separate NCList within the containing interval. Thanks to this invariant, the coordinates of interval ends in one level of the list are also increasing in the list sorted by interval starts.

To find all intervals overlapping a queried range, binary search is used on the outermost list of intervals to find the first interval with overlap. The list is then traversed sequentially starting from this interval until the first interval with no overlap with the query range is encountered. All intervals thus traversed are part of the resulting set because they intersect the query. From the property we mentioned follows that none of the intervals appearing further in the list overlap the range. As each interval may store a list of intervals contained in it, this algorithm is applied recursively on all lists nested in the traversed intervals.

In the worst case, the running time of this algorithm is linear in the number of stored intervals. Nevertheless, it performs well in cases where there are not many levels of the nested structure.

Chapter 2

Proposed data structure

Our solution to the defined mapping problem is based on the *rank* and *select* operations on binary strings. In this chapter we provide a definition of these operations and an overview of currently known solutions. Then we show how they can be used to solve the problem and describe the structure of our implementation.

2.1 Rank and select

Let $B[1..n]$ be a bit vector. Operation $rank(B, i)$ returns the number of elements equal to 1 in $B[1..i]$. Operation *select* is the inverse operation to $rank$. Given a number i, it returns the position of the i-th bit set to one.

For example, in bit string 0111001001, the value of $rank(4)$ is obtained as the sum of its first four elements, which means $rank(4) = 0 + 1 + 1 + 1 = 3$. In the same bit string, the value of $select(4)$ is 7.

In the rest of this section we will walk through a list of known algorithms and data structures that can be used to implement these two operations. The goal of these data structures is to provide fast query time while at the same time using as little memory as possible.

2.1.1 Rank

Naive prefix sums

A trivial solution for *rank* is to store an array of prefix sums for the whole sequence. Each element of this array will then store the rank of the corresponding bit. To store each element, we need $\log_2 n$ bits, therefore this solution requires $O(n \log_2 n)$ bits in total.

For very long bit sequences, this amount of memory is too high. To alleviate this problem, various succinct data structures have been invented. A data structure is considered succinct if the data stored in addition to the original bit string is asymptotically smaller than the bit string itself.

Jacobson

Jacobson [Jac88] proposes a solution for *rank* with constant time per query and $o(n)$ extra space (excluding the bit vector itself). This works by splitting the original vector into b blocks of length $b' = \left\lfloor \frac{\log_2 n}{2} \right\rfloor$, which are then grouped into s superblocks of length $s' = b' \cdot \lfloor \log_2 n \rfloor$.

For each of the s superblocks we precalculate the rank of the bit immediately preceding it. That is, if i is the zero-based index of the superblock, this value is

$$R_s(i) = rank(B, i \cdot s')$$

Then, for each block we calculate the rank of the preceding bit considering only the region of the original vector covered by its containing superblock. If i is the index of the containing superblock, S_i is the substring corresponding to this superblock and j is the index of the block, this value is

$$R_b(j) = rank(S_i, j \bmod s) = rank(B, j \cdot b') - rank(B, i \cdot s')$$

Lastly, for each bit sequence T of length b' and for each $k < b'$ we calculate the rank of the k-th bit in T:

$$R_p(T, k) = rank(T, k)$$

<table>
<tr><td>R_s</td><td colspan="12">0</td><td colspan="12">6</td><td colspan="12">12</td></tr>
<tr><td>R_b</td><td colspan="4">0</td><td colspan="4">1</td><td colspan="4">4</td><td colspan="4">0</td><td colspan="4">2</td><td colspan="4">3</td><td colspan="4">0</td><td colspan="4">1</td><td colspan="4">3</td></tr>
<tr><td>B</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
</table>

Figure 2.1: An example of Jacobson's structure for rank. In this example, $n = 36$, $b' = 4$ and $s' = 12$.

Figure 2.1 shows an example of a bit sequence with the precomputed values of R_s and R_b for each block and superblock.

Now, using these three precomputed tables we can express the rank of any bit of the original bit vector B as

$$rank(B, i) = R_s\left(\left\lfloor \frac{i}{s'} \right\rfloor\right) + R_b\left(\left\lfloor \frac{i}{b'} \right\rfloor\right) + R_p(T, i \bmod b')$$

where T is the substring of B of length b' starting at $b'\left\lfloor \frac{i}{b'} \right\rfloor + 1$. Clearly, this operation can be done in $O(1)$.

The table storing precalculated values of R_s requires $\log_2 n$ bits per superblock, which makes the following total.

$$O(s \cdot \log_2 n) = O\left(\frac{n \log_2 n}{\log_2^2 n}\right) = O\left(\frac{n}{\log_2 n}\right)$$

To store the precomputed values of R_b, we need $\log_2 s'$ bits per block (since s' is the length of substring S_i), which is

$$\begin{aligned} O(\log_2 s' \cdot b) &= O\left(\frac{n}{\log_2 n} \cdot \log_2(b' \log_2 n)\right) = O\left(\frac{n(\log_2 b' + \log_2 \log_2 n)}{\log_2 n}\right) \\ &= O\left(\frac{n \log_2 \log_2 n}{\log_2 n}\right) \end{aligned}$$

Finally, the table storing R_p has $2^{b'}$ rows, b' columns and each element requires $\log_2 b'$ bits, therefore its memory requirements are

$$\begin{aligned} O\left(2^{b'} \cdot b' \cdot \log_2 b'\right) &= O\left(2^{\frac{1}{2}\log_2 n} \cdot \log_2 n \cdot \log_2 \log_2 n\right) \\ &= O\left(\sqrt{n} \cdot \log_2 n \cdot \log_2 \log_2 n\right) \end{aligned}$$

As each of the three tables requires $o(n)$ space to be stored, the whole data structure requires $o(n)$ bits in addition to the original bit sequence.

González et al.

González et al. [Gon+05] improved the memory requirements in practical applications by fixing the size of blocks at a multiple of 32 bits, omitting the layer of superblocks and using population counting combined with bitwise operations instead of the R_p lookup table.

Population counting is an operation which, given a word w, returns the number of bits in w set to 1, which is the rank of its last bit. This operation is implemented in some CPUs; González et al. implemented it by splitting the word into bytes and using a lookup table of 256 elements.

The authors compare the performance and memory requirements for various combinations of block size and population counting implementations. They show that in practical applications, their implementation is both faster and has a smaller memory footprint compared to Jacobson's solution, although asymptotically it is worse.

Raman, Raman, Rao

Raman et al. [RRR02] described a solution that does not need to store the original bit sequence. Its memory requirements are

$$n \cdot H_0(B) + O\left(\frac{n \log_2 \log_2 n}{\log_2 n}\right)$$

bits where $H_0(B)$ is the zero-th order empirical entropy of sequence B defined as follows

$$H_0(B) = -\frac{n_0}{n} \log_2 \frac{n_0}{n} - \frac{n_1}{n} \log_2 \frac{n_1}{n}$$

where n_0 is the number of 0 bits and n_1 is the number of 1 bits in B. For each bit string B, the value of $H_0(B)$ is a real number from the interval $[0, 1]$. This value equals 1 for sequences where the number of 1 bits equals the number of 0 bita,s and it is small for sequences where most of the bits have the same value.

Instead of the blocks themselves, the data structure stores the number of 1s within each block,u and the lexicographical order of the block among all

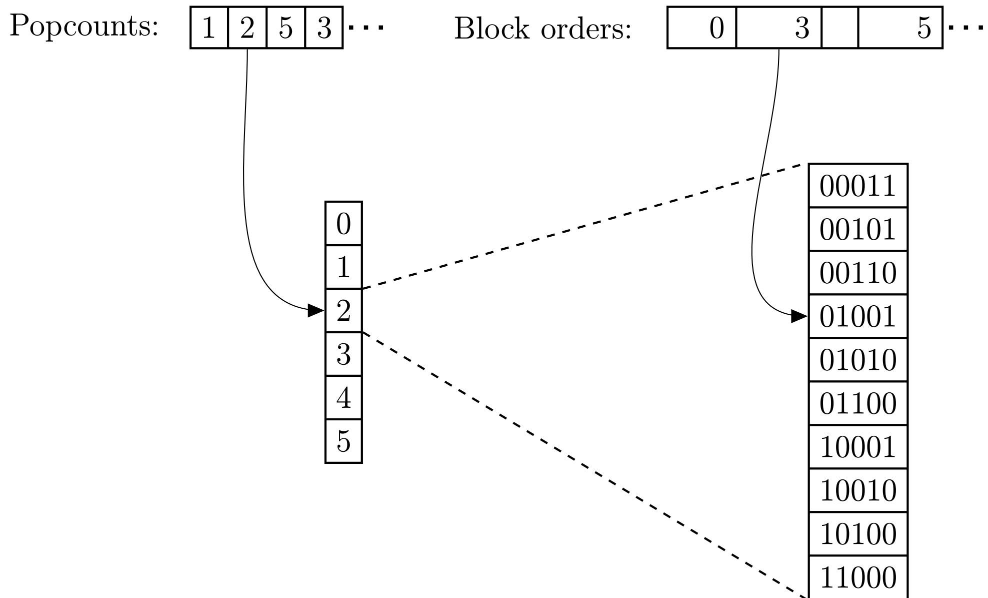

Figure 2.2: Demonstration of the compression used by Raman et al. For block 01001, the population count is 2 and it is the third block with this population count in lexicographical order. Since there are 10 blocks with population count 2, we need $\lceil \log_2 10 \rceil = 4$ bits to store the position in lexicographical order and additional $\lceil \log_2 6 \rceil = 3$ bits to store the population count. In this example the compressed representation is longer than simply storing the 5 bits of the block, but for longer block sizes it is generally more space-efficient.

possible blocks with the same number of 1s as shown in figure 2.2. This way, the data structure requires a smaller number of bits per block for blocks with a small number of zeroes or ones. The overall memory might be even smaller than the uncompressed length of B. At the same time it retains the ability to perform constant-time rank using modified lookup tables similar to those in Jacobson's data structure.

2.1.2 Select

Trivial solutions

Once rank is implemented, an easy way to perform select is to do a binary search using rank in $O(\log_2 n)$ time, i.e. for a given i find the smallest j such that $rank(B, j) \geq i$.

Simple as it is, this is not an optimal solution; select is possible in $O(1)$ time as well. However, the known data structures are in general more complicated than those for rank.

For bit sequences containing a small number of 1 bits, we can directly store an ordered list of positions at which these are located. This makes it possible to answer a *select* query in constant time. However, this is only efficient for really sparse bit sequences; in the average case, to store such a list we need $\Theta(n \log_2 n)$ bits, which makes this a suboptimal solution.

Clark

Clark's implementation of select [Cla98] works with $O\left(\frac{n}{\log_2 \log_2 n}\right)$ bits in addition to the bit vector. It splits the original bit string into variable width substrings, each containing $\log_2 n \log_2 \log_2 n$ ones. There are two cases of how these substrings are handled. For very sparse ones, the positions of all 1 bits are stored and for dense substrings, the same procedure is repeated. In the end, the second-level dense substrings are handled by a lookup table similar to table R_p used in Jacobson's rank implementation.

González et al.

González et al. implemented a solution based on binary search. It takes advantage of the two layers of structures used to solve *rank*. First, binary search is used to find the block containing the answer, then a sequential search within the block finds the answer. Their benchmarks show that for bit sequences of lengths up to 2^{26}, their implementation is faster than Clark's

constant-time algorithm, while also keeping the memory overhead significantly lower.

Raman, Raman, Rao

The structure proposed by Raman et al. is capable of constant-time select as well. Groups of successive blocks are created based on the positions of every $\left(\log_2 \frac{2n}{\log_2 n}\right)^2$-th bit. As in Clark's solution, the answers for sparse groups are stored directly. For each sparse group, an additive interval tree with a branching factor dependent on n is created. This tree stores in each node the sum of the range it spans and makes it possible to find the block within which the result is located. The variable branching factor allows for $O(1)$ bound depth, which is needed for *select* queries to run in constant time.

2.2 The use of rank and select for position and region mapping

In this section we describe how a whole-genome alignment can be preprocessed and stored in order to make it possible to use *rank* and *select* to implement the *map_position* and *map_region* operations. We provide a theoretical description and an estimate of time and memory complexity of the solution upon which our implementation is based. We also describe some possible improvements that are not part of the implementation.

2.2.1 Single-position mapping

The problem of mapping a single position in the reference genome to a given informant can be divided into two subproblems. First, we need to find the block containing the specified position, and then we have to find the position in the informant within this block.

Finding the block

The first subproblem can be solved in two ways. The first possibility is to sort the blocks by their position in the reference sequence. Then we can use binary search to find the right block.

The second approach uses *rank*. Again, we need to sort the list of blocks. Then, we create a bit sequence of the same length as the whole reference sequence. For each block, we mark the position at which the range from the reference sequence contained within this block begins with a 1 in the bit sequence. All other bits are set to 0. Calling $rank(p)$ on this sequence yields the number of blocks beginning before position p. This is also the index into a sorted array of blocks.

The time complexity of a search using binary search is $O(\log_2 m)$ where m is the number of blocks, the solution using *rank* requires $O(1)$ per query. Both require preprocessing before a search can be performed. The first approach needs to sort the blocks, which can be done in $O(m \log_2 m)$. The second one needs to sort the blocks as well, and in addition it needs to create a bit sequence of the same length as the reference sequence, which we will denote by n. The total time required to preprocess the list of block to use *rank* is $O(m \log_2 m + n)$.

Since binary search does not require any precomputed structures except for the list of blocks, its memory requirements are $O(1)$ in addition to the list. The solution based on *rank* requires an amount of memory dependent on the implementation used. However, since the bit sequence will in general contain very few one bits compared to zeroes, its empirical entropy will be low, which makes the Raman et al. implementation a good candidate.

Mapping a position within a block

To map positions within a block, we will preprocess the multiple-sequence alignment in the following way. For each line of the alignment, we create a bit sequence of the same length as the line. In this bit sequence, we store a 1 in each position corresponding to a nucleotide and a 0 in each position

corresponding to a gap. To prepare these bit sequences we need $O(b \cdot s)$ time, where b is the length of the block and s is the number of lines in it.

After the data structures for these bit sequences are prepared, we use the following algorithm to find the result of $map_position(P, g)$:

1. subtract the starting position of the region in the reference sequence overlapping this block from the given position P, let us mark it as p'
2. call $select(p')$ on the bit sequence corresponding to the reference line, the result c is the column in this block containing position p
3. call $rank(c)$ on the bit sequence corresponding to the requested informant G_g, denote the result as r
4. add the starting position of the informant region to r to obtain the resulting position

Since both *rank* and *select* can be performed in $O(1)$ time, the time complexity of this operation is constant.

2.2.2 Region mapping

Algorithms for single-position mapping can be subsequently used to solve region mapping. First we need to find blocks containing the end positions of the region. If the whole region falls within a single alignment block, the answer is obtained simply as the mapping of its end positions to the informant genome.

If the region spans multiple blocks, we have to verify that a valid answer exists. The trivial solution is to iterate through the list of blocks from the block containing the first position of the source region on, verifying the monotonicity of the aligned informant regions between each two adjacent blocks. The running time of this algorithm is linear in the number of blocks in this range. In the worst case this is $O(m)$ where m is the number of blocks in the whole-genome alignment. The memory required by this algorithm is $O(1)$ in addition to the structures needed to map positions.

This approach can be improved by precomputing groups of successive blocks such that regions inside one group can be mapped. This computation needs to be done for each informant separately. We can assign a unique ID to each of these groups and store the ID of group for each pair of block and informant. Then for each *map_region* query we just compare the group IDs for blocks containing the boundaries of the queried region, corresponding to the requested informant and test if they are equal, which is possible in constant time.

These groups can be prepared using the following algorithm. We initialize the group ID for the first block with 0 for each informant. Then we iterate over all alignment blocks. For each pair of successive blocks we test each informant whether it follows the region in the previous block. For each informant passing this test we set the group ID to the same value as in the previous block, otherwise we increase it by 1. This can be carried out in $O(m \cdot g)$ time and to store the precomputed IDs, we need $O(m \cdot g \cdot \log_2 m)$ bits.

To improve this even further, we can reduce the memory required to store the IDs using *rank*. For each informant we will create a bit sequence of length m. Each bit of this sequence corresponds to one alignment block and indicates whether the block breaks colinearity with the preceding one. In other words, instead of assigning an ID increased by one to such blocks, we set their corresponding bits in the bit sequence to 1. Then, given a block, we find its ID by calling *rank* on the respective bit sequence.

Clearly, the preprocessing time remains $O(m \cdot g)$ and we can still perform queries in $O(1)$ time, however, the memory requirements have dropped to $m \cdot g + o(m \cdot g)$ bits.

2.3 Implementation design

We have implemented the data structure described in the previous section. Our goal was to create flexible data structure which can be used to measure

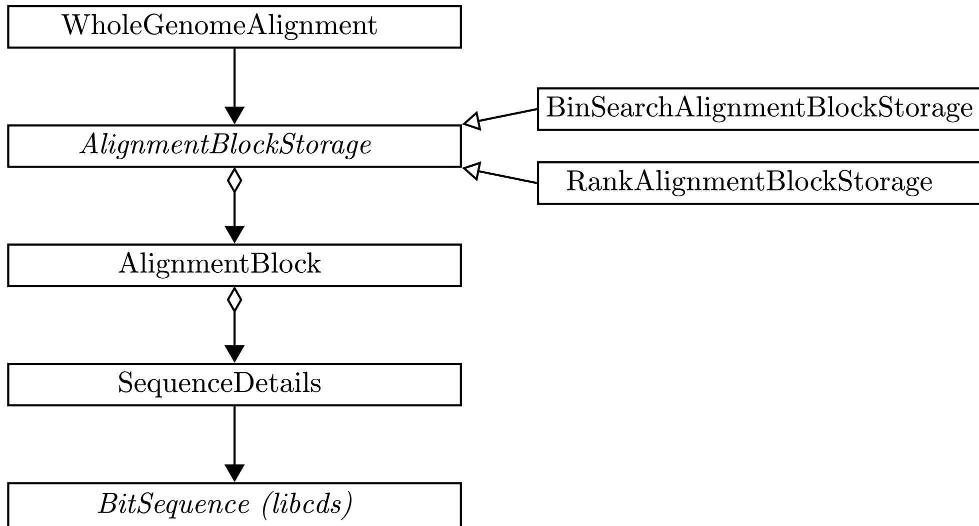

Figure 2.3: UML class diagram of the main classes in our implementation

the impact of various changes in the data structure on its performance. In this section we present the layout of our implementation. We decided to use libcds [Cla], an open-source library of compact data structures, which provides implementations of the *rank* and *select* data structures described in section 2.1. This allows us to measure the efficiency in terms of speed and memory of each *rank* and *select* implementation in this particular application.

The language of choice was C++, as this is the language in which libcds is written. In addition, it compiles compiles into fast machine code and provides a high degree of control over object size and lifetime, allowing for low-level optimizations while also supporting object-oriented design.

The basic structure of the classes is shown in figure 2.3. In addition the implementation contains a few other classes and functions, which simplify reading MAF files.

The most straightforward way of storing bit sequences is to create a separate instance of BitSequence (provided by libcds) for each row of each alignment block. However, all implementations have a certain constant memory overhead, which means keeping fewer instances with each storing a longer

bit string is more efficient than having many instances with short bit strings. Therefore we considered two additional alternatives: keeping a single bit sequence for each alignment block and making a single bit sequence global for the whole alignment.

These shared bit sequences then simply contain concatenations of the individual lines. For each line we store the index of the position within the global bit string where the line begins. When performing *rank*, we then need to subtract the rank of the stored index from the result. Similarly, when performing *select*, we need to add the rank of the stored index to the argument. This effectively means that we have to double the number of *rank* and *select* operations in exchange for possibly considerable savings in terms of memory consumption.

The following paragraphs give an overview of bit sequence implementations provided by libcds. A description of individual classes in our implementation follows. For each class we provide an estimate of the memory size of its instances. We assume the size of an integer to be 64 bits. We will denote the number of blocks as m, the length of the reference sequence as n, the number of lines in the whole-genome alignment as ℓ, and the number of distinct chromosomes present within the alignment as g.

2.3.1 BitSequence in libcds

Libcds provides an abstract class called BitSequence, which defines operations $rank1$, $select1$ and $access$, among others. This interface is implemented by the following subclasses.

- Class BitSequenceRG implements the *rank* and *select* operations as proposed by González et al. [Gon+05] It takes a parameter specifying the block size at runtime as the number of 32-bit words per block.

- Class BitSequenceRRR implements the compressed data structures described by Raman et al. [RRR02]

2.3.2 Class WholeGenomeAlignment

This class encompasses the whole alignment extracted from a MAF file and provides the high-level mapping operations defined in section 1.3. Currently only the linear-time region mapping algorithm is implemented.

Internally, each sequence name is assigned a unique 32-bit integer. This integer is called a sequence ID. The public interface of this class uses strings to identify sequences, which means it needs to store one map which translates strings into IDs and a vector of strings to perform the mapping in the other direction.

Each instance also contains a pointer to an instance of class Alignment-BlockStorage, which acts as a collection of alignment blocks.

The size of an instance of WholeGenomeAlignment in bytes is at least twice the sum of lengths of the names of distinct sequences plus $4g$ (as each ID is a 32-bit integer). In the approach with a single BitSequence instance global to the whole alignment, this class stores an additional pointer to the instance, which takes 8 bytes.

A more precise estimate of memory size is not possible, as the size of the map is dependent on the implementation of the standard template library provided by the compiler used. On the other hand, these structures are stored only once per whole-genome alignment, and in the average case the number of sequences will be orders of magnitude lower than the number of blocks or lines, which means that a higher constant factor in the size of this class will have a smaller impact on the overall memory footprint than constant factors in classes such as AlignmentBlock or SequenceDetails.

2.3.3 Class AlignmentBlockStorage

This abstract class defines the interface of a collection which stores the blocks of an alignment. It is implemented by classes BinSearchAlignmentBlockStorage and RankAlignmentBlockStorage. This class hierarchy implements the strategy design pattern.

The interface defined by this class provides the following operations:

- adding an AlignmentBlock instance
- finding an AlignmentBlock by position in the reference sequence
- iteration through the list of blocks in the order of their position in the reference sequence

In order to support iteration, this class and all of its subclasses are accompanied by a collection of iterator classes with the same interface as the standard template library iterators.

2.3.4 Class BinSearchAlignmentBlockStorage

This class implements the AlignmentBlockStorage by storing pointers to AlignmentBlock instances in a vector. This vector is sorted on the first read access and binary search is used to find the block containing the given position.

There are two data members in this class: a boolean to indicate whether the vector needs to be sorted and a vector of pointers. By shrinking the unused capacity of the vector at the same time as it is sorted, instances of this class achieve in-memory size of $8m + 1 + c$ bytes. The c constant is claimed by the vector instance and its value is 24 bytes on 64-bit Linux with GCC 4.4.

2.3.5 Class RankAlignmentBlockStorage

This class implements block search using the Raman et al. implementation of *rank*. Same as BinSearchAlignmentBlockStorage, it stores pointers to instances of AlignmentBlock in a vector which is sorted and shrunk on first access. In addition, an instance of class BitSequenceRRR from libcds is created in this stage.

Consequently, the members of this class are the same as in BinSearchAlignmentBlockStorage with an additional pointer to a BitSequenceRRR instance, the size of which depends on n and m.

2.3.6 Class AlignmentBlock

Instances of this class represent individual blocks in the alignment. Each block consists of several lines represented by instances of SequenceDetails. These instances are stored in a vector, which keeps the instances directly as opposed to pointers. This vector is sorted by the ID of the sequence whose region the particular SequenceDetails instance represents. Similar to BinSearchAlignmentBlockStorage, the vector is sorted and shrunk on the first read access, which means that an additional boolean flag is required to indicate whether this has been already done.

The memory required per instance is therefore $1 + c$ bytes, where c is the vector overhead, as in the discussion of BinSearchAlignmentBlockStorage. This quantity does not include the size of the stored SequenceDetails instances. An additional 8-byte pointer to a BitSequence is stored if BitSequence is shared for all instances of SequenceDetails within a block.

The most notable method in this class takes a position in the reference sequence, which has to lie in this block, and the ID of an informant for which a region needs to be present in the block, and returns the position in this informant sequence aligned to the queried reference position.

2.3.7 Class SequenceDetails

Each instance of this class represents a single line of the alignment. Its members include the starting position of the region represented by this line (integer), the size of the whole chromosome to be able to convert positions in various strands (integer), information about the strand (boolean), ID of the sequence to which this region belongs (32-bit integer) and a pointer to a BitSequence.

If each BitSequence is owned by a single SequenceDetails, a shared_ptr needs to be used. This smart pointer class counts references to objects managed by its instances and once this count reaches zero, it deletes the instance. It is required in order to ensure the BitSequence owned by each instance of SequenceDetails is properly freed. Note that we cannot do this in the destructor of SequenceDetails because instances are copied and deleted by the vector in AlignmentBlock, which means at some point the same instance of BitSequence might be shared by multiple instances of SequenceDetails.

This increases the number of bytes by an amount specific to the implementation of the C++ standard library, making the size of this class $8 \cdot 2 + 4 + 1 + d = 21 + d$ bytes, where d is the size of the shared_ptr implementation.

We can replace the shared_ptr with a regular pointer if the BitSequence is shared with other instances as in this case the BitSequence is owned by an instance of AlignmentBlock or WholeGenomeAlignment. Also, additional members are required, namely, the index in the BitSequence where this particular line begins (integer) and the length of the block (integer), which means the total size is $8 \cdot 5 + 4 + 1 = 45$ bytes.

This class also implements the methods in which *rank* and *select* are used to translate sequence positions into alignment block columns and vice versa. It is also responsible for recalculating positions in the reverse strand to their corresponding positions in the forward strand.

2.3.8 Memory size estimate

We may now proceed to estimate the amount of memory required to store the instances of our classes based on parameters m, ℓ and g specified earlier, not including the memory required by the instances of BitSequence.

For each whole-genome alignment, one instance of WholeGenomeAlignment and one AlignmentBlockStorage is required; m instances of AlignmentBlock and ℓ instances of SequenceDetails.

With the option of storing a separate instance of BitSequence per line,

this amounts to

$$(4g + 2s) + (8m + 1 + c) + m \cdot (1 + c) + \ell \cdot (21 + d)$$

bytes. s is the sum of lengths of the names of distinct sequences.

When each instance of BitSequence is owned by a block, the expected amount of memory in bytes is

$$(4g + 2s) + (8m + 1 + c) + m \cdot (9 + c) + \ell \cdot 45$$

Finally, in the case of a single BitSequence instance being global to the whole alignment, the expected number of bytes is

$$(4g + 2s + 8) + (8m + 1 + c) + m \cdot (1 + c) + \ell \cdot 45$$

Chapter 3

Performance and testing

In this chapter we measure the efficiency of our implementation using real-world data. 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 48 GB of RAM. We used the GCC 4.4 C++ compiler with -O3 optimizations.

3.1 Used test data

All tests described in this chapter were performed using the same alignment file. We chose the alignment of the human chromosome 2 to the following species: chimpanzee, orangutan, rhesus macaque, marmoset, mouse, rat, and dog. The length of this human chromosome is 243,199,373 nucleotides.

The alignment file was provided by Brian Raney from UCSC. [Ran10] It contains

- 489 different chromosomes
- 1,201,710 alignment blocks
- 7,765,103 sequence lines
- 1,367,643,053 nucleotides and 129,273,663 gaps

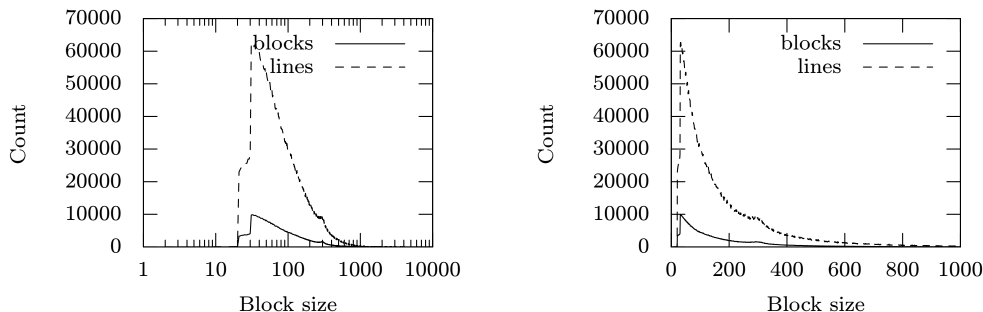

Figure 3.1: Distribution of block and line lengths in the test file. Left: full range in logarithmic scale. Right: lengths up to 1000 in linear scale.

- its total size is 1762.2 MB

Figure 3.1 displays the distribution of block lengths within the file. It shows that the vast majority of blocks are short (less than 500 columns).

The total number of bits required to be stored in instances of BitSequence representing nucleotides and gaps is equal to 1,496,916,716, which makes the lower bound for the memory required to represent uncompressed bit strings 187,114,590 bytes or 178.4 MB.

We generated a list of 2,000,000 positions in the reference chromosome selected at random from all positions that can be mapped to the genome of a rhesus macaque (an old world monkey). These positions are used in subsequent speed comparisons to ensure each query is valid and therefore performs all operations.

3.2 Ownership of BitSequence

In this section we assess the three options of storing BitSequence instances, described in section 2.3. We measure their impact on memory requirements and the speed of position mapping.

The effect on memory usage was measured in two ways. First, we used a fake implementation of BitSequence, which discards the sequence itself and only stores the required control structures amounting to 16 bytes. The

Implementation	Used memory	Expectation
one BitSequence per line	925 MB	312 MB
one BitSequence per block	478 MB	380 MB
one global BitSequence	506 MB	371 MB

Table 3.1: Memory usage with fake BitSequence. The expectation is based on estimates provided in 2.3.8 and values listed in section 3.1.

amounts of memory used by the three implementations after loading the whole MAF file are listed in table 3.1.

As we can see, the measured values differ significantly from the expectation. One likely reason is that the compiler pads the data types to better fit into memory. If we round the sizes of AlignmentBlock and SequenceDetails to the nearest multiples of 8, the estimate for the second case grows to 410 MB. Another reason might be a dynamically allocated structure used by vector, which would increase the size of AlignmentBlock. In addition, Linux does not provide a reliable and efficient method to determine the amount of memory used by a process, therefore our tool calculates the total size of virtual memory segments of the process not backed by a file, which might include some irrelevant segments.

The fact that using one global instance of BitSequence results in higher memory consumption than using a separate instance for each AlignmentBlock despite an opposite expectation might be caused by the algorithm used to read a MAF file in the former case. As the final size of the bit string cannot be known in advance, a fixed-length bit string is created first and its capacity is doubled each time it is filled. This means large amounts of memory are allocated and freed repeatedly while reading a MAF file. Usually, library functions to deallocate memory keep a certain amount of memory in reserve instead of releasing it immediately.

Finally, the disproportional difference in the first case is likely to be caused by the implementation of shared_ptr. This smart pointer class needs to store a shared object containing a counter, the managed pointer and a deallocator,

Implementation	Load time	Memory usage	Mapping time
one BitSequence per line	56 s	2480 MB	8.8 μs
one BitSequence per block	51 s	768 MB	11.0 μs
one global BitSequence	52 s	603 MB	15.8 μs

Table 3.2: Memory usage and position mapping times with the three approaches to BitSequence sharing. Load time is the time required to read the whole MAF file, mapping time is the average from 2,000,000 position mappings.

which means the size of its instances is likely multiple times the size of an instance of SequenceDetails itself.

We also performed a test with a real implementation of BitSequence in which we mapped 2,000,000 positions. The implementation used was BitSequenceRRR; BinSearchAlignmentBlockStorage was used to store blocks. The results are presented in table 3.2.

These results show that the choice of implementation does not have any practical effect on the time required to construct the data structure. This is true even in the third case, where an amount of memory proportional to the size of the MAF file needs to be reallocated and copied multiple times throughout the process.

The measurements of memory usage support the idea laid out in section 2.3, that joining short bit sequences into a larger one results in smaller memory overhead. Clearly, the first approach with millions of instances of BitSequence shorter than 1000 bits is impractical as in this case the memory consumed exceeds the size of the original MAF file by a factor of 1.5, while the other two implementations require less than a half of its size.

Regarding mapping times, we can see that the need to perform two additional *rank* operations per mapping with shared BitSequences reflects only lightly in mapping times. This is likely due to the effects of the processor cache, especially in the second case – since most alignment blocks are short, the whole BitSequence representing a block fits within the 8 MB cache, mak-

Implementation	Load time	Memory usage	Mapping time
binary search	51 s	603 MB	15.8 μs
rank	53 s	613 MB	12.6 μs

Table 3.3: Effect of the chosen implementation of block search on the time required to load a file, memory usage and time per query

ing the extra *rank* cheap. On the other hand, when the BitSequence is global, it no longer fits within the cache, thus slightly increasing the mapping time.

3.3 Block search algorithm

In this section we focus on the difference between BinSearchAlignmentBlockStorage and RankAlignmentBlockStorage. We mapped 2,000,000 positions with either of the two implementations while using a single global instance of BitSequenceRRR to perform mappings. The results are shown in table 3.3.

The extra memory required by the *rank*-based solution compared to binary search is just 10 MB, even for the second longest chromosome in the human genome. Compared to the rest of the used memory, this amount is practically negligible as it increases the memory usage by less than 2%.

We can also see that using *rank* reduces the time required per query by 3.1 μs, which amounts to about 20% of the overall time per query with the selected set of parameters.

The difference in time required to initialize either of the two implementations is in this case is negligible in contrast with the time needed to load an entire MAF file.

Overall, the gain of using *rank* is not big, but still it has some significance and the benefit outweighs the cost.

Implementation	Block size	Memory usage	Mapping time
BitSequenceRRR	—	613 MB	12.6 μs
BitSequenceRG	64 bits	786 MB	4.62 μs
BitSequenceRG	96 bits	756 MB	4.65 μs
BitSequenceRG	128 bits	741 MB	4.67 μs
BitSequenceRG	640 bits	706 MB	5.45 μs

Table 3.4: Comparison of memory usage and position query speed based on the implementation of BitSequence used. In case of BitSequenceRG, various block sizes were evaluated.

3.4 Implementation of BitSequence

In this section we study the effects of the chosen implementation of BitSequence. We ran the mapping test using BitSequenceRG and BitSequenceRRR to perform position mappings. With BitSequenceRG we tried block sizes of 64, 96, 128 and 640 bits. In each run a single global instance was used to perform mappings and RankAlignmentBlockStorage was used to store blocks. Table 3.4 lists the measurements obtained.

The results show a significant effect of the bit sequence compression used by BitSequenceRRR as this implementation beats even the most memory-efficient instance of BitSequenceRG by almost 90 MB. We can see the expected drop in memory usage with higher block sizes as well.

At the same time, using BitSequenceRG results in a major speed gain, compared to BitSequenceRRR – even in the slowest case, BitSequenceRG is more than twice as fast as BitSequenceRRR. This is due to the fact that BitSequenceRG is optimized to take advantage of the CPU cache while BitSequenceRRR requires more accesses to independent memory locations to perform each *rank* and *select* query, especially with long sequences, as noted in section 3.2. Also, the impact of higher block size on the execution speed is relatively low – negligible for block sizes between 64 and 128 bits; in the case of 640 bits the losses are somewhat higher but still less than 1 μs, which

is reasonably small.

To sum up, BitSequenceRG with block size of 640 bits seems like a reasonable choice. The higher amount of memory required by this implementation might be compensated for by further reducing the sizes of classes stored besides the bit sequence.

3.5 Region mapping and comparison to liftOver

In the last set of tests we will compare the performance of our solution to the liftOver utility from the Kent tools, described in section 1.5.1 on both position and region mapping.

In our first test, we map again 2,000,000 positions to the genome of a rhesus macaque. As liftOver only maps regions, to compare its performance when mapping positions, we used it to map regions of size 1 instead. Also, we are not able to separately measure the initialization stage and the actual mapping. Therefore we measure the execution time of the whole command for both liftOver and our implementation. We tried global instance of BitSequenceRG with block size of 640 bits and global instance of BitSequenceRRR; RankAlignmentBlockStorage was used to find blocks. Since liftOver uses pairwise alignment data, we did two runs with our implementation. In the first run we used the original MAF file containing the sequences of 8 species, and in the second run we used a modified version of this file containing only the pairwise alignment between the human chromosome and the genome of rhesus macaque.

We compared the outputs of the two tools and found that most of the mapped positions were equal. However, there were 23,853 positions which liftOver failed to map altogether. This is caused by the fact that the liftOver chains do not contain regions of the reference sequence at either end of a block aligned to a gap in the informant genome, making liftOver unable to map such positions. On the other hand, our implementation maps these positions to the boundaries of the informant regions in respective blocks.

Implementation	MAF file	Memory usage	Execution time
BitSequenceRG	full	706 MB	54.25 s
BitSequenceRRR	full	613 MB	81.66 s
BitSequenceRG	pairwise	264 MB	29.44 s
BitSequenceRRR	pairwise	230 MB	48.96 s
liftOver	—	791 MB	109.67 s

Table 3.5: Comparison of our implementation and liftOver by mapping the same set of positions

The output also differed for 1124 positions. In each case the difference is less than 20 nucleotides, most likely caused by slightly different handling of positions aligned to a gap.

According to the measurements shown in table 3.5, our implementation is both faster than liftOver and uses less memory. In addition, our implementation is capable of parsing a MAF file directly and it supports mapping into multiple informants at the same time while liftOver requires a separate chain for each informant genome. We remind that the size of the full MAF file is 1762 MB, while the size of the liftOver chain is only 135 MB. The size of the MAF file obtained by filtering out all irrelevant species is 541 MB. This makes the chain format much more space-efficient for pairwise alignments than MAF.

We can also see that by using a reduced MAF file, which in turn leads to shorter initialization times, the gain of our implementation over liftOver increases even further, providing mapping times an order of magnitude faster than liftOver.

To compare region mapping, we used 23,294 exons in the second human chromosome, retrieved from the RefSeq database [Pru+12] and mapped them to the genome of a rhesus macaque. Exons are parts of genes encoding protein sequences, and their mapping to different organisms is a common task. Again, we tried both BitSequenceRG with 640 bits per block and BitSequenceRRR, in both cases in conjunction with RankAlignmentBlockStorage.

Implementation	MAF file	Memory usage	Execution time
BitSequenceRG	full	706 MB	40.07 s
BitSequenceRRR	full	613 MB	56.29 s
BitSequenceRG	pairwise	264 MB	16.19 s
BitSequenceRRR	pairwise	230 MB	20.80 s
liftOver	—	405 MB	2.48 s

Table 3.6: Size and speed comparison of our implementation and liftOver when used to map 23,294 exons

Same as in the position test, we measured the performance of our implementation both with the complete MAF file and with the reduced pairwise version. Refer to table 3.6 for measurements.

The output regions differed in 114 cases, which is again caused by different handling of gaps.

As the results indicate, if we are using the full MAF file aligning the genomes of multiple species, the liftOver utility outperformed our implementation both in terms of memory usage and speed, achieving times an order of magnitude lower than our solution. However, by using a reduced version of the MAF file our implementation achieves considerably smaller memory usage than liftOver, but in terms of time it is still at a disadvantage.

While watching the memory usage of liftOver throughout its operation, we noticed that the amount of allocated memory starts at a certain number and gradually increases. This, along with the difference in memory usage in previous tests suggests that either liftOver leaks memory, or it caches successfully mapped regions. On the other hand, the running time of our implementation seems to be dominated by its expensive initialization phase, while actual mappings are performed faster than with liftOver.

To verify this hypothesis, we measured the execution times and memory usage of liftOver and our solution using BitSequenceRG and RankAlignmentBlockStorage on sets of positions of sizes up to 2,000,000 with a step of 200,000. We used the reduced version of the MAF file. Figure 3.2 displays a

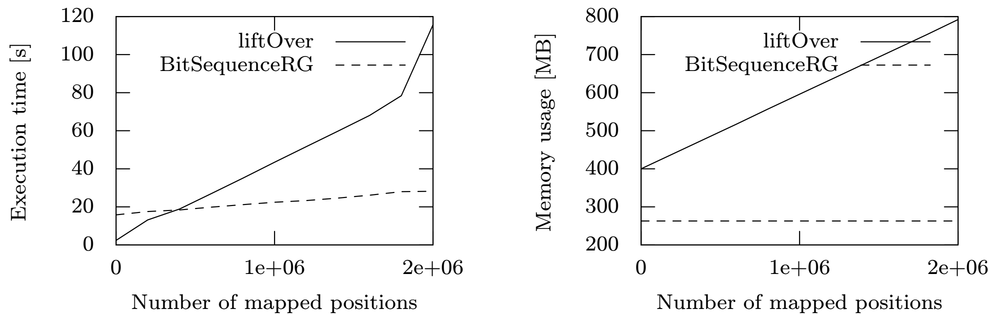

Figure 3.2: Comparison of execution times (left) and memory usage (right) of liftOver and our solution on data sets of different sizes

plot of the results.

These results support both claims of our hypothesis. We can clearly see that while the memory usage of our implementation remains constant across all input sizes, the memory consumed by liftOver has a linear tendency. According to the plot of execution times, our solution does indeed start at a higher execution time, but the rate at which it rises is much lower than the rate of liftOver.

3.6 Summary and possible improvements

As we have shown in this chapter, our solution is highly efficient in position mapping, once its data structures are initialized. However, to map regions into a single informant genome known in advance, liftOver provides a better solution, especially if the number of mapped regions is relatively low.

There are multiple areas in which our solution can be improved. Based on the first set of measurements, we can conclude that the amount of memory allocated by *rank* and *select* data structures is considerably lower than that used by our classes. To improve this situation it is possible to remove multiple data members from the SequenceDetails class and pass their values as parameters to methods which require them. For instance, we could completely remove the pointer to BitSequence or store the block length in

AlignmentBlock. With these two changes it is possible to reduce the memory usage in previous tests by 120 MB.

Another possibility for improvement is to implement faster algorithms to map regions. The ones described in section 2.2.2 might be a good starting point. Furthermore, other sets of restrictions on region mappings than those described in section 1.3 might be formulated, suitable for other applications. These might require completely different approaches to the problem.

Lastly, our implementation suffers from slow initialization of its data structures. This is especially evident in the last set of measurements. We could implement facilities to store the data structures in a file or database in a compact manner, making it possible to load them immediately.

Conclusion

The goal of this thesis was to create an efficient representation of whole-genome alignments that supports position and region mapping. For this purpose we studied currently known data structures for *rank* and *select* operations on binary strings, which have not yet been used in this context.

We designed a set of data structures and algorithms making use of these two operations, supporting the defined mapping operations in constant time. Then we proceeded to implement them in order to compare the known solutions for *rank* and *select* and decide which one is the most appropriate for the mapping task.

We also compared our implementation using the best two combinations of parameters, with an existing tool that solves the mapping problem. We came to the conclusion that while our solution takes more time to preprocess alignment data, it is faster at performing mappings, making it suitable for mapping of large batches of positions.

Overall, we have created a practical tool for mapping genomic positions between species, which is an important task in several bioinformatics problems. Our results also suggest several avenues for future improvement in this area.

Bibliography

[AL07] Alexander V. Alekseyenko and Christopher J. Lee. "Nested Containment List (NCList): a new algorithm for accelerating interval query of genome alignment and interval databases". In: *Bioinformatics* 23.11 (2007), pp. 1386–1393. DOI: `10.1093/bioinformatics/btl647`.

[Bla+04] M. Blanchette, W. J. Kent, C. Riemer, L. Elnitski, A. F. Smit, K. M. Roskin, R. Baertsch, K. Rosenbloom, H. Clawson, E. D. Green, D. Haussler, and W. Miller. "Aligning multiple genomic sequences with the threaded blockset aligner". In: *Genome Res* 14.4 (2004), pp. 708–715.

[Cla] Francisco Claude. *libcds: Compact Data Structures Library*. URL: `http://libcds.recoded.cl/`.

[Cla98] David Richard Clark. "Compact pat trees". PhD thesis. Waterloo, Ont., Canada, 1998.

[Dur+98] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, July 1998. ISBN: 0521629713.

[Gon+05] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. "Practical implementation of rank and select queries". In: *Poster Proceedings Volume of 4th Workshop on Ef-*

ficient and Experimental Algorithms. WEA '05. Greece, 2005, pp. 27–38.

[Jac88] Guy Joseph Jacobson. "Succinct static data structures". PhD thesis. Pittsburgh, PA, USA, 1988.

[Ken+02] W. James Kent, Charles W. Sugnet, Terrence S. Furey, Krishna M. Roskin, Tom H. Pringle, Alan M. Zahler, and David Haussler. "The Human Genome Browser at UCSC". In: *Genome Research* 12.6 (2002), pp. 996–1006. DOI: `10.1101/gr.229102`.

[Li+09] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, and 1000 Genome Project Data Processing Subgroup. "The Sequence Alignment/Map format and SAMtools". In: *Bioinformatics* 25.16 (2009), pp. 2078–2079. DOI: `10.1093/bioinformatics/btp352`.

[Maf] *MAF file format description.* University of California, Santa Cruz. URL: `http://genome.ucsc.edu/FAQ/FAQformat.html#format5`.

[Pru+12] Kim D. Pruitt, Tatiana Tatusova, Garth R. Brown, and Donna R. Maglott. "NCBI Reference Sequences (RefSeq): current status, new features and genome annotation policy". In: *Nucleic Acids Research* 40.D1 (2012), pp. D130–D135. DOI: `10.1093/nar/gkr1079`.

[Ran10] Brian Raney. *Personal communications.* University of California, Santa Cruz, 2010.

[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.

Appendix A

Implementation

This thesis includes an attached CD, containing the source code of our implementation, which is also available at `https://github.com/koniiiik/libmultialn`.