Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Algorithms for Segmentation of Biological Sequences

Master's thesis

2022

Bc. Dávid Simeunovič

Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Algorithms for Segmentation of Biological Sequences

Master's thesis

Study program:	Computer Science
Branch of Study:	2508 Computer Science
Department:	Department of Computer Science
Supervisor:	doc. Mgr. Bronislava Brejová, PhD.

Bratislava, 2022
Bc. Dávid Simeunovič

67646186

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Dávid Simeunovič
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Algorithms for Segmentation of Biological Sequences

Annotation: DNA and protein sequences can often be viewed as a mosaic of regions of diverse evolutionary origin. This complex structure is the result of evolutionary mechanisms which move or copy segments of DNA to new locations in the genome. Some portions of DNA can be also lost or gained from outside sources. Study of the underlying evolutionary mechanisms is simplified if we can first identify atomic segments which were likely not disrupted by any large-scale mutations in recent evolutionary history. The goal of the thesis is to develop algorithms for finding such atomic segments in input sequences. Most related approaches are not based on any clear formulation of the computational problem, and thus an important part of the thesis is also development of suitable formalizations.

Supervisor:	doc. Mgr. Bronislava Brejová, PhD.
Department:	FMFI.KI - Department of Computer Science
Head of department:	prof. RNDr. Martin Škoviera, PhD.
Assigned:	14.12.2016

Approved: 14.12.2016 prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

....

Student Supervisor

67646186

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Bc. Dávid Simeunovič
Študijný program:	informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor:	informatika
Typ záverečnej práce:	diplomová
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Algorithms for Segmentation of Biological Sequences
Algoritmy pre segmentáciu biologických sekvencií

Anotácia: Sekvencie DNA a proteínov sú často tvorené mozaikou oblastí s rôznymi evolučnými pôvodmi. Táto zložitá štruktúra je výsledkom evolučných mechanizmov, ktoré presúvajú alebo kopírujú oblasti DNA na nové miesta v genóme. Niektoré oblasti DNA môžu byť tiež zmazané alebo získané z externých zdrojov. Štúdium týchto evolučných mechanizmov sa zjednoduší, ak sa nám podarí identifikovať atomické oblasti, ktoré počas nedávnej evolučnej histórie neboli prerušené žiadnou mutáciou väčšieho rozsahu. Cieľom práce je vyvinúť algoritmy na hľadanie takýchto atomických oblastí vo vstupných sekvenciách. Väčšina súvisiacich prác nie je založená na jasnej formulácii výpočtového problému a preto dôležitým aspektom práce bude aj vyvinutie vhodného formalizmu.

Vedúci:	doc. Mgr. Bronislava Brejová, PhD.
Katedra:	FMFI.KI - Katedra informatiky
Vedúci katedry:	prof. RNDr. Martin Škoviera, PhD.
Dátum zadania:	14.12.2016

Dátum schválenia: 14.12.2016

prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

....
študent

....
vedúci práce

Acknowledgment: I would like to thank my supervisor doc. Mgr. Bronislava Brejová, PhD. for her invaluable guidance and support. Without her knowledge, patience and motivation, this thesis would not have come into existence.

Abstract

DNA and protein sequences can often be viewed as a mosaic of regions of diverse evolutionary origin. This complex structure is the result of evolutionary mechanisms which move or copy segments of DNA to new locations in the genome. Some portions of DNA can be also lost or gained from outside sources. Study of the underlying evolutionary mechanisms is simplified if we can first identify atomic segments which were likely not disrupted by any large-scale mutations in recent evolutionary history. The goal of the thesis is to develop algorithms for finding such atomic segments in input sequences. Most related approaches are not based on any clear formulation of the computational problem, and thus an important part of the thesis is also development of suitable formalization.

Keywords: DNA sequence, evolution history, local alignment, synteny blocks, atoms

Abstrakt

Sekvencie DNA a proteínov sú často tvorené mozaikou oblastí s rôznymi evolučnými pôvodmi. Táto zložitá štruktúra je výsledkom evolučných mechanizmov, ktoré presúvajú alebo kopírujú oblasti DNA na nové miesta v genóme. Niektoré oblasti DNA môžu byť tiež zmazané alebo získané z externých zdrojov. Štúdium týchto evolučných mechanizmov sa zjednoduší, ak sa nám podarí identifikovať atomické oblasti, ktoré počas nedávnej evolučnej histórie neboli prerušené žiadnou mutáciou väčšieho rozsahu. Cieľom práce je vyvinúť algoritmy na hľadanie takýchto atomických oblastí vo vstupných sekvenciách. Väčšina súvisiacich prác nie je založená na jasnej formulácií výpočtového problému a preto dôležitým aspektom práce bude aj vyvinutie vhodného formalizmu.

Kľúčové slová: DNA sekvencia, evolučná história, lokálne zarovnanie, syntenické bloky, atómy

Contents

List of Figures

Introduction

Evolution theory assumes that all life on Earth shares the last universal common ancestor *(LUCA)*, and genetic mutations affecting DNA play a crucial part in species evolution. Ideally, if we knew genomes of all species at every moment of evolution history, we would be able to fully reconstruct tree of life, all its nodes, connecting all species to $LUCA$, and for every present and past species reconstruct succession of evolutionary events that led to its genome. This is a nontrivial task due to the time scale of evolution, spanning billions of years from the $LUCA$ to a present day species. As a result of this long timescale, information about genomes of past species is lost and it is hard to reconstruct events that led from one observed sequence to another, or which connects them to a common ancestor.

In practice, studies reconstructing the tree of life, its parts (phylogenetic trees), or rearrangements between two sequences often rely on the maximum parsimony principle, which minimizes the total number of evolutionary events. Study of the underlying evolutionary mechanisms is simplified if we can first identify atomic segments which are parts of the sequences that were likely not disrupted by any large-scale mutations in recent evolutionary history. Further analysis may then, instead of the whole DNA sequence, work only with a sequence of atomic segments placed in classes. An example of such atomic segments frequently used are genes, as protein coding regions of DNA often show higher extent of conservation. In the case of using genes as atomic segments, apart from problems with finding and annotating genes, we also lose information contained in non coding regions, which may be valuable mainly in study of closely related sequences, which show a high level of similarity.

In this work, we study the approach of finding such conserved segments based on information about segment similarity obtained from local alignments. We call such conserved segments which we want to obtain *atoms*, and the set of atoms for single or multiple input sequences is called *atomization*. The main goal of the thesis is to introduce a formal definition of a computational problem that matches our intuition of what atoms are. For the newly created definition, we also want to develop an algorithmic way to obtain atomization. Our work is based on the previous work by Višňovská et al. [17], and our proposed definition is inspired by the definition presented there.

In Chapter 1 we present basic biological terms and processes necessary for this thesis. We introduce the problem of atomization, show previous works that are predecessors of our approach, as well as alternative approaches to atomization and closely related problems. Chapter 2 introduces our formal definition of atomization and presents algorithms used to construct it. Lastly, in Chapter 3, we compare atomizations created by our algorithm with atomizations by two different algorithms on simulated as well as real data.

Chapter 1

Background, related work and problems

This chapter offers introduction into basic biological terms needed for understanding of this thesis, shows progress in the field made by earlier research and introduces problems that are very similar to ours.

1.1 Basic biological terms and processes

The *genome* is the entire genetic material of a living organism. It's made of DNA and encodes information needed to create, maintain and reproduce the organism. The *Deoxyribonucleic acid (DNA)* is the carrier of genetic information in every living organism. A DNA molecule consists of two complementary anti-parallel strands, coiled around each other to form a double helix. Those strands are composed of nucleotides, which contain one of four bases: adenine, thymine, cytosine, guanine. DNA is usually written down as a sequence of bases on one strand; each base is abbreviated to its first letter {A, T, C, G}. DNA forms larger structures called chromosomes. A *gene* is a functional sequence of DNA; most genes encode proteins.

1.1.1 Evolution of DNA sequence

According to evolutionary theory, DNA sequences mutate over time. We recognize two kinds of mutations, based on the length of DNA they affect. Local changes, affecting one or few adjacent nucleotides, include such as single-nucleotide variants, short insertions or deletions. In contrast, large-scale mutations affect long DNA sequences, and include the following types of events.

- *Insertion* - DNA sequence is inserted into the original DNA sequence, resulting in a longer sequence

- *Deletion* - a contiguous part of the original DNA sequence is removed, shortening the DNA sequence
- *Transposition* - a part of a DNA sequence is moved into a new location within that sequence.
- *Inversion* - like transposition, but the inserted DNA strand is rotated
- *Duplication* - a part of the sequence, called the source sequence is copied, and a newly formed copy is then inserted somewhere in the former DNA sequence.
- *Speciation* - this event represents creation of a new species, thus creating an exact copy of the original genome, and in each copy the evolution proceeds independently.

Such large-scale mutations as mentioned above, due to their key role in species evolution, are often referred to as evolutionary events [3]. A sequence of evolutionary events leading from one common ancestral species to one or several present-day species is called *evolutionary history*, and speciations in such evolutionary history are often visualized with use of *phylogenetic tree*, where the common ancestor is the root of the tree, and branches represent evolutionary events leading from one species to another. Leaves are either present-day species or terminated branches of evolution ended due to extinction.

1.1.2 Additional terms

Homology is the relationship of two entities based on their common descent, without specification of evolutionary event [10]. In particular we are interested in homology of two genes or other DNA sequences. Such genes or sequences are called *homologs*. Possible source of homology is duplication, where source and its copy are homologs.

Orthology is a specific case of homology, created by speciation. Two homologs, which were created by speciation, are called *orthologs* [10].

De Bruijn graph is graph representation of sequence of symbols, based on k-mers, substrings of the sequence of length k. For instance, the sequence GATA, for $k = 2$, has three k-mers, GA, AT, TA. Vertices of the graph are k-mers from the sequence, and identical k-mers are represented by a single vertex. Vertices for subsequent k-mers from the sequence, obtained by single symbol shift, are connected by an edge. The sequence corresponds to a walk in the graph.

sequence T_1 :	G	A	T	-	-	C	A	G	C	A
sequence T_2 :	G	A	T	T	A	C	A	-	C	A

Figure 1.1: An alignment of sequences T_1 and T_2.

1.1.3 Local alignment

Sequence alignment is a computational problem, where we try to align two sequences by inserting gap symbols optimizing a selected scoring scheme. If nucleotide bases in an alignment column match, the score is increased. Gaps and non-matching bases decrease the score or leave it unchanged. Usually, high-scoring alignments represent significant similarity between sequences, and nucleotides in the same column are thought to share evolutionary origin. An example of an alignment can be seen in Figure 1.1. Local alignment tries to find alignment of segments of sequences that align nicely, instead of aligning whole sequences. In sequences that underwent evolutionary events, and share common ancestor, segments from one species, that were either transposed, inversed or duplicated, will align with their orthologs in second species, indicating existence of underlying evolutionary events. In a similar manner, deletion is indicated, when segment from one species do not have the counterpart in the second species to align with. Segments affected only by short evolutionary events also align, and are present at similar positions in both sequences. We can even apply local alignment to one sequence, to identify segments within the sequence with a desired level of similarity to each other (typically homologs resulting from a duplication). Local alignments can be found for example with the help of programs such as LASTZ [6] or LAST [9].

1.1.4 Problem of atomization

Evolution history typically happens at very long time scales. For instance, the Last Universal Common Ancestor (LUCA), the most recent organism that is ancestral to all life on earth, has lived 4.5 billion years ago [2]. Thus, the evolutionary history is unknown, and subject to study and reconstruction. In the past, studies compared fossils and present day organisms, based on observable traits, for example shape of bones. DNA sequencing brought another reliable source of information, earned from present-day species, as well as recently (in a scale of evolution) extinct species as Woolly Mammoth [12] and Neanderthal [15]. Often, studies trying to reconstruct some part of an evolutionary history are based on the parsimony principle. That means, the resulting evolutionary history uses the lowest number of evolutionary events to get from an unknown DNA sequence of the common ancestor to the DNA sequences of the present-day species, which can be obtained by DNA sequencing.

X: ABCDEFG
Y: ABCDEB'C'D'FG
Z: ABCDEB'D'FC'G'

Figure 1.2: An example of an atomization of sequence X which evolved into Z, with middle step Y. Every letter represents one atom. First BCD is copied, creating B'C'D'. If this was the end of the history, atoms would have been {A,BCD,BCD',FG}. Then, C' is transposed in between FG, breaking BCD' into three atoms B' C' D', and this breakage is back-propagated into BCD, leaving us with atoms {A,B,C,D,E,F,G,B',C',D'}. [17]

We can imagine DNA sequence as a simple string, and all evolutionary events as operations that cut that string into pieces, reordering, copying, removing or inserting those pieces, and then tying them back together. Imagine that we model the evolutionary history in this manner, but with a simple condition, that all cutting has to be done in advance. DNA sequence of the common ancestor would be split into shorter pieces called atoms (conserved segments), meaning pieces of sequence that were never split during the reconstructed evolution. Such an example of sequence evolution consisting of atoms is illustrated in Figure 1.2.

1.2 Related works

There is an ongoing research about sequence segmentation at our faculty [3], [17], primarily targeted at fine-scale analysis of events, which are relatively recent. Shared characteristic of both papers is the effort to use the whole DNA sequence to obtain atoms.

1.2.1 Iterative homology mapping

In early work by Brejová et al. [3] atomization is constructed with the help of local alignments. Local alignments are used to find segments with high similarity among multiple sequences. Such similar segments are presumably orthologs, and indicate the existence of evolutionary events and their boundaries. As shown in Section 1.1.4, atoms are pieces of sequence that were never split, and atoms affected by a single evolutionary event are affected in their entirety, thus boundaries of alignments are likely boundaries of atoms. Based on this assumption, the presented approach is to create an atom between every two adjacent boundaries of local alignments. This approach would work well, if we found every homology and aligned it perfectly. In the example seen in Figure 1.3, one homology was not found, leading to a missed boundary, and a wrong

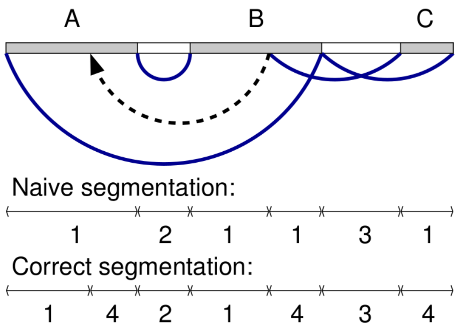


Figure 1.3: An alignment between A and C was not found, leading to a wrong segmentation. This situation can be fixed by proposed mapping of boundary of C through alignment between A and B marked as a dotted line [3].

resulting segmentation. To avoid this, all boundaries are mapped through overlapping alignments. Mapping a boundary through an alignment can be easily done, if it's located at an aligned nucleotide. If it's located at a nucleotide aligned with a gap, the boundary is moved to the nearest aligned nucleotide for the purpose of mapping. This has to be done iteratively, so that newly created boundaries can be mapped further. Another problem which arises with iterative mapping is that boundaries of alignments are spread around the real boundary. If those spread boundaries were mapped through overlapping alignments, it might result in a long chain of iteratively mapped boundaries that are of the same origin and very short atoms between them. An illustration of this problem is shown in Figure 1.4. To avoid it, no two boundaries can be closer than L. Nearby boundaries are clustered, and replaced with a new set of boundaries, which doesn't violate this rule. The new set is selected so that it minimizes squared distances between replaced boundaries and closest newly selected ones. So the algorithm in each iteration maps overlapped boundaries and clusters them into new ones. Boundaries aren't mapped multiple times, even if their position changed slightly.

1.2.2 Segmentation problem

In later work, which continues in attempt to obtain atomization from local alignments, Višňovská et al. [17] provides a more formal approach to the problem, formulating an optimization problem, proving its NP hardness and providing a heuristic algorithm leading to an approximate solution. Reimplementation of this algorithm in *C++*, supporting parallel computation, is presented by Rubert et al. [16] under name GEESE.

Formal definition

Beginning with sequence S, set of alignments α, and length parameter L, segmentation of sequence S is a set of atoms A, for which the following conditions are valid:

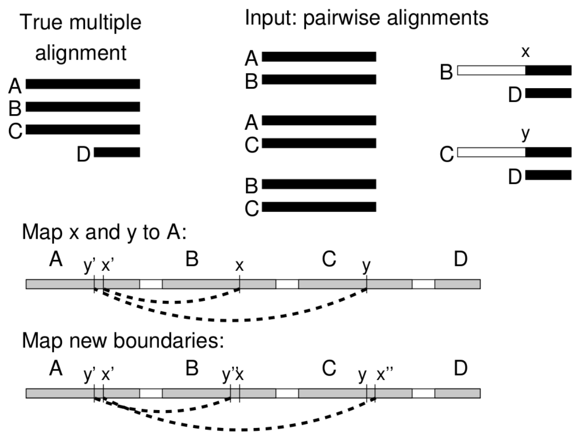


Figure 1.4: An example of imprecise boundary mapping. Boundaries x and y are in a slightly different spots, resulting in x' and y' not being mapped into a single boundary in A. If these boundaries are further mapped back as x'' and y'', creating new short atoms. [3]

C1. No two atoms from A overlap.

C2. The length of each atom is at least L.

C3. If the source or the destination of alignment $a \in \alpha$ overlaps some atom, it also covers that atom.

C4. If the source of alignment $a \in \alpha$ covers some atom E, then the region $a(E)$, obtained by mapping atom E through alignment a, overlaps with exactly one atom from A.

In other words, condition 3 ensures that boundaries of alignment are not inside any atom, because alignment boundaries might be seen as breaks in the sequence. Condition 4 covers the problem like the one in example Figure 1.3, where single atom A is aligned with two atoms in B, in order to satisfy this condition, we will have to split A into multiple atoms. In contrast to Iterative Homology Mapping, and in order to be able to satisfy conditions, Višňovská et al. approach does not have to cover the whole sequence with atoms. Regions not covered by atoms are called *waste regions.* Such waste segments are for instance spanning between alignment boundaries that are too close for placement of the atom, as it would not achieve length L. They also offer another way to satisfy condition 4 when A is aligned to multiple atoms. Part of A might be then turned into waste, so that the rest of A will align only with a single atom.

Cost function

Višňovská et al. also introduces a cost function, preferring the lowest number of nucleotides in the waste regions (maximal coverage by atoms), and with a lower priority, it's also preferred to have a small number of atoms.

The problem is proved to be NP-hard. Its NP-hardness is proved by a reduction from the one-in-three 3SAT problem.

Algorithm

Because the problem is NP-hard, the paper describes a practical heuristic algorithm. Starting with a set of evolutionary related sequences or a single sequence, LASTZ is used to obtain local alignments, alignment preprocessing is used to discard short and weak alignments. Initial waste regions are created at beginning and end of sequence, and at location of every alignment boundary. If any two waste regions are closer than L, they are merged into a single waste region, because it is impossible to create an atom between them, due to condition 2. This results in a set of proto-atoms that satisfies conditions 1, 2 and 3.

If condition 4 isn't satisfied, some proto-atoms map to a region overlapping multiple proto-atoms or a region completely covered by waste. Such a proto-atom is either split into multiple atoms, or shortened by expanding waste region on one of its ends. There may be multiple possible places, at which a proto-atom might be split. To do so effectively, splitting requirements from all alignments covering the currently studied proto-atom are collected, and an optimal (cost effective) solution is found using an algorithm called IMP (Inverse Mapping to Proto-atom).

1.2.3 Atom classification

After finding a set of atoms, those atoms can be classified into classes, where all atoms from one class are considered to be homologous, and atoms from different classes share minimal resemblance. Višňovská et al. present a solution to the atom classification problem based on graph structure. Atom classification is solved as a graph clustering problem, where vertices represent atoms, and alignments between atoms form the edges of the graph. Given the cost of adding and deleting edges, the goal of graph clustering is to turn the graph into cliques (fully connected components) at minimal cost. Adding and deleting edges corresponds to missed alignments between homologs, and incorrect alignments between unrelated atoms. This problem can be solved independently for each component of the graph, because atoms from different components will be in different clusters. Also components that form a clique don't need to be processed, as they are already a cluster. For the rest of the components, edges are added and deleted to turn the component into clusters at lowest price based on cost of edge deletion and addition.

1.3 Alternative approach

1.3.1 Genes as atoms

As we have mentioned at the beginning of Section 1.2, atomization created from local alignments on the whole genome is targeted at fine-scale analysis of events, which are relatively recent. While homologs and orthologs are same when created, accumulation of mutations in distantly related sequences leads to significant drop in their similarity. This may render our approach to create atomization from local alignments unfeasible, and similarity between organisms have to be found otherwise. Coding parts of DNA (genes) often exhibit a significant extent of conservation, due to mechanisms protecting them from harmful mutations. This makes genes ideal candidates for atoms in study of distantly related species.

Cons of using genes as atoms Brejová et al. [3] show various cases, in which protein coding genes selected as atoms are problematic. Firstly, we need to find and annotate genes and to find homologies or orthologies among them. This required preprocessing is a non-trivial task, and may introduce errors. Brejová et al. proceed to show problem with gene having multiple copies mutated into inactive form (pseudo genes); chimeric genes assembled from two parts with independent ancestry; UGT1A cluster containing 13 copies of the first exon (segment of gene translated into protein), created by duplication.

1.3.2 Sibelia and SibeliaZ

The tools named Sibelia [14] and SibeliaZ [13] allow finding synteny blocks (highly conserved non-overlapping segments) in multiple closely related bacterial genomes. Genomes are concatenated into a single sequence S_0, and special delimiter symbols are inserted. Instead of finding local alignments, which can be time consuming with a rising number of genomes, Sibelia and SibeliaZ use de Bruijn graphs. Iteratively, multiple de Bruijn graphs are created for different values of k. In the first iteration, a small k is selected, the graph is created, and non-branching paths are merged into a single node. Sequence S_0 is modified by *sequence modification algorithm*, to remove short bulges (branching of similar walks) caused by single point mutations or indels, sequence S_1 is obtained, and the next de Bruijn graph for larger k may be created. Iteration continues, until it reaches k large enough to reveal large-scale synteny blocks.

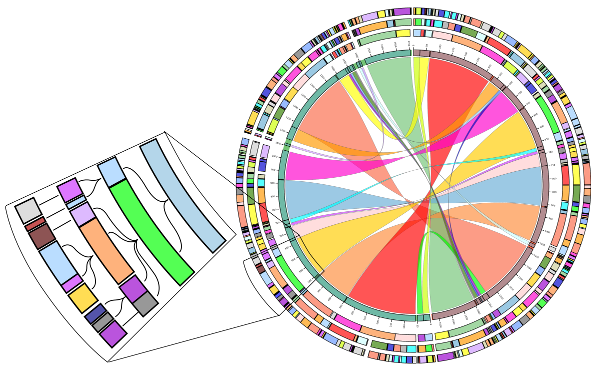


Figure 1.5: An example of Sibelia synteny finder for two sequences of *Helicobacter pylori*: F32 and Gambia94/24. Each layer represents atoms created in one stage of the iteration, outermost layer being the first stage. As can be seen in the zoomed panel, blocks are iteratively merged to create new larger blocks.[14]

This process reveals synteny blocks of different granularity, as can be seen in Figure 1.5.

1.4 Usage of atoms

One of the frequent problems in reconstruction of evolutionary history is to reconstruct evolutionary events leading from one genome to another minimizing the number of operations (maximum parsimony principle). The extension of this problem is reconstruction of a phylogenetic tree, where either shape of the tree is known, and minimal number of operations is computed to get from genome in one vertex to another, or shape of the tree is being reconstructed as well. Often, instead of using original sequence in such studies, blocks or markers, holding information about large scale events, are derived from original sequence and used to replace the original sequence. One example of such blocks are genes, for instance used by Lajoie et al. [11] to reconstruct a phylogenetic tree of tandemly arrayed gene clusters (copies of gene adjacent to original). Using atoms instead of genes brings additional valuable information into process, as we are using data with finer granularity. Example of usage of such fine grained data for reconstruction is found in paper by Drillon et al. [5], where they present *PhyChro*, tool for phylogenetic reconstruction working with synteny blocks (highly conserved segments), and show its robustness to different definitions of synteny blocks. Note that we have abstained from referring to our atoms as synteny blocks, due to different definitions of synteny blocks, of which some of them align with our definition of atoms and some do not, but our atoms may be viewed as a kind of synteny blocks. Accordingly, in the context of PhyChro, our atoms represent possible candidates for synteny blocks used in reconstruction.

1.5 Related problems

The problem of sequence segmentation, is related to the problem of multiple sequence alignment, where the goal is to arrange k sequences into a matrix with k rows, so that each column contains homologous symbols (or gaps). However, duplications and rearrangements make it impossible to represent an alignment of longer sequences in this linear form, so the programs often split the genomes into smaller blocks and align each block separately [1] [4]. These blocks are similar to our atoms, but the goal of multiple alignment programs is not the creation of optimal blocks, but rather the best alignment.

Also a very similar problem can be found in the field of protein decomposition, where the goal is to split proteins into modular domains. These domains resemble atoms, because new proteins often evolve by rearrangement or modification of existing domains, not from scratch. Heger et al. [7] present a method for protein decomposition and family classification based on sequence, thus strongly resembling the problem of atomization. Family classification is viewed as a graph partitioning problem, with weighted edges due to different degrees of similarity between proteins.

Chapter 2

Atomization

In this chapter, we present our approach to obtain atomization from local alignments. We first present an updated definition of atomization and scoring scheme, and design new algorithms for finding a pseudo-atomization and a full atomization. In comparison to previous work by Višňovská et al. [17], we account for inaccuracies of alignment ends in rule 3, and tighten requirements in rule 4.

2.1 Basic notation

Sequence indexing In the sequence we want to atomize, we are indexing the positions between the nucleotide bases. The sequence of length n has n nucleotide bases, indexed from 0 to $n-1$ and $n+1$ indexed positions from 0 to n. The k-th nucleotide base is bounded by indexed position k and $k+1$.

Segment An atomization of sequence S is a set of segments which satisfies some specific conditions, which we describe in section 2.2.1. A segment ranging from index j to index i will be denoted as $[j \dots i]$, and contains nucleotide bases from j-th to $(i-1)$-th. We reserve the name *sequence* for the input sequences of nucleotide bases, and *segment* for sub-sequences of given inputs, created by some of our algorithms. Individual segments in an atomization are also called atoms.

Alignment Local alignment a describes a bidirectional relationship of two segments E_1 and E_2. The first segment is referred to as the *source* of the alignment, and the second is the *target*. For every alignment a, we can easily create an alignment a' by switching the source with the target. This allows us to consider an alignment as a uni-directional relationship where the source is aligned to the target. This also eases up definition of atomization, and description of algorithms.

Matched pair is composed of two nucleotide bases, one from the source and one from the target of alignment, which are aligned - matched to each other. A base is either matched to a single base in alignment or not matched to any. The number of matched bases in the alignment source and target is equal. Ordering of matched bases from the alignment source is the same as ordering of their matched counterparts in the alignment target. Inverse alignments and their effect on atomization are discussed separately in subsection 2.4.5 and ignored until that point.

Alignment boundaries are borders of alignments source and target. In line with sequence indexing, alignment boundaries for the segment E_1 denoted as $[p \dots q]$ are placed at indexed positions p and q, before first base and after last base. Boundaries for the target of the alignment E_2 are obtained in the same way. We will keep a set of all indexes of alignment boundaries, together with their count for a given index, as multiple boundaries might be placed at a single position.

Segment projection In our definition of atomization as well as in the algorithms, we will need to take a segment that is inside the alignment source and search for its counterpart in the alignment target. For segment E and alignment a we construct the projection $a(E)$ by finding the first base q_f and the last base q_l from E, which are matched to base in alignment target. For those two bases, we find t_f and t_l, bases they are matched to, in the target sequence of alignment a. Segment $[t_f \dots t_l + 1]$ starting with t_f and ending with t_l is then the projection of segment E through alignment a denoted as $a(E)$.

Note that some segments cannot be projected very well. We obtain an empty projection if none of the bases from segment E is part of any matched pair in alignment a. Similarly, we obtain a single-base projection if only one base of E belongs to a matched pair. An alignment with overlapping source and target might lead to a projection $a(E)$ overlapping with segment E.

Segment match with tolerance Given segments E_1, E_2 and constant x, we say that segment E_1 matches segment E_2 with tolerance x, if the start of E_1 is within distance x from the start of E_2 and the end of E_1 is analogously within distance x from the end of E_2.

2.2 Atoms

We continue in the approach used by Višňovská et al. to use local alignments as a source of information for finding an atomization. We propose a changed definition of atomization which aims to tackle the problem of inexact alignment ends. In preprocessing, we filter out alignments with a low score and very short ones, which would cause problems in finding real atomization. Even in correct alignments, alignment ends are prone to error, as they might be shifted slightly from the position where the true atom would end. To tackle this problem, we introduce the terms core and edges of an atom (or any other segment of the sequence). *Edges* of an atom are the regions of a given fixed length d_1, spanning the first and last d_1 bases inside the atom. The center of an atom, between the edges, is then called its *core*. To differentiate between two edges of a single atom, the atom start edge spans the first d_1 bases and the atom end edge spans the last d_1 bases.

2.2.1 Formal definition of atomization

Consider sequences $S_1, S_2, \ldots, S_n$, set of alignments α, length parameter L, constants d_1 for length of segment edges and d_2 for segment matching. Atomization of sequences $S_1, S_2, \ldots, S_n$ is a set of segments A, for which the following conditions hold:

C1. No two segments from A overlap.

C2. The length of each segment from A is at least L.

C3. If the source of some alignment $a \in \alpha$ overlaps the core of some segment from A, it also completely covers the core of that segment.

C4. If the source of alignment $a \in \alpha$ covers the core of some segment $E \in A$, then the projection $a(E)$ matches exactly one segment from A with tolerance d_2.

Segments from atomization A are called *atoms*. If a set of segments A satisfies the first three conditions, but not necessarily C4, it is called a *pseudoatomization*, and its segments are *pseudoatoms*. Both d_1 and d_2 are user-set parameters, and we can assume $d_1 \ll L$ and $d_2 \ll L$.

Rules $C1$ and $C2$ are the same as in the definition by Višňovská et al. (see Section 1.2.2). Rule $C3$ is altered to allow alignment boundaries inside atoms, but only in the atom *edges*, not in their core, with the goal of countering the influence of inexact alignment boundaries. Finally, rule $C4$ reflects the fact that if there is some atom E inside the alignment source, a similar atom E' is expected to be present in the alignment target. In comparison to the previous definition of rule $C4$ by Višňovská et al., which accepted overlap between E and E' of any length, even a single base, our definition requires that these atoms match with tolerance d_2, which for small values of d_2 means that most of their bases overlap. Also, if for the atom projection $e(A)$ there is one matching atom B, short overlaps of $e(A)$ with its successor and predecessor are tolerated by our new definition. Those overlaps cannot be longer than d_2, as the atom projection can not extend further past the matching atom. We allow those overlaps because they could have been caused by incorrect alignment boundaries, which shortened the matching atom B.

Atomization waste segments For nucleotide bases present in a gap between two subsequent atoms, not covered by the atomization - that is not inside any atom, we introduce the term *waste segment*. Other than between two atoms, bases not in atom might be preceding the first atom or succeeding last atom, those are also covered by waste segments. Special case of waste segment of length 0 is allowed. That is a waste segment containing 0 bases, placed at an indexed position between two bases. Such a waste segment exists for the purpose of scoring and algorithm description. Each pair of the subsequent atoms is delimited by some waste segment, a single waste segment covers a continuous stretch of bases, and no two waste segments are touching. Waste segment of length 0 is placed as a delimiter between two consecutive atoms. Waste segments of length 0 or longer are also placed at the sequence beginning and end, thus the number of the atoms in each sequence is equal to the number of waste segments minus one.

Scoring scheme

Among many possible atomizations of a given sequence we want to select the one that is optimal according to some scoring scheme. Our main goal is to maximize the coverage by atoms, so first we penalize atomization for each nucleotide base that ends up in a waste region with a penalty p_1. Secondly we want the minimal number of atoms to avoid unnecessary fragmentation of atoms into smaller parts. As the number of atoms is the same as the number of waste segments minus one, we will add a penalty p_2 for each new waste segment started. The tertiary penalty p_3 will penalize atoms for alignment boundaries inside them, in particular for each boundary inside an atom edge, as alignment boundaries cannot be inside an atom core. We will add this penalty as squared distance of an alignment boundary from the closest atom border multiplied by p_3.

The importance of primary, secondary and tertiary penalty is set so that we consider the penalty of a higher order in comparison of two atomizations only if previous penalties are equal. This can be achieved by setting an appropriate weight for each penalty and adding them up to a single value, with weight of primary penalty, $p_1 = 1$. and $p_2 = 1/(n+1)$ where n is the total number of nucleotide bases in all input sequences. Finally, $p_3 = w_2/(|B| \cdot d_1^2 + 1)$, where $|B|$ is the number of atom boundaries, and d_1^2 is the maximum tertiary penalty for a single boundary. In our implementation, the same effect is achieved by storing the overall penalty as 3-tuple instead of using these weights.

Let us now illustrate the influence of the tertiary penalty in two different scenarios. First, consider two adjacent atoms and alignment boundaries scattered at their border region. If the boundaries are not scattered too widely, those two atoms will likely touch and the optimal position of the border is in the place where the combined tertiary penalty is minimal. This corresponds to our intuition where border of such atoms should be placed. Unfortunately, if we had several alignments starting at the same place, but the atom starting in this location would be preceded by a waste region, our scoring scheme would favor the atom to be extended maximally until the alignment boundaries touch the core of the atom. This is not very intuitive, as in this case it would be more desirable to have the atom start at the alignment boundary. To handle cases like this, it might be a good idea to modify the scoring function so that there is a balance between the first and the third penalization weights, or to allow atom extension past alignment boundaries only if it is extending towards some different nearby alignment boundary.

2.3 Pseudoatomization

In the first part of our algorithm, we create a *pseudoatomization*, which is a segmentation satisfying the first three rules from the definition. With our scoring scheme and alignments boundaries, we construct the optimal pseudoatomization. This pseudoatomization will be later further processed. All pseudoatoms will be checked for rule C4 and modified if needed. The constructed pseudoatomization is optimal with respect to the scoring scheme, but unfortunately we have no guarantee that an optimal pseudoatomization leads to the optimal atomization.

If we take a look at the first three rules, the goal of pseudoatomization is to split a sequence into non-overlapping segments (pseudoatoms), meeting the minimal length criteria and not having any alignment boundaries inside their cores. Višňovská et al. [17] in their work also construct a pseudoatomization as an intermediate product, before computing atomization. Their algorithm processes input sequence S and places a pseudoatom into every gap between alignment boundaries that is at least L bases long. This simple algorithm is not usable with our definition, because we allowed for alignment boundaries to reach into atom edges. This may lead to scenarios where each of two possible pseudoatoms exclude the existence of the other one, because both can reach sufficient length only if they extend past the alignment boundary. We describe three algorithms to find the optimal pseudoatomization. The first is a simple dynamic programming, and the other two decrease its running time. Note that if we have multiple input sequences $S_1, S_2, \ldots, S_n$ we can construct the optimal pseudoatomization for each of them separately (but the set of alignment boundaries will also include the boundaries of alignments between different sequences).

2.3.1 Simple dynamic programming

For sequence S with length n, and a set of alignment boundaries B, we can create the optimal pseudoatomization using a simple dynamic programming algorithm 1. For every index i from 0 to $n+1$, we will compute the lowest penalty achievable in a pseudoatomization of subsequence $[0 \dots i]$. For each index, we will store this penalty in array *subsequence*. We will also store additional information that tells us if an atom or a non-zero length waste segment is ending at the position, and where it started. This will allow us to reconstruct the pseudoatomization.

We start our justification of the algorithm with the observation that if we have pseudoatomization A with waste region ending at a certain position i, we can compute the penalty for the whole A as the sum of penalties for regions $[0 \dots i]$ and $[i \dots n]$, and we compute penalties for each of these two regions independently without knowing how the other region looks like. Therefore, if we place a waste end somewhere, it splits the problem into two sub-problems. If we know the best score we can achieve for all previous indexes, we can compute the lowest penalty for position i by considering all previous positions j. For each j we take the pre-computed penalty for subsequence $[0 \dots j]$ and sum it with the score for $[j \dots i]$. Function atomize tells us whether $[j \dots i]$ is an atom or a waste. In computation of $atomize(j, i)$ we are checking if segment $[j \dots i]$ can be a pseudoatom. Firstly we check if that region is long enough. Then we check alignment boundaries from B overlapping with $[j \dots i]$; if there are none reaching into the core, we can place a pseudoatom there. The penalty will be computed for alignment boundaries inside the atom, plus the penalty for a new waste region of length 0 (the number of atoms penalty). If an atom cannot be placed there, the penalty for the number of wasted bases will be returned, as we are not creating new waste or a new atom.

The previously described algorithm will create an optimal set of pseudo-atoms in $O(n^2(\log(|B|) + d_1))$ time where n is the length of the sequence we want to split, and $|B|$ is the number of alignment boundaries. The factor n^2 comes from the nested loops. Function *atomize* runs in time $\log(|B|) + d_1$, where we use the binary search to find the index of the pseudoatom core start and end in B, which we keep in a sorted order. This allows us to quickly figure out if some alignment boundary breaches into the core. We compute *b_penalty* by traversing the previous and next boundaries from the returned indexes, while in the range of the pseudoatom edge. In the worst-case scenario we will traverse d_1 indexes on each side of the pseudoatom.

Algorithm 1 Dynamic programming

```
1: $region\_type[0] = 1$        ▷ We manually set first index to waste end, starting at 0
2: $region\_start[0] = 0$
3: $subsequence[0] = p_2$
4: for $i = 1, 2, \ldots, n$ do
5:     for $j = 0, 1, 2, \ldots, i-1$ do
6:         $seq\_type, seq\_penalty = atomize(j, i)$
7:         $whole\_penalty = subsequence[j] + seq\_penalty$
8:         if $whole\_penalty < subsequence[i]$ then        ▷ update penalty
9:             $region\_type[i] = seq\_type$
10:            $region\_start[i] = j$
11:            $subsequence[i] = whole\_penalty$
12:        end if
13:    end for
14: end for
```

Function $atomize(start, end)$ checks if it is possible to create an atom from j to i. If so, it returns the penalty for alignment boundaries inside edges of that atom and for a new waste region. If an atom cannot be created, it returns the penalty for the number of wasted bases. The first returned parameter marks if an atom or a waste penalty is returned Def atomize(start,end):

```
15: $len = end - start$
16: if $len < L$ then
17:     return $1, p_1 * len$
18: end if
19: if Boundary in $[start + d_1 + 1, end - d_1 - 1]$ then
20:     return $1, p_1 * len$
21: end if
22: $b\_penalty = 0$
23: for Boundary $b$ in $[start, start + d_1]$ do
24:     $b\_penalty += p_3 \cdot (b - start)^2$
25: end for
26: for Boundary $b$ in $[end - d_1, end]$ do
27:     $b\_penalty += p_3 \cdot (end - b)^2$
28: end for
29: $b\_penalty += p_2$        ▷ penalty for new region
30: return 0, b_penalty
```

2.3.2 Algorithm speed-up

With a few simple observations, we can speed up the algorithm described in the previous section. Firstly, let us consider the part, where we are checking all sub-indexes j for some value of index i. For an easier explanation, we will iterate over values of j in the reversed order from $i-1$ to 0. In each step, we would be checking the score obtainable for the region $[j \dots i]$, which is getting longer in every step. For the first L indexes, the region is not long enough to be an atom, but after L steps the region might constitute a pseudoatom if no alignment boundary is reaching into its core. Every next decrease of index j then shifts potential boundaries closer to the core. Once a boundary reaches the core, every lower value of j will result in a waste region.

So we have a sweet spot of values of j, where we are finding pseudoatoms; elsewhere we are adding waste. We can further extend this idea by showing that it is sufficient to check only certain positions around a specific alignment boundary. A dummy alignment boundary is placed at index 0 and at the last indexed position, they don't affect pseudoatomization or its score, and only serve to simplify algorithm description.

First, for index i we will find its *restricting* alignment boundary. That is the alignment boundary b with the largest index from $[0, i - d_1 - 1]$. Any index $i < L$ is processed automatically as a waste segment, and for all other indexes a dummy boundary at index 0 guarantees the existence of a restricting alignment boundary. This alignment boundary is decisive for placement of j, setting smallest viable j to $b - d_1$ as any previous index j would place b into the atom core of $[j \dots i]$, thus leading to a waste region. Placement of j is not limited by any alignment boundary with greater index. For any atom ending at index i, alignment boundary with the index from $[i - d_2 \dots i]$ would end in the atom edge closer to i regardless of atom start index j. Also, alignment boundary b' preceding b is insignificant for determining smallest viable j, as for any atom $[j \dots i]$ where b' is inside the atom core, b have to be inside atom core also.

Next, we will show it is sufficient to check for atoms starting between $b - d_1$ and $b + d_1$. As any index lower than $b - d_1$ ends in waste and any index greater than $b + d_1$ is unnecessary to check due to Lemma 2.3.1.

Lemma 2.3.1. *For index i and its restricting alignment boundary b, if pseudoatom $[j \dots i]$ exists in optimal pseudoatomization then $j \leq b + d_1$.*

Proof. Assume we have optimal pseudoatomization containing atom $E = [j \dots i]$ while $j > b + d_1$. We will show the existence of a pseudoatomization with a lower penalty, thus contradicting that the initial pseudoatomization was optimal. We inspect base $j - 1$, which is either covered by some other atom or by waste segment. There either exists a waste segment $[k \dots j]$ longer than 0 or atom $[k \dots j]$.

If there is a waste segment, we construct new pseudoatomization where waste $[k \dots j]$ and atom $[j \dots i]$ is replaced with waste $[k \dots j-1]$ and atom $[j-1 \dots i]$, improving primary penalty - covering more bases by atoms, without effect on the number of waste regions (atoms) and alignment boundaries inside atom edges. All atoms except E remain intact.

If there is atom $E' = [k \dots j]$ its core has to be placed after b, because atom with core before b ends at most at index $b + d_1 \leq j - 1$ thus not covering base $j - 1$. We replace E' and E with a single atom $E_{new} = [k \dots i]$ and remove the waste segment $[j \dots j]$ while doing so. As there are no boundaries between b and E core, and the core of E' has to be placed after b, there are also no boundaries between core of E' and core of E. This guarantees there are no boundaries in the core of E_{new} as it spans cores of original atoms and gap between them. This leads to a new pseudoatomization, with improved penalty for the number of atoms, as we subtracted one, and no change to primary and tertiary penalty. Tertiary penalty remains the same because E_{new} end edge is the same as E end edge and E_{new} start edge is the same as E' start edge. There are no boundaries between those two edges (in E_{new} core) so there was no penalty for E start edge and E_n end edge. We have contradicted that original atomization was optimal. $\square$

Iterative waste build The next realization is that for waste segments, it is enough to check the score while adding a single base of waste to the previous indexed position. If for index i the optimal penalty is achievable with waste $[k \dots i]$, then for every index z from (k, i) the optimal penalty is achieved with waste $[k \dots z]$; for $k + 1$ optimal penalty is achieved by adding single waste to atom ending at position k. So we know that it is sufficient for each index i to add a penalty for one base in a waste to the score stored in *subsequence*$[i - 1]$ and then to check of a better score can be achieved by an atom starting in the region around the alignment boundary that is restrictive for pseudoatom start.

When those changes are incorporated into the previously described simple dynamic programming algorithm, new algorithm with running time of $O(n \cdot d_1 \cdot (\log(|B|) + d_1))$ is obtained. As we no longer need to go through all nested indexes, element n^2 becomes $n \cdot d_1$. Function *atomize* is not modified, and keeps running in time $O(\log(|B|) + d_1)$.

2.3.3 Boundary neighborhood exploration

It is possible to create an even faster algorithm if we smartly select the set of indexes for which we compute *subsequence* score. Previously we have pointed out that for index i, its restricting alignment boundary b and pseudoatom ending at index i, its start will be placed at index from $[b - d_1, b + d_1]$ as there is no way it can extend more than d_1 past b, and also no reason to place its start at index greater than $b + d_1$. As the start of each atom is placed in the d_1 neighborhood of some boundary, in an analogous way we can show that the end of each atom is in the d_1 neighborhood of some boundary. Specifically, for start index j of pseudoatom, we can find its restricting alignment boundary b_j (the first boundary index greater than $j + d_1$), in this case limiting pseudoatom end index i, as $i > b_j + d_1$ results in b_j inside $[j \dots i]$ atom core. The same construction of proof as for Lemma 2.3.1 is used to show that $i < b_j - d_1$ leads to suboptimally short pseudoatom.

This allows us to compute the achievable penalty only for indexes around boundaries. Given a set of alignment boundaries B, we will create a set of indexes C for which we want to compute *subsequence*. Namely, for every b from B we add $[b - d_1, b + d_1]$ to C, restricting the set to indexes that are at least 0 and at most the last indexed position of sequence.

In the algorithm, we will pass through indexes in C in order from smallest, and process them, while keeping two LIFO queues for boundaries possibly after and before the atom core. For each considered value of i, we check if an alignment boundary exists at that index, if so, it is added to the back of the first queue, and queues are then shifted accordingly. Each new alignment boundary firstly gets into the queue for alignment boundaries within distance d_1 from the current index i. Once index i moves and the front of the first queue is smaller than $i - d_1$, the boundary at the front is popped, and placed in the back of the second queue. The front of the second queue is maintained so that it is at distance at most d_1 from its back. Any alignment boundaries not satisfying this condition are removed from the queue. For every processed value of index i, we then either add a waste region to a previously processed index, or try placements of pseudoatoms starting within distance d_1 around the back of the second queue, which is the same as restricting boundary alignment for i. As we are computing the value of *subsequence* only for certain indexes, and the last previously processed index might be distant from the current index i, we are no longer adding a waste region of length 1 to the previous index, but possibly a longer waste region. This is sufficient because there is no logic in starting or ending waste regions at different indexes than atoms, and similarly as in *iterative waste build* a longer waste segment can still be built up iteratively, but the increments are not fixed to length of a single base.

This newly created algorithm has running time of $O(|B| \cdot d_1 \cdot (\log(|B|) + d_1^2))$. As we have at most $|B| \cdot 2d_1$ indexes for which we compute *subsequence* score (where the atom end is possibly placed), and for each of these indexes, we go through $2d_1$ sub indexes for atom start position. We no longer use the original *atomize* function, but for each atom (index and sub index combination), compute its penalty based on boundaries stored in the first and the second queue. Each of those queues holds at most d_1 boundaries. Managing the queues has amortized complexity of $O(1)$ for each of $|B|$ boundaries, as each boundary is moved once in each queue. This leads to running time of $O(|B| \cdot d_1^3))$. Initial sorting of $|B| \cdot 2d_1$ indexes has running time of $O(|B| \cdot d_1 \cdot \log(|B| \cdot d_1)$. Resulting running time is a combination of those two.

2.4 Atomization

Once pseudoatomization, as described in previous Section 2.3 is constructed, we have a set of segments fulfilling conditions 1, 2 and 3 from definition of atomization in Section 2.2.1. To obtain a full atomization, each pseudoatom needs to satisfy condition 4. For the pseudoatoms that do not, we introduce three operations to alter them to satisfy it if possible. Those operations are shortening a pseudoatom, splitting one pseudoatom into two or removing a pseudoatom. If a pseudoatom is shortened or split, the new pseudoatoms have to be at least L long; this will guarantee that the first three conditions are satisfied all the time. This should guarantee that:

1. The algorithm will end in a finite number of steps as atoms are only getting shorter, and once they are split, there is no way to combine them back.
2. The first three rules for atomization will be satisfied all the time, so in each step it is enough to check rule 4 and alter pseudoatoms accordingly.

2.4.1 Relevant pseudoatoms

For pseudoatom E and alignment a, where the core of E is in the source of a, we will go through all pseudoatoms E_x overlapping the target of a, and find those relevant for E and a. For pseudoatom E and alignment a we consider pseudoatom E_x in the alignment target relevant, if it passes one of these conditions:

1. $a(E)$ covers E_x.
2. The overlap of $a(E)$ with E_x has length at least $L - d_2$.
3. E_x covers $a(E)$ and $a(E)$ is at least $L - 2d_2$ long.

These rules consider E_x relevant, if there is a possibility that the projection of some sub-segment of E will match with the tolerance some sub-segment of E_x, while both those sub-segments will be at least L long. If $a(E)$ is not at least $L - 2 \cdot d_2$ long, it can match with tolerance only with segments shorter than L, and there is no way it will satisfy condition 4.

Symmetry of matching atoms

In Lemma 2.4.1 we show that matches with tolerance d_2 ($d_2 < L/2$) between atoms E_1 and E_2 in final atomization have to be *symmetrical*, if E_1 matches with tolerance E_2, E_2 matches with tolerance E_1. From now on, we will use terms *matches* and *matches with tolerance* interchangeably to describe match with tolerance between two atoms, and assume $d_2 < L/2$.

Lemma 2.4.1. *For atomization A, alignment a and atoms E_1 and E_2, if $a(E_1)$ matches with tolerance E_2, then $a'(E_2)$ matches E_1.*

Proof. First, we will show that projections $a(E_s)$ and $a(E'_s)$ of two distinct atoms E_s and E'_s from source cannot match the same atom E_t, based on two facts:

1. As the E_s and E'_s are not overlapping, their projections $a(E_s)$ and $a(E'_s)$ are also not overlapping.
2. If for one of the atoms, let us say E_s, $a(E_s)$ matches E_t, then E_t can extend at most d_2 past $a(E_s)$, and its farther end would be at least $L - d_2$ from $a(E'_s)$. This means that this edge of E_t is too distant from $a(E'_s)$ because $L - d_2 > d_2$.

Next, if $a(E_1)$ matches E_2, but $a'(E_2)$ does not match E_1, it has to match some other atom, otherwise it would violate condition $C4$ from Section 2.2.1, so let us say $a'(E_2)$ matched E_3 successor of E_1. Iteratively $a(E_3)$ has to match some different atom E_4 and $a'(E_4)$ matches E_5 and so one, as one atom cannot be matched by multiple atoms. This would lead to a long zig-zagging chain of atoms each matching atom next to the one they are matched by, until we get to the atom that has to match the atom already matched by his predecessor, because there is no other atom left. The same ordering of aligned bases in alignment source and target also prevents matched atoms from forming a cycle, as that would be possible only if for some atom E' successor of E_1, $a(E')$ preceded $a(E_1)$. Argument applies analogously if we picked E_3 as predecessor of E_1.

$\square$

Pseudoatom trimming position

One of the modifications of pseudoatoms we will perform in order to obtain atomization is trimming of existing pseudoatoms. We will take pseudoatom $E_1 = [j \dots i]$, and shorten it to $E'_1 = [l \dots k]$, sub-segment of $[j \dots i]$. Bases that were covered by $[j \dots i]$, but are not covered by $[l \dots k]$ have turned into waste, so we want to trim away as few bases as necessary. Consider pseudoatom $E_1 = [j \dots i]$, alignment a and relevant pseudoatom $E_2 = [f \dots e]$, we want to find minimal index l and maximal index k to which E_1 can be trimmed to match the E_2 or its sub-segment. If we take a look at l, it is determined by one of the three factors.

(i) Projection $a(E_1')$ can start at most d_2 indexes before E_2 starts. So the first matched base q_x from E_1' has to be aligned to the base t_x in the alignment target that is either at most d_2 indexes before f or inside E_2. After finding the leftmost q_x with this property, this limits l to be to the right of q_{x-1}, the matched base that precedes q_x.

(ii) If the projection of E_1' through a will match E_2, then E_2 through a' has to match E_1' in the final atomization. If the first aligned base t_z from E_2 is aligned with base q_z from the target of a', this limits E_1 to start at most d_2 indexes before q_z.

(iii) As we are not adding any new bases into pseudoatoms, l is limited by the position j where E_1 already starts.

To find the earliest position where E_1' can start, we pick the maximum from all three. In similar fashion, we can find the last index where E_1' can end given the current end of the E_1, index before base that is aligned to base too far from e and position where E_2 can match end of E_1'.

This way we obtain *trimming segment* $[l \dots k]$, where the l is the earliest start index and the k is the last end index. If $[l \dots k]$ is shorter than L, we will remove E_1 instead of trimming. When we trim the E_1 into segment $[l \dots k]$, we have wasted only bases for which there is no way to be preserved in the final atomization (given we are not allowing any extension of existing pseudoatoms or addition of new ones, and if E_2 is the only pseudoatom relevant for E_1 and a). For trimmed E_1 we have no guarantee it is matching E_2 yet, as additional trimming of E_2 might be needed first.

2.4.2 Splitting position

For the pseudoatom E and alignment a which have two relevant pseudoatoms E_1 and E_2, we could select a single one of them, and trim E accordingly, but this would lead to a lot of wasted bases. A better strategy is to split E into two pseudoatoms while wasting a minimal number of bases. Newly created atoms have to be at least L long to satisfy condition C2. To do so, we want to find the optimal index inside E where the split will be placed. To find an ideal position, we will collect trimming segments $[b \dots a]$ and $[d \dots c]$ for relevant atoms E_1 and E_2, respectively. As before, trimming segments tell us the earliest start and end of E so $a(E)$ match E_1 and E_2 respectively. We will take a look at the end of the first segment $[b \dots a]$ and the start of the second segment $[d \dots c]$. For indexes a and d one of three scenarios will occur:

1. $a > d$, segments are overlapping, and bases included in that overlap might belong into both new pseudoatoms, and splitting point s is placed into that overlap $d \leq s \leq a$, any different placement would lead to unnecessary wasted bases.

2. $a = d$, segments are touching, E can be split at index a, and split will not result in wasted bases.

3. $a < d$, segments are separated by a gap; for the bases in the gap, there is no way they could be included in atomization, and they have to be converted to a waste, which splits atom E.

In the second and third case, we have an optimal split position, as both resulting pseudoatoms will end up with maximum possible length. We will call this *soft splitting* as we have certainty of the ideal splitting position. We will call the first scenario *force splitting*, as we have to pick one among multiple splitting positions.

In the description of splitting we used two relevant pseudoatoms E_1 and E_2, but the same principle applies for any number of relevant pseudoatoms. If two subsequent relevant pseudoatoms generate non overlapping trimming segments, the split can be placed safely there and all such splits will be selected at once. If some of the trimming segments is shorter than L, it can be ignored. If all of the segments overlap we will place a split between the two segments based on rules in the following paragraph.

Force splitting continued Not all force splitting scenarios are equally good, and once we are performing force splitting, we want to avoid ones where we are placing split between very short segments, as this could lead to bases from overlap assigned to one of the atoms being wasted together with that atom in the future. We will select force split with preference of lowest category, and secondary shortest overlap. For the pseudoatom E, alignment a, and relevant pseudoatoms E_{k-1}, E_k, E_l, E_{l+1} and $[s_{k-1} \dots e_{k-1}]$, $[s_k \dots e_k]$, $[s_l \dots e_l]$, $[s_{l+1} \dots e_{l+1}]$ - their trimming segments for E, first we compute their split position p between E_k and E_l as following:

We will take the last aligned base from E_k and the first aligned base from E_l, and put p as centered between bases in the source they are aligned to as possible.

Once we have precomputed split position p, we categorize this split between E_k and E_l as following:

1. We place split in lowest category if $[e_{k-1} \dots s_l]$ and $[e_k \dots s_{l+1}]$ are both at least L long - this means there is enough non-overlapping bases in E_k and E_l trimming segments to create atom on their own, even if we placed splits around them in least favorable position. If E_k does not have a predecessor in E relevant atoms we inspect $[s_k \dots s_l]$, similarly we inspect $[e_k \dots e_l]$ if E_l has no successor in relevant atoms. For split in this category, split position p is used once splitting is performed.

2. For the second category, we modify the previous one to consider real placement of p and inspect segments $[e_{k-1} \dots p]$ and $[p \dots s_{l+1}]$ if they are at least L long. -

meaning if we make a split now, future splits should not turn them into waste, even if placed in least favorable positions. For no predecessor or successor, we check $[s_k \dots p]$ and $[p \dots e_{l+1}]$ respectively.

3. In the third category, we reduce the consideration of what effect will future splits have on our newly created segments. We inspect only $[s_k \dots p]$ and $[p \dots e_l]$ for length L. - meaning if we make a split now, ideal placement of future splits will not turn them into waste (if placed at s_k and e_l). Third category is same as previous if there were no E_{k-1} and E_{l+1}, we inspect only $[s_k \dots p]$ and $[p \dots e_l]$ for length L.

4. If previous condition failed, instead of p we will consider all possible split positions p' in overlap, and check if at least one exist such $[s_k \dots p']$ and $[p' \dots e_l]$ are both at least L long, if so, split is placed in the fourth category.

5. Last category is for splits where all placements of p' in overlap leads to at least one of trimming segments becoming shorter than L. We place split to preserve one, such p' have to exist, since both $[s_k \dots e_k]$, $[s_l \dots e_l]$ are at least L long so s_l or e_k as p' both conserve one of the trimming segments long enough.

Initial selection of split position might end up with two equally centered positions p. Split position p is used (split is placed there) for splits in the first three categories, but not in the condition of the first category. Second and third conditions are evaluated individually for each position p. If both of them result in the same category of split, we pick one at random, same as in the first category. If one of them places split in a higher category than the other, only the former is considered. Similarly, if we have multiple positions p' in the last two categories, we pick one at random. We have no guarantee this selection of split position is optimal.

2.4.3 Breaking rule 4

To summarize the previous discussion, the algorithm proceeds as follows. Consider pseudoatom E and alignment a, to assess if E passes rule 4, or can be modified to pass rule 4, we will first compute a list of relevant atoms for E and a. There are three possible outcomes: 1, list of relevant atoms will be empty, E can not be modified to pass rule 4, and can be removed. 2, One relevant atom exists, we want to modify E to match w.t. this relevant atom, bases from start and end of E can be trimmed 3, There are multiple relevant atoms, we want to split E into an equal number of pseudoatoms, where each pseudoatom created by splitting will have one relevant atom through alignment a.

Algorithm 2 Atomization

```
1: We start with a set of p.a. $P$, alignments $\alpha$ and constant $d_2$
2: $Q_{process} = \{\}$
3: $Q_{split} = \{\}$
4: for each $E$ from $P$ and $a$ from $\alpha$ do
5:     if $a$ source covers $E$ core then
6:         if $|a(E)| < L - 2 * d_2$ then delete $E$ and continue with next $E$
7:         $E.linked\_alignments$ add $a$
8:         $a'.target\_atoms$ add $E$
9:         $Q_{process}.add(E, a)$
10:    end if
11: end for
12: while $len(Q_{process}) + len(Q_{split}) > 0$ do
13:     if $len(Q_{process}) > 0$ then
14:         $E$, $alignment \leftarrow Q_{process}.pop()$
15:         $tasks = \{alignment\}$
16:         while $tasks$ not empty do
17:             $alignment \leftarrow tasks.pop()$
18:             $process(E, alignment)$
19:             if $E$ was trimmed or soft splitted then
20:                 $tasks = E.linked\_alignments$
21:             else $E$ was deleted
22:                 $tasks = None$
23:             end if
24:         end while
25:         If $E$ was modified or deleted then $Notify(E.linked\_alignments)$
26:     else
27:         $E$, $alignment \leftarrow Q_{split}.pop()$
28:         $split(E, alignment, force)$
29:         $Notify(E.linked\_alignments)$
30:     end if
31: end while
```

2.4.4 Atomization algorithm

We have presented two key parts of the algorithm, how to compute atom trimming position, and how to compute atom splitting position. Those computations will be used to perform one of three basic operations in our algorithm when processing existing pseudoatom. We will either trim pseudoatom, split pseudoatom or delete pseudoatom. Pseudoatom deletion wasn't discussed yet, but it is a simple step in which an atom is removed. There is one other possible outcome of pseudoatom processing, when we don't take any action, which happens in two cases. In the first case, the inspected pseudoatom satisfies condition 4, and none of its relevant pseudoatoms require it to trim itself, so it does not need to be processed. In the second case, in order for the pseudoatom to satisfy condition 4, its relevant atom has to trim itself, and no trimming or splitting is currently required for inspected pseudoatom.

Starting with a set of pseudoatoms P, a set of alignments α and constant d_2 and two empty sets $Q_{process}$ and Q_{split} we will take the steps outlined in Algorithm 2 to obtain atomization. We will keep in $Q_{process}$ pseudoatoms that need to be processed and in Q_{split} pseudoatoms requiring force splitting. First, we will link the alignments with the pseudoatoms, we want to have ability to say in which sources of alignments lays the core of a pseudoatom, and which pseudoatoms lay in the targets of alignments. Linked alignments are the ones we need to project pseudoatoms through, and match those projections with other pseudoatoms.

We will construct a projection of all pseudoatoms through all their linked alignments, and check if they are at least $L - 2d_2$ long. If not, the pseudoatom gets deleted, if yes, we add it to $Q_{process}$ which keeps the set of atoms and alignments we need to process. For initial processing, we want to process the pseudoatom through all its linked alignments.

Main loop While there is some element either in $Q_{process}$ or Q_{split} we keep processing pseudoatoms in the main loop (line 12 to 31 in Algorithm 2). We process elements from $Q_{process}$ and trim or soft split those. When there is no element in $Q_{process}$ left, we take one from Q_{split} and force split it, preferring one with the lowest category of force splitting.

When we trim or split some atom E_p it triggers a cascade of reprocessing. We need to reprocess E_p, or pseudoatoms produced by split, through all of their linked alignments, even if previously those did not require any processing. We also notify all atoms in all alignments linked to E_p because those should need to reprocess themselves if E_p is relevant to them. In Algorithm 2, this step is written as $Notify(E_p.linked_alignments)$. We put the aforementioned atoms into $Q_{process}$ together with alignment that they are linked to E_p through. If some pseudoatom E_r in target of alignment a linked to E_p is notified, and does not need to reprocess itself through a' there is no need to re-check it through other of its linked alignments.

As trimming and soft splitting are optimal steps, we can perform those anytime, and perform force splitting only if $Q_{process}$ is empty as there is no other action to be taken. Pseudoatoms created by split inherit list of linked alignments from the original pseudoatom.

2.4.5 Negative strand alignments

To simplify the explanation of the atomization algorithm, we have not discussed *inverse alignments*, where the segment from the forward strand is aligned to the segment on the reverse strand. For computation of pseudoatomization, this does not make any difference, as in those algorithms, only information taken from the alignment is the start and end index of each aligned segment, on the forward strand, regardless of their orientation. For the atomization algorithm, we are using aligned bases from alignment to construct projection of segment through that alignment, and based on pseudoatoms this projection is overlapping either split or trim original pseudoatom. Projection through *inverse alignment* leads to rotated projection, where the end of the pseudoatom is projected first followed by its start. This effect can be easily counteracted. Consider segment E and alignment a with forward strand source aligned to reverse strand target. To create projection, we will take the same steps as for non-inverse alignments, and find aligned bases from E and their counterparts in the a target. The first aligned base from E is used to define the end of $a(E)$, and the last aligned base from E is used to define the start of $a(E)$. For projection constructed this way, rule number 4 from the definition of atomization remains the same. From there, for $a(E)$ and some relevant atom E' we can compute the trimming position for start of E based on end of $a(E)$ and end of E'. Cutting away aligned bases from E start will lead to shortening of $a(E)$ end which can end at most d_2 indexes after E'. The last aligned base from E' projected through a' has to match with tolerance E start. We obtain rules similar to those used to compute trimming positions in regular alignments. The trimming segments for pseudoatoms in inverse alignment are then used to find splitting position in the same way as in a regular alignment, when put in the right order.

2.5 Pre and post processing

In addition to our core atomization algorithm, we need to run several other steps in order to prepare data used by our algorithm, and post process atomization it produces. Full pipeline begins with a fasta file, containing, for all sequences we want to atomize, their name and nucleotide bases; and ends with full atomization with atoms divided in classes. The goal is to show how full atomization is obtained using our scripts, enabling replication of results and usage of pipeline. Following steps are taken, and scripts run, to obtain atomization:

PSL alignment file is created by an external program; in our work we used either LASTZ [6] or LAST [9] aligner to obtain alignments in *psl* file format, where each alignment is represented by single line containing details about this alignment. Scripts *align.sh* and *align2.sh*, taking fasta file as argument, are wrappers of LASTZ and LAST, producing output in *psl* file format. Script *filter-psl.py* is then used to filter out weak and short alignments from a given *psl* file. Parameters for score and minimal length can be provided as well to override defaults. Script *align.sh* is taken from work by Brejová et al. [3]; script *filter-psl.py* is our python reimplementation of filtering also presented by Brejová et al.; *align2.sh* is our wrapper of LAST, obtained by modifying *align.sh*.

Atomize.py is the main file in our project, as it orchestrates usage of other bits of code to obtain atomization. As arguments, it takes input file (*.psl* file with alignments), minimal length of atom L, constants d_1 and d_2 for atom edges slack and tolerance of match. Alignments from the input file are loaded with help of *load_psl.py*. From there pseudoatomization is created first, using code from *pseudosementation.py*, and this is later processed into atomization with code from *segmentation.py*. Additional helper functions used in this process are stored in *tools.py*. Once atomization that satisfies our definition is created, it is stored in file format, where each atom is represented by a single line, containing sequence name, atom id, class, strand, start and end index. Atom id is a unique number assigned to each atom, atom class and strand are assigned to atoms based on graph where atoms are vertices, and alignments are edges, connecting two atoms that match with tolerance. Traversal of connected components in such a graph is used to assign the same class to its atoms, and assign strand placement to atoms based on whether they are connected by regular or inverse alignment. Strand assignment for all atoms in a component is flipped if this leads to the majority of atoms on positive strand. All the scripts mentioned in this paragraph are our own original work.

Perl-atoms.pl is used instead of *atomize.py* if we want to obtain *IMP* atomization. Alignment filtering might be skipped, as it is included in perl-atoms.pl. Output format is the same as in atomize.py. This is the original script for atomization presented by Višňovská et al. [17].

Ilp-atoms.pl is used to further improve placement of atoms into classes. For the graph constructed from atoms connected by alignments, same as in atomize.py, graph clustering problem as mentioned in Section 1.2.3 is solved. In this problem, edges are either added or removed to turn graph into cliques at minimal costs. Ideally, classes that were previously connected by a misplaced alignment are separated, and only strongly connected atoms are left in the same class. Strand is also reassigned, based on the updated graph. Previously created atoms file together with file for alignments are used as an input. This script is taken from work by Brejová et al. [3].

Collapse-atoms.py is the last step used in post processing of atoms. Two atoms E_1 from class C_1 and E_2 from class C_2 are merged together, if all atoms from C_1 are adjacent to some atoms from C_2 and vice versa, while the gap between them is smaller than the set parameter for maximum gap. Consistent strand placement of atoms is also required, meaning that atoms from C_2 have to be the successor to C_1 atoms on the positive strand or the successor on the negative strand. The successor on the negative strand is an atom with lower indexes for start and end, as we are indexing all atoms with respect to the positive strand. If this condition is met, all pairs of atoms from C_1 and C_2 are collapsed together, each creating one new atom, which keeps the strand and class of its parental atom from class C_1, and covers the original pair of atoms together with a gap between them. The insight behind this step is to restore longer real atoms, that might have been split due to a misplaced short alignment, or a short evolutionary event inside them. This script is our python reimplementation of the script presented by Brejová et al. [3].

Chapter 3

Experiments and results

To prove feasibility of our approach to atomization, and show its performance, we introduce three data sets, on which our algorithm was run and evaluated. Two of them are real data, and one of the data sets is simulated data. Running atomization on simulated data allows us to compare the results of our atomization algorithm *ACS* (*Atomization with core and slack*) with known correct atomization, and compare its accuracy with *IMP*, the algorithm previously introduced by Višňovská et al. [17], directly.

3.1 Methodology

3.1.1 Quality measurements

To evaluate the atomization, we have adopted previously used approaches of matching atoms from two atomizations, *Reciprocal best matches* (BRM) and *boundary fitting matches* (BFM). BRM was introduced by Brejová et al. [3]. Two atoms E_t from atomization A_t and E_p from atomization A_p are BRM to each other if E_t and E_p overlap, and overlap of E_t with any atom from A_p and E_p with any atom from A_t is shorter than their mutual overlap. BFM was introduced by Višňovská et al. [17]. Previously mentioned atoms E_t and E_p are BFM to each other, if their start and end indexes are at most in distance k from each other, where k is set parameter. Using $k < L/2$ guarantees each atom has at most one BFM match, which if exists, is also a BRM match. In practice, we are using $k = L/4$ for BFM. For atoms matched by BRM and BFM we may then measure their sensitivity and specificity, as well as sensitivity and specificity for their classes. For true atomization A_t with the number of atoms t_{atoms}, predicted atomization A_p with the number of atoms p_{atoms}, m_{brm} atom pairs matched by BRM and m_{bfm} atom pairs matched by BFM, we measure the following four metrics: *BRM specificity* as m_{brm}/p_{atoms}, *BRM sensitivity* as m_{brm}/t_{atoms}, *BFM specificity* as m_{bfm}/p_{atoms} and *BFM sensitivity* as m_{bfm}/t_{atoms}.

In a similar fashion, we may measure *BRM* $\backslash$*BFM* matches between two atom classes, C_t from A_t and C_p from A_p. Classes C_t and C_p are *BRM* $\backslash$*BFM* matched, if every atom from C_t is *BRM* $\backslash$*BFM* matched to an atom in C_p and vice versa. Since each atom has at most one *BRM* $\backslash$*BFM* match, this means atoms from a matched class are not matched to the atoms from any other class. We say classes C_t and C_p are *partially BRM$\backslash$BFM* matched if atoms from C_t are only matched to atoms from C_p and atoms from C_p are only matched to atoms from C_t, dropping requirement for all atoms from class to have some *BRM* $\backslash$*BFM* match. Specificity (*sp*) is then calculated as *matched/predicted* and sensitivity (*sn*) as *matched/true* for both *BRM* $\backslash$*BFM* and both fully (*sp* $\backslash$*sn*) or partially (*sp_p* $\backslash$*sn_p*) matched.

Apart from specificity and sensitivity there is one additional metric for *BRM$\backslash$BFM* and one for *BRM* only. The fraction of sequence covered by bases in *BRM$\backslash$BFM* pair overlaps is marked as *nucl* in tables. Metric *BOUND* measures what portion of *BRM* atom pairs have its end in distance up to $\lceil \frac{x}{2} \rceil$ from each other.

For data sets without known true atomization *BRM* $\backslash$*BFM* metric may still be used to compare two different atomizations, as the output shows similarity between the two compared atomizations.

Other than *BRM* and *BFM* metrics, we inspect *coverage* - what fraction of input sequence is covered by atoms, the number of atoms and the number of classes as well as the median and mean length of atoms in an atomization.

3.1.2 Data sets

We have inspected the performance of our algorithm on the three data sets described below.

Simulated data actually contains ten individually simulated data sets with known atomization. Those datasets were previously introduced by Brejová et al. [3] and used by Višňovská et al. [17]. The data sets were generated as a simulation of sequence evolution, including substitutions, short insertions and deletions, as well as large-scale duplication and deletion events. Each simulation started with a single sequence and followed human, chimpanzee and rhesus macaque phylogeny. This resulted in three sequences, one for each organism, each in length of roughly 400 *kb*. Main benefit of simulation is the existence of the real atomization, corresponding to shared ancestor relationships in the simulated evolution.

UGT1A primate gene cluster data set contains three sequences for human, chimpanzee and orangutan. Lengths of sequences are 90, 87 and 108 *kb* respectively. Data for the UGT1A cluster were taken from work by Višňovská et al. [17].

L	m.a.l.	d_1	d_2	BFM parameter
50	13	10	13	13
100	25	13	20	25
250	63	20	40	63
500	125	25	80	125

Table 3.1: Overview of parameters selected for atomization and evaluation based on selected minimal atom length L. *M.a.l.* represents minimal length of alignment, as shorter alignments were filtered out, also alignments with score lower than 0.8 were filtered out, regardless of L. *BFM* parameter sets maximal distance between ends of two atoms to be considered *BFM* match. Both *m.a.l.* and *BFM* are set to $L/4$. d_1 and d_2 are used only for the *ACS* algorithm, where d_1 sets atom edge length, and d_2 sets tolerance of match between atom projection and another atom.

Plague data set contains three strains of bacteria *Yersinia pestis*, that causes plague. Strains *Shasta*, *El Dorado* and *PBM19* were used [8]. Each strain contains one chromosome and three plasmids. Each chromosome has length of 4.6 *mb* and plasmids are in length of 96, 70 and 9 *kb*.

3.2 Atomization settings

For each data set and each run, we first create local alignments, and filter out alignments with low score or length shorter than $L/4$. To obtain alignments either *LASTZ* [6] or *LAST* [9] was used. Those alignments were used to construct atomization by *IMP* and our *ACS* algorithm. One of four minimal lengths of atoms $L = [50, 100, 250, 500]$ was used in each run, and the rest of parameters was selected accordingly to L (see Table 3.1). Same program was used for *IMP* and *ACS* atoms to put them in classes and collapse them, (see collapse-atoms.py in Section 2.5). *SibeliaZ* (SBZ) created atoms directly from sequence data and atoms shorter than L were discarded. Thus, data for (SBZ) are same for both LASTZ and LAST runs, as alignment file has no effect on SibeliaZ atomization.

Atomizations created on simulated data were then directly compared with known true atomization, and metrics for *BRM* and *BFM* were computed. Baseline for atomization was created from true atomization, where we filtered out atoms shorter than L and compared it with full true atomization. This is listed in the table under algorithm *TRUE*. We have repeated this process eight times, for four different minimal lengths of atoms $L = [50, 100, 250, 500]$, and two different aligners *LASTZ* and *LAST*. Known real atomization corresponding to shared ancestor relationships in the simulated evolution enables us to compare our algorithm directly with the *IMP* algorithm presented by Višňovská et al. [17].

UGT1A and plague do not have known real atomization, which might be compared with created atomizations. Thus, for atomizations created on those data sets, we may only draw an indirect comparison or compute $BRM \setminus BFM$ matches between two created atomizations. In this case $BRM \setminus BFM$ metrics serve as a form of similarity measurement, and should be interpreted only in this way. For UGT1A and Plague data sets, we have not created atomization for $L = 250$.

3.3 Results

3.3.1 Simulated data

Each of the ten data sets from simulated data was atomized individually, for each combination of alignments source and L. For the sake of clarity, we compiled scores for all ten data sets in a single run and provided their average.

ACS versus IMP In Table 3.2 we have comparison of *TRUE* atomization, *IMP* atomization and *ACS* atomization for each minimal atom length L, and rest of parameters from Table 3.1. If we take a look at *IMP* and *ACS* performance, we may see a change in trends between $L = 100$ and $L = 250$. For small d_2 in $L = 50$ and $L = 100$ we require atoms in *ACS* to match with high precision, because tolerance of match (d_2) is either 13 or 20 bases long, even if median atom is over 1500 bases long. This leads to a lot of trimming and deletion of some atoms that cannot be trimmed to match, as seen in lower *BRM* sensitivity, lower coverage, and lower number of atoms than in *IMP*. On the other hand, *IMP* in those cases cares only about the existence of a single overlap of each atom projection, regardless of its quality. Thus *ACS* leads in *BRM* specificity, as some of the weak atoms were deleted, but loses in sensitivity and coverage a little.

With larger L, d_2 may be set larger, and match of atom projection is given more freedom. *ACS* is able to keep high coverage, and produces more atoms than *IMP*. This may be attributed to two reasons, first *ACS* may create more pseudoatoms due to tolerance of alignment boundaries in atom edges, and once those pseudoatoms are created, their projections may overlap other atoms slightly, as long as they match some atom with the tolerance. On the other hand, each such overlap needs to be trimmed down in *IMP*. While both *IMP* and *ACS* achieve 100% specificity in *BRM*, *ACS* has slightly higher sensitivity, presumably due to the higher number of atoms produced.

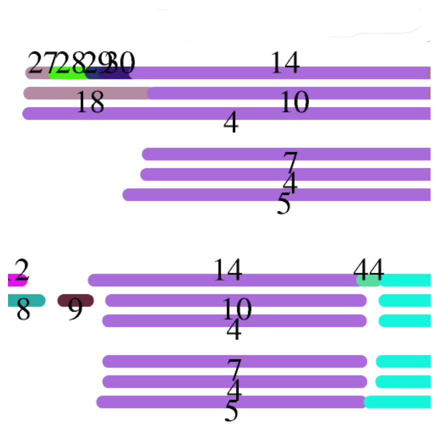
In *BFM* metrics, *ACS* atomization outperforms *IMP* for each selected L, this was expected behavior, as *BFM* match corresponds to our definition of atoms projection match, while *IMP* leaves more freedom in the size of overlap.

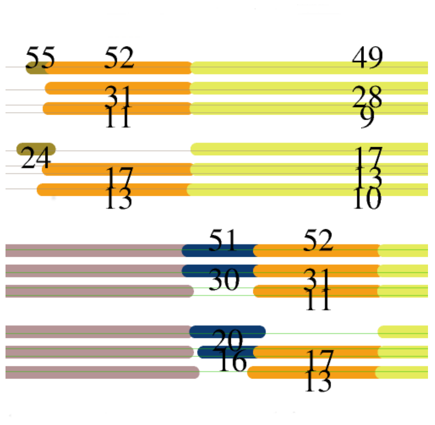
LASTZ versus LAST The second table, Table 3.3, holds results for *TRUE* atomization, and *ACS* atomization obtained from LASTZ and LAST alignments. LAST creates more alignments (see Table 3.6 in Appendix). For low L, this leads to higher number of shorter atoms, higher sensitivity (BRM) and coverage at the cost of low specificity when compared to atomization created from LASTZ alignments. This can be seen mainly in $L = 50$, where specificity of LAST atomization is over 20 percentage points lower than that of LASTZ, the number of atoms for LAST atomization is over 28 percent higher than true number of atoms. Increasing L limits placement of very short atoms in LAST atomization. Number of atoms, as well as median and mean atom length gets closer to *TRUE* and LASTZ atomization. Coverage gets slightly worse than for LASTZ, specificity improves, LAST loses lead in sensitivity.

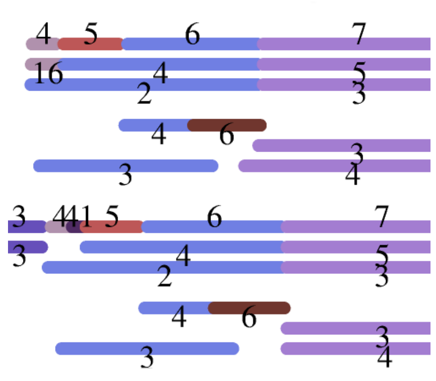
This makes *LASTZ* preferred source of alignments for, as atomization created from those alignments has overall better performance, clearly leads in *BFM* metrics, and is better reconstruction of true atomization - this can be seen at specificity, number of atoms and their lengths as well as coverage of sequence by $BRM \setminus BFM$ overlaps (*nucl* metric).

SibeliaZ atomization

For all data sets mentioned, we have constructed atomization using *SibeliaZ* [7], as a comparison of different approaches to atomization. SibeliaZ takes only sequence file, and skips the step of local alignments creation. To perform in a similar manner, SibeliaZ threshold value for minimal length of blocks outputted was used to limit minimal atom length. This does not only filter out blocks that are too short, but affects the whole creation of atomization by SibeliaZ. We have not added SibeliaZ atomization into Table 3.2 to compare with *IMP* and *ACS* on simulated data as detailed comparison is unnecessary. For instance, for $L = 50$, *SibeliaZ* achieved coverage of 76.7%, *BRM* *atom_sp* of 81.8% and *atom_sn* of 88.8%. While sensitivity is only 5 percentage points lower, other metrics dropped by 15 percentage points or more. Results for *BFM* are even worse, as *SibeliaZ*, for $L = 50$, has achieved score in all *BFM* metrics in range from 13.7% to 18.6%, that is over 50 percentage points lower than *IMP* or *ACS*. While approach used by *SibeliaZ* may have its benefits, or yield better results in certain scenarios, for our data sets, and using our measurements, direct comparison with *IMP* or *ACS* does not show them.


(a) Figure A


(b) Figure B


(c) Figure C

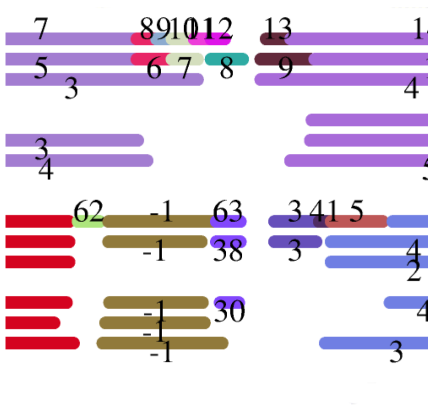

(d) Figure D

Figure 3.1: Four figures illustrating differences in atomizations produced by *IMP* and *ACS* on UGT1A gene cluster. Complete atomizations of three sequences (Human, Chimp and Orangutan) were created, and selected fragments from them are shown in figures. Each figure contains fragments from three IMP and three ACS atomizations, with $L = [50, 100, 500]$, created from LASTZ alignments. Atoms depicted are from the Human and Chimp sequence, Orangutan sequence was also atomized, but it is not used in image. When we mark each atomization by L, we have, in lines from top, for Human IMP50, IMP100, IMP500, ACS50, ACS100, ACS500, and the next six lines are the same atomizations for Chimp. Numbers assigned to atoms are their classes, and are comparable only in the same atomization. Atoms in the same regions from different atomization were assigned the same color, as those represent the same true atom. Individual sub-figures are discussed in Section 3.3.2.

3.3.2 UGT1A gene

In Table 3.4 we introduce statistics for *ACS* and *IMP* atomizations for three different values of L, created from *LASTZ* and *LAST* alignments as well as three SibeliaZ atomizations. In atomizations from LASTZ alignments, we can see the number of atoms dropping significantly in IMP for longer L, while in ACS the change is subtle, as the number was lower from beginning. Also, coverage for IMP drops, while for ACS it rises with longer L. This can be explained by differences in algorithms, which we illustrate in Figure 3.1 and discuss below. Main difference is, while ACS struggles with placement of atoms for low L, where d_2 is also low (atoms have to match precisely), and short alignments are not filtered out, IMP fills nearly the whole sequence with short atoms. This reflects in all statistics, number of atoms and classes, their mean and median, as well as ACS shortest atom in $L = 100$ being 214 bases long (it was not created by collapsing two shorter atoms). Once L is set to 500, ACS and IMP create similar atomizations, this shows in the table, but also when we take ACS as true atomization and compute their BRM and BFM matches. BRM achieves specificity of 99% and sensitivity of 97%, in BFM those scores are 67% and 66% respectively. And while SibeliaZ is also able to score in BRM (sp 60%, sn 90% to ACS, $L = 500$), in BFM its scores are less than 1%.

In LAST atomizations, we see a similar effect as discussed in **LASTZ versus LAST**, where LAST creates more alignments, leading to a higher number of shorter atoms. As we do not know true atomization, we can not verify if this also leads to lower BRM\BFM sensitivity and specificity.

In Figure 3.1a we want to point out two differences between IMP and ACS. Both found presumably correct purple atom, but for shorter L, IMP is able to fill space left of that purple atom with very short atoms (IMP50 and IMP100), and then in Human IMP500 (3rd line) extend purple atom to cover that region, once its not occupied by short alignments. We see discrepancy between purple atoms in IMP500 and ACS500, while they roughly match in Chimp, and ACS one from Human has to match the one from Chimp, for IMP overlap is enough.

In Figure 3.1b we may observe contra intuitive behavior, where ACS is able for $L = [100, 500]$ to place an orange atom, which it did not manage to place for $L = 50$. This can be explained with increase in constant d_2 for longer L, increasing tolerance of match of orange atom with its other copies.

Figure 3.1c shows us, that while IMP finds some variant of blue atom for each L, ACS needs ideal combination of L and d_2, to filter out some alignments, while having enough tolerance d_2 and being allowed to place short enough atoms.

Lastly Figure 3.1d illustrates multiple behaviors, brown atom on negative strand (in class -1) was created in ACS500, but not IMP500. This is due to one of the copies of the atom being longer than L only because parameter d_1 allowed boundary in its edge. Other than that, we see IMP placing very short atoms of questionable quality whenever possible; for instance atom in class 62, left of previously discussed brown atom, is the only atoms in its class.

3.3.3 Plague genomes

Table 3.5 holds comparison of atomization created by SibeliaZ and two atomizations created by ACS and IMP algorithm from LASTZ and LAST alignments. In table, we can see differences between algorithms described in Section 3.3.2 and illustrated in Figure 3.1 having similar effect, but not amplified to same extent, presumably due to different structure of data as we are atomizing whole genomes, not just a gene cluster. In atomizations from LASTZ alignment, ACS is able to achieve high coverage even for short L. ACS still produces fewer atoms in fewer classes than IMP, leading to greater atoms mean and median (also longer longest atom). In comparison to UGT1A atomization, ACS drop in the number of atoms for different L is more noticeable, and such drop was not observed even in simulated data.

With exception of this drop, we see patterns we have observed previously. For example, using LAST alignments leads to a higher number of shorter atoms and higher coverage compared to LASTZ, and we can only assume how they would compare to the true atomization. Again, ACS and IMP atomizations created from LASTZ alignments for $L = 500$ are similar. As IMP matched to ACS has BRM sensitivity and specificity of 97.8% and 100% respectively and BFM sensitivity and specificity of 95% and 97% respectively. SibeliaZ matched to ACS keeps high sensitivity in BRM (93.5%), but BFM sensitivity and specificity drops below 1%.

We can see that selection of threshold for shortest block SibeliaZ outputs in atomization affects it significantly. In an unexpected manner it produces the highest number of atoms among the three atomizations when threshold is set to 500, as it affects whole computation of atoms.

	min. atom length	L = 50			L = 100			L = 250			L = 500		
	algorithm	TRUE	IMP	ACS	TRUE	IMP	ACS	TRUE	IMP	ACS	TRUE	IMP	ACS
COVERAGE		100.0%	99.8%	98.6%	99.9%	99.8%	99.8%	99.7%	99.5%	99.7%	98.6%	98.5%	99.0%
BRM	BRM_sp	100.0%	98.0%	98.2%	100.0%	99.8%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
	BRM_sn	97.4%	95.9%	94.2%	95.0%	93.7%	93.2%	89.7%	88.1%	88.7%	79.5%	77.2%	78.7%
	BRM_nucl	100.0%	98.9%	97.2%	99.9%	98.8%	98.6%	99.7%	98.5%	98.5%	98.6%	96.9%	97.2%
	CLS_sn	98.4%	96.1%	94.5%	96.8%	94.8%	94.6%	93.0%	91.0%	91.5%	85.5%	82.6%	84.0%
	CLS_sn_p	98.4%	96.8%	94.9%	96.8%	95.1%	94.6%	93.0%	91.5%	91.5%	85.5%	83.2%	84.0%
	CLS_sp	100.0%	97.3%	98.0%	100.0%	99.2%	100.0%	100.0%	99.4%	100.0%	100.0%	99.4%	100.0%
	CLS_sp_p	100.0%	98.0%	98.4%	100.0%	99.5%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
	BOUND	100.0%	88.6%	90.0%	100.0%	94.7%	96.2%	100.0%	98.9%	98.4%	100.0%	97.8%	98.0%
BFM	ATOM_sp	100.0%	77.2%	80.9%	100.0%	88.1%	91.1%	100.0%	96.5%	96.7%	100.0%	97.6%	97.9%
	ATOM_sn	97.4%	75.6%	77.6%	95.0%	82.7%	84.9%	89.7%	85.0%	85.8%	79.5%	75.3%	77.0%
	ATOM_nucl	100.0%	75.6%	75.8%	99.9%	85.4%	87.6%	99.7%	94.5%	94.0%	98.6%	91.8%	92.8%
	CLS_sn	98.4%	66.7%	72.8%	96.8%	78.3%	82.6%	93.0%	86.4%	87.7%	85.5%	80.1%	81.7%
	CLS_sn_p	98.4%	81.6%	81.4%	96.8%	86.8%	88.4%	93.0%	88.6%	89.2%	85.5%	80.8%	81.7%
	CLS_sp	100.0%	67.5%	75.5%	100.0%	81.9%	87.3%	100.0%	94.4%	95.8%	100.0%	96.3%	97.3%
	CLS_sp_p	100.0%	82.6%	84.4%	100.0%	90.8%	93.5%	100.0%	96.8%	97.5%	100.0%	97.1%	97.4%
# atoms		375.2	377.1	369.4	365.8	361.6	359	345.6	339.2	341.7	306.3	297.2	303
# classes		111.6	112	109.4	109.8	108.4	107.3	105.5	103.8	103.8	97	94.3	95.2
atoms median		1620.5	1601	1590.5	1698	1688.5	1737.5	1863	1879	1882	2191	2222	2198
atoms mean		3549.01	3526.17	3555.49	3638.30	3674.33	3700.79	3841.12	3906.52	3887.80	4288.68	4413.95	4350.25

Table 3.2: Atomization of simulated data set with *IMP* and *ACS* algorithm. *TRUE* atomization was created from true atoms, with atoms shorter than *L* filtered out. *BRM* and *BFM* metrics for all atomizations were computed with matches in full true atomization. For selected *L*, rest of the parameters can be found in Table 3.1

	min. atom length	L = 50			L = 100			L = 250			L = 500		
	algorithm	TRUE	LAST	LASTZ	TRUE	LAST	LASTZ	TRUE	LAST	LASTZ	TRUE	LAST	LASTZ
COVERAGE		100.0%	99.6%	98.6%	99.9%	99.9%	99.8%	99.7%	99.6%	99.7%	98.6%	98.5%	99.0%
BRM	BRM_sp	100.0%	77.4%	98.2%	100.0%	89.6%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
	BRM_sn	97.4%	97.2%	94.2%	95.0%	95.4%	93.2%	89.7%	89.1%	88.7%	79.5%	78.7%	78.7%
	BRM_nucl	100.0%	92.7%	97.2%	99.9%	96.8%	98.6%	99.7%	98.6%	98.5%	98.6%	96.8%	97.2%
	CLS_sn	98.4%	97.3%	94.5%	96.8%	96.5%	94.6%	93.0%	92.2%	91.5%	85.5%	84.1%	84.0%
	CLS_sn_p	98.4%	98.0%	94.9%	96.8%	96.6%	94.6%	93.0%	92.2%	91.5%	85.5%	84.1%	84.0%
	CLS_sp	100.0%	73.9%	98.0%	100.0%	88.4%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
	CLS_sp_p	100.0%	74.5%	98.4%	100.0%	88.6%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%
	BOUND	100.0%	80.8%	90.0%	100.0%	90.1%	96.2%	100.0%	96.5%	98.4%	100.0%	94.6%	98.0%
BFM	ATOM_sp	100.0%	60.6%	80.9%	100.0%	78.9%	91.1%	100.0%	94.1%	96.7%	100.0%	92.1%	97.9%
	ATOM_sn	97.4%	76.0%	77.6%	95.0%	84.0%	84.9%	89.7%	83.8%	85.8%	79.5%	72.5%	77.0%
	ATOM_nucl	100.0%	68.4%	75.8%	99.9%	85.5%	87.6%	99.7%	92.7%	94.0%	98.6%	87.9%	92.8%
	CLS_sn	98.4%	75.7%	72.8%	96.8%	84.7%	82.6%	93.0%	87.0%	87.7%	85.5%	77.7%	81.7%
	CLS_sn_p	98.4%	79.5%	81.4%	96.8%	87.7%	88.4%	93.0%	88.4%	89.2%	85.5%	78.9%	81.7%
	CLS_sp	100.0%	57.5%	75.5%	100.0%	77.6%	87.3%	100.0%	94.4%	95.8%	100.0%	92.3%	97.3%
	CLS_sp_p	100.0%	60.5%	84.4%	100.0%	80.4%	93.5%	100.0%	95.9%	97.5%	100.0%	93.8%	97.4%
# atoms		375.2	483.5	369.4	365.8	409.9	359	345.6	343.1	341.7	306.3	303	303
# classes		111.6	149.2	109.4	109.8	123.7	107.3	105.5	104.6	103.8	97	95.4	95.2
atoms median		1620.5	1142	1590.5	1698	1405	1737.5	1863	1846	1882	2191	2188.5	2198
atoms mean		3549.0	2744.9	3555.5	3638.3	3246.0	3700.8	3841.1	3866.9	3887.8	4288.7	4327.7	4350.2

Table 3.3: Atomization of simulated data set with *ACS* algorithm, for two different sources of alignment generated by LASTZ and LAST. *TRUE* atomization was created from true atoms, with atoms shorter than L filtered out. *BRM* and *BFM* metrics for all atomizations were computed with matches in full true atomization. For selected L, rest of the parameters can be found in Table 3.1

		L = 50			L = 100			L = 500		
	algo.	**ACS**	**IMP**	**SBZ**	**ACS**	**IMP**	**SBZ**	**ACS**	**IMP**	**SBZ**
LASTZ	atoms	147	420	631	128	258	457	128	126	193
	classes	29	69	138	25	45	103	22	22	47
	coverage	76.31%	93.67%	88.48%	79.70%	92.30%	88.01%	83.98%	87.19%	73.57%
	shortest at.	50	51	49	214	101	99	501	503	515
	at. mean	1483.0	637.1	400.6	1778.9	1022.0	550.2	1874.3	1976.9	1089.1
	at. median	685	133	140	1075	501.5	201	1155	1139	708
LAST	atoms	431	526	631	361	355	457	97	100	193
	classes	86	108	138	59	64	103	20	21	47
	coverage	87.18%	95.85%	88.48%	91.75%	91.63%	88.01%	75.99%	73.34%	73.57%
	shortest at.	50	51	49	100	101	99	507	501	515
	at. mean	577.9	520.6	400.6	726.1	737.4	550.2	2237.9	2095.3	1089.1
	at. median	153	164.5	140	268	295	201	1420	984	708

Table 3.4: UGT1A data set atomization statistics. Atomization created with *IMP*, *ACS* and *SibeliaZ* (SBZ). Statistics for both atomizations created from LASTZ and LAST alignments. As the alignment file is not used for *SBZ* atomization, it contains the same data for both. For selected L, rest of the parameters can be found in Table 3.1

		L = 50			L = 100			L = 500		
	algo.	**ACS**	**IMP**	**SBZ**	**ACS**	**IMP**	**SBZ**	**ACS**	**IMP**	**SBZ**
LASTZ	atoms	1875	3485	5959	1447	2219	4853	803	821	6365
	classes	347	511	1360	263	338	1191	160	163	1912
	coverage	98.46%	99.00%	99.73%	98.42%	99.09%	99.67%	98.75%	98.67%	95.30%
	longest at.	115600	67508	25794	115606	97390	25794	223425	223375	25793
	at. mean	7632.5	4128.9	2432.6	9886.8	6490.7	2985.1	17874.0	17468.5	2176.3
	at. median	997	348	805	1443	625	1354	2874	2666	1089
LAST	atoms	2712	3414	5959	2249	2357	4853	1194	801	6365
	classes	427	559	1360	339	369	1191	223	160	1912
	coverage	99.66%	99.35%	99.73%	99.62%	99.41%	99.67%	98.72%	98.77%	95.30%
	longest at.	73848	68452	25794	97404	97391	25794	106612	223472	25793
	at. mean	5341.4	4230.0	2432.6	6438.5	6130.5	2985.1	12017.9	17923.2	2176.3
	at. median	467	495	805	544	548	1354	2056	3077	1089

Table 3.5: *Yersinia pestis* strains data set atomization statistics. Atomization created with *IMP*, *ACS* and *SibeliaZ* (SBZ). Statistics for both atomizations created from LASTZ and LAST alignments. As the alignment file is not used for *SBZ* atomization, it contains the same data for both. For selected L, rest of the parameters can be found in Table 3.1

Conclusion

In this thesis, we have revisited the problem of atomization created from local alignments, presented in earlier works by Brejová et al. [3] and Višňovská et al. [17]. We introduced a new formal definition of the problem, inspired by the definition by Višňovská et al., which aligns better with our understanding of what atoms really are. The main goal of our improved formal definition of atomization was to tighten requirements of how two atoms in alignment source and target should match, as well as to account for slight errors in alignment end placements.

Atomization is produced in two steps, first we create pseudoatoms, segments fulfilling the first three conditions of our definition. We have presented a dynamic programming algorithm leading to an optimal set of pseudoatoms, and improved its time complexity with multiple modifications. In the second phase, atomization is created by modification of pseudoatoms. We modify pseudoatoms by trimming and splitting to satisfy all conditions.

We have tested and compared the accuracy of our new algorithm with an older algorithm (*IMP* [17]) on simulated as well as real data. On simulated data, with known true atomization, atomization created with our algorithm was able to outperform the one created with *IMP* in most measures in all scenarios. On real data, we may see differences in observable characteristics of our atomizations, and those created with *IMP* and *SibeliaZ* [13], however we have no way of telling which one is correct and may only make assumptions, based on similarities with atomizations created on simulated data with known true atomization.

The result of this thesis is a computer program, which creates atomization of input sequences satisfying our definition. Created atomization might then be used in further analysis of inspected sequences, simplifying reconstruction of large-scale evolutionary events.

In the end, we list several possible improvements to our algorithms and definition. In the current algorithms, in situations where particular bases could be placed in one of two pseudoatoms or atoms, we have no guarantee that our approach leads to the optimal atomization. This may happen when two pseudoatoms touch, and their ends might be placed at one of multiple indexes, or when we are force splitting. In the first case, bases in that overlap might end up wasted if assigned to one pseudoatom, or preserved in other. This could be possibly counteracted by expansion of atoms once atomization is created, if such an expansion will not break atomization definition. In case of force splitting, collecting all splits through all alignments, for all atoms, and placing the split based on this information might improve accuracy. This proposed solution resembles the step present in the *IMP* algorithm, but due to differences in definitions (namely restriction to match of atoms in condition 4), its implementation for our atomization is more complicated. Alternatively, we may try to improve the performance of force splitting by running an atomization algorithm multiple times for one pseudoatomization. In this scenario, the algorithm has to be modified, to select order of force splits at random, maybe even select a split position randomly from overlap, and in the end output single atomization with highest score.

We propose two changes of atomization definition, based on observed behavior of our algorithm in experiments. In the first modification, a pseudoatom might be allowed to extend past alignment boundary, only if it is getting closer to the boundary of another alignment, which covers its core. The second possible change worth exploring might be to allow the atom edge length (d_2) to scale dynamically, giving longer atoms higher tolerance in their projection match with tolerance. As currently, to achieve higher tolerance, we have to increase L and lose very short atoms. Ideally this would lead to atomization with short atoms included, and long atoms not trimmed extensively. For such atomization, the BFM measure might also be adapted accordingly.

Bibliography

[1] Samuel V. Angiuoli and Steven L. Salzberg. Mugsy: fast multiple alignment of closely related whole genomes. *Bioinformatics*, 27(3):334, 2010.

[2] Holly C Betts, Mark N Puttick, James W Clark, Tom A Williams, Philip CJ Donoghue, and Davide Pisani. Integrated genomic and fossil evidence illuminates life's early evolution and eukaryote origin. *Nature ecology & evolution*, 2(10):1556–1562, 2018.

[3] Brona Brejova, Michal Burger, and Tomas Vinar. Automated Segmentation of DNA Sequences with Complex Evolutionary Histories. In Teresa M. Przytycka and Marie-France Sagot, editors, *Algorithms in Bioinformatics, 11th International Workshop (WABI)*, volume 6833 of *Lecture Notes in Computer Science*, pages 1–13, Saarbrücken, Germany, September 2011. Springer.

[4] Alexander Dobin, Carrie A Davis, Felix Schlesinger, Jorg Drenkow, Chris Zaleski, Sonali Jha, Philippe Batut, Mark Chaisson, and Thomas R Gingeras. STAR: ultrafast universal RNA-seq aligner. *Bioinformatics*, 29(1):15–21, 2013.

[5] Guénola Drillon, Raphaël Champeimont, Francesco Oteri, Gilles Fischer, and Alessandra Carbone. Phylogenetic reconstruction based on synteny block and gene adjacencies. *Molecular biology and evolution*, 37(9):2747–2762, 2020.

[6] Robert S Harris. *Improved pairwise alignment of genomic DNA*. The Pennsylvania State University, 2007.

[7] Andreas Heger and Liisa Holm. Exhaustive enumeration of protein domain families. *Journal of Molecular Biology*, 328(3):749 – 767, 2003.

[8] Shannon L Johnson, Hajnalka E Daligault, Karen W Davenport, James Jaissle, Kenneth G Frey, Jason T Ladner, Stacey M Broomall, Kimberly A Bishop-Lilly, David C Bruce, Susan R Coyne, et al. Thirty-two complete genome assemblies of nine Yersinia species, including Y. pestis, Y. pseudotuberculosis, and Y. enterocolitica. *Genome announcements*, 3(2):e00148–15, 2015.

[9] Szymon M Kiełbasa, Raymond Wan, Kengo Sato, Paul Horton, and Martin C Frith. Adaptive seeds tame genomic sequence comparison. *Genome research*, 21(3):487–493, 2011.

[10] Eugene V. Koonin. Orthologs, paralogs, and evolutionary genomics 1, December 2005.

[11] Mathieu Lajoie, Denis Bertrand, and Nadia El-Mabrouk. Inferring the evolutionary history of gene clusters from phylogenetic and gene order data. *Molecular biology and evolution*, 27(4):761–772, 2010.

[12] Vincent J Lynch, Oscar C Bedoya-Reina, Aakrosh Ratan, Michael Sulak, Daniela I Drautz-Moses, George H Perry, Webb Miller, and Stephan C Schuster. Elephantid genomes reveal the molecular bases of woolly mammoth adaptations to the Arctic. *Cell reports*, 12(2):217–228, 2015.

[13] Ilia Minkin and Paul Medvedev. Scalable multiple whole-genome alignment and locally collinear block construction with SibeliaZ. *Nature communications*, 11(1):1–11, 2020.

[14] Ilya Minkin, Anand Patel, Mikhail Kolmogorov, Nikolay Vyahhi, and Son Pham. *Sibelia: A Scalable and Comprehensive Synteny Block Generation Tool for Closely Related Microbial Genomes*, pages 215–229. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[15] Kay Prüfer, Fernando Racimo, Nick Patterson, Flora Jay, Sriram Sankararaman, Susanna Sawyer, Anja Heinze, Gabriel Renaud, Peter H Sudmant, Cesare De Filippo, et al. The complete genome sequence of a Neanderthal from the Altai Mountains. *Nature*, 505(7481):43–49, 2014.

[16] Diego P Rubert, Fábio V Martinez, Jens Stoye, and Daniel Doerr. Analysis of local genome rearrangement improves resolution of ancestral genomic maps in plants. *BMC genomics*, 21(2):1–11, 2020.

[17] Martina Višňovská, Tomáš Vinař, and Broňa Brejová. DNA Sequence Segmentation Based on Local Similarity. In Tomáš Vinař, editor, *Information Technologies - Applications and Theory (ITAT)*, volume 1003 of *CEUR-WS*, pages 36–43, 2013.

Appendix

This thesis comes with digital attachment either published at same web page, or downloadable from web `https://gitlab.com/simeunovic/atomization` under name *digital_attachment.zip.*

It contains following folders and files:

Folder *code* contains code and scripts created by us, as well as other scripts created by Višňovská et al. [17] and Brejová et al. [3] (sub-folders *scripts-tina* and *scripts* respectively). Sub-folder *git* is a copy of the aforementioned git repository, and contains scripts for *ACS* atomization plus a few extra scripts that are our reimplementation of existing scripts.

We have three folders, one for each data set, inside them, we may find sub-folders for each atomization run encoded by combination of L and source of alignment (lastal \al for LAST and lastz \z for LASTZ). Inside we keep SibeliaZ atoms (*.sbz_atom), *IMP* atoms (*.tatoms3) and *ACS* atoms (*.newdatoms4).

For *UGT1A*, we have also a folder with input fasta file, and created LASTZ (ugt1a.fasta.psl) and LAST (ugt1a.fasta.2.psl) alignments. Folder *ugt1a_stats* contains a file with basic statistics for each algorithm, L and alignment.

Similarly for *plague*, the sub-folder *plague_stats* contains basic statistics for combination of algorithm, L and alignment. Fasta file is not included due to size restrictions, but the README file contains instructions for creation of the input fasta file.

For a simulated data set, we provide a folder with fastas and alignments, and two folders with summed stats. One for LAST and one for LASTZ atomizations. Inside are not basic statistics of atomizations, but files with $BRM \setminus BFM$ metrics for SibeliaZ atoms (sbz_atom.*), *IMP* atoms (comp.*) and *ACS* atoms (newdat.*).

Each of the sub-folders with atomization also contains *Makefile* that was used to create contained atomization. Due to restructuring of folders, it is no longer working, but provides an easy to follow outline of how to create atomization. Other than correct paths, the correct alignment wrapper has to be selected in a Makefile.

		filtered	none	low score	low score and shorter than minLen			
		minLen			13	25	63	125
Simulated data set	LAST	count	643	643	643	643	643	629
		median	2272	2272	2272	2272	2272	2339
		mean	6521	6521	6521	6521	6521	6670
	LASTZ	count	502	502	502	502	499	488
		median	2904	2907	2907	2907	2950	3031
		mean	8363	8366	8366	8366	8413	8603
UGT1A	LAST	count	2444	1469	1469	1469	1469	1321
		median	472	744	744	744	744	968
		mean	1089	1461	1461	1461	1461	1613
	LASTZ	count	2300	942	942	942	890	830
		median	764	1132.5	1132.5	1132.5	1336	1832.5
		mean	1502	2535	2535	2535	2680	2868
Plague	LAST	count	153051	131038	131038	131038	131038	118928
		median	375	711	711	711	711	715
		mean	929	1031	1031	1031	1031	1126
	LASTZ	count	283842	161089	161089	161089	149479	125254
		median	166	196	196	196	220	715
		mean	587	864	864	864	927	1088

Table 3.6: Basic statistics of alignments produced by LAST and LASTZ for each data set. Number of alignments, their median and mean length are presented. First two columns contain statistics for all unfiltered alignments, and alignments where only those with low score were filtered out, however no atomization was created from the set of alignments filtered this way. Last four columns correspond to alignments used for atomization with $L = [50, 100, 250, 500]$ as we are filtering alignments shorter than $L/4$. For sample data set, number of alignments is average of all ten included data sets.