Department of Computer Science
Faculty of Mathematics, Physics and Informatics
Comenius University, Bratislava

Atomization of DNA Sequences with Complex Evolutionary History

(Master's Thesis)

Michal Burger

9.2.1 Informatics

Advisor: Mgr. Tomáš Vinař, PhD.

Bratislava, 2010

I hereby declare that I wrote this thesis by myself, only with the help of the referenced literature, under careful supervision of my thesis advisor.

. . . .

Abstract

DNA sequences which have undergone repeated duplications provide a significant challenge to DNA sequence analysis. Such sequences are believed to be a major source of evolutionary innovation, possibly being responsible for the rapid evolution of human species, and are of great interest to molecular biologists. Their analysis is difficult due to the many repeating regions which cause existing techniques to become unusable.

Some of the new approaches to analyzing these sequences rely on finding the atomization of a set of related DNA sequences. Currently, there are no known algorithms that would produce atomizations of required quality and precision. This thesis proposes a new atomization algorithm and attempts to estimate its effectiveness. We have tested our approach both real and simulated data and, the results are compared to an existing atomization algorithm.

Keywords: bioinformatics, DNA sequence analysis, evolution, ancestral reconstruction, simulation.

To Kubus,
so that he remembers.

Contents

1 Introduction **8**
1.1 Structure of a DNA molecule . . . 8
1.2 Evolution of DNA sequences . . . 9
1.3 The atomization problem . . . 10
1.3.1 Intuitive description . . . 12
1.3.2 Formal definition . . . 13

2 Approaching the problem **16**
2.1 Local alignments . . . 16
2.2 The trivial algorithm . . . 17
2.3 Integer programming . . . 19

3 Proposed solution **23**
3.1 Modifying the trivial algorithm . . . 23
3.2 Merging the atomic boundaries . . . 24
3.3 The final algorithm . . . 25

4 Simulating DNA sequences **27**
4.1 Mathematical model of sequence evolution . . . 28
4.1.1 Substitution model . . . 28
4.1.2 Large-scale events . . . 31
4.2 Implementation of SimSeq . . . 32
4.2.1 Generating large-scale events . . . 33
4.2.2 Simulating the substitution model . . . 35
4.3 Inferring the correct atomization . . . 36

5 Results **38**
5.1 Simulated data . . . 38
5.2 Real DNA sequences . . . 39

6 Conclusion **44**

A GNU Free Documentation License **45**

List of Figures

1 The phylogenetic tree of mammalian species 11

2 A global alignment of two sequences 17

3 Local alignments of the IFN region in the Rhesus Macaque and human genomes 22

4 Processing alignments in the trivial algorithm 24

5 Number of correctly identified atoms in simulated sequences . 41

6 Input adherence on real DNA sequences 42

7 Output consistency on real DNA sequences 43

Chapter 1

Introduction

Bioinformatics is a growing field of computer science applied to problems of molecular biology. One of its biggest areas of research is the analysis of biological sequences and in particular the sequences of DNA. Problems in bioinformatics are motivated by complicated molecular processes, therefore, they usually do not resemble classical problems of theoretical computer science, and they require very specialised solutions.

This thesis will discuss the problem of atomization of DNA sequences. An atomization of a DNA sequence is an abstraction used in recent research for finding evolutionary history of related mammalian species. Some authors simply assume the existence of an atomization algorithm (Bertrand et al., 2008; Ma et al., 2008; Zhang et al., 2009), others attempted to provide a trivial implementation (Vinar et al., 2009), but their solution produced poor results. In this work, we look for a better algorithm that could be successfully applied to real-world data.

This chapter will explain the terminology and basic concepts necessary for understanding the problem. It will also formally and informally describe the problem of atomization and mention the motivation behind the problem.

1.1 Structure of a DNA molecule

Cells of all known living organisms contain molecules of DNA which carry their genetic information. DNA molecules are large polymers made of structural units called *nucleotides*. Nucleotides of a DNA molecule are arranged in a sequence, and for this reason the DNA molecules are also called DNA strands. Each nucleotide of a DNA strand contains one of the four DNA bases: adenine, cytosine, guanine or thymine (abbreviated A, C, G and T). The nucleotides are internally non-symmetric, therefore it is possible to tell the orientation of a DNA strand and to distinguish its beginning from its end.

For our purposes, the relevant information contained in the DNA is represented in the order of the bases on DNA strand. We will describe this information by the corresponding DNA sequences – strings of characters from the alphabet $\{A, C, G, T\}$. A *segment* of a DNA sequence is a substring of the sequence. A *site* of a DNA sequence is a specific position in the sequence. For example, we could say that every site of a DNA sequence contains one of the DNA bases. Thus we will call the elements of a DNA sequence sites and the elements of the alphabet $\{A, C, G, T\}$ the bases.

Molecules of DNA are usually found in pairs forming the well-known double helix structure. The double helix consists of two DNA strands of equal lengths running in opposite directions. The strands are aligned so that the i-th nucleotide on one strand is connected by a hydrogen bond to the $(n+1-i)$-th nucleotide on the other strand. The two aligned nucleotides are called a DNA base pair, and they always contain complementary DNA bases – adenine (A) is paired with thymine (T) and cytosine (C) is paired with guanine (G). Two DNA strands that form a double helix are called complementary, and two DNA sequences are complementary if they correspond to complementary DNA strands.

In this work, we will describe species in terms of their genetic information and the correcponding DNA sequences. It should be noted that there is, in fact, no such thing as “the” genetic information of a species because every two individual organisms, even of the same species, usually carry different DNA. However, DNA sequences of organisms from the same species typically differ only in a small fraction of sites. Therefore, we will simply pick one representative of the species and assume that all organisms of the same species carry the same genetic information.

1.2 Evolution of DNA sequences

According to the evolutionary theory, biological species evolve over time and their evolution is caused by mutations of their genetic information. If we had an opportunity to observe a DNA sequence of evolving species, we would see a series of evolutionary events that modify the sequence. There are five basic types of evolutionary events we will take into account when modeling the evolution of DNA sequences.

- *Single-base substitutions.* These modify a single site of the DNA sequence by exchanging the base appearing at this site for a different one.

- *Insertions.* An arbitrary DNA sequence is inserted between two adjacent sites of the original sequence. The resulting sequence will be longer than the original sequence.

- *Deletions.* A contiguous subsequence is deleted from the DNA sequence. The resulting sequence will be shorter than the original sequence.

- *Duplications.* Just like insertions, but the inserted sequence is not arbitrary: it is a copy of an existing DNA sequence, called the *source sequence*. The source sequence can be either a contiguous subsequence of the original DNA sequence, or it can be located on the complementary DNA strand, in which case it is a contiguous subsequence of the complementary DNA sequence. The duplication is then either a *same-strand duplication*, or a *cross-strand duplication*. The molecular mechanism underlying duplications will not allow the copied sequence to be inserted inside the source region (or the corresponding complementary region in case of a cross-strand duplication).

- *Speciations.* These are a bit different from the rest of the evolutionary events in that they do not modify the DNA sequence. A speciation occurs when a single species splits into two species. The sequences of the two new species will evolve independetly, both species starting from the original DNA sequence.

We will refer to the insertions, deletions and duplications as *large-scale evolutionary events.*

There is an evidence which suggests that most of the present-day species evolved from a single common ancestral species by a number of speciations (Darwin, 1859). We could visualise this process by a large binary tree called the phylogenetic tree. In a phylogenetic tree, the present-day species appear in leaves, the ancestral species appears in the root of the tree and every internal node represents a speciation. We can create a phylogenetic tree for any subset of the present-day species in a similar way – Figure 1 provides an example of such phylogenetic tree.

1.3 The atomization problem

Let $S = \{s_1, \ldots, s_n\}$ be a set of DNA sequences which evolved from the same ancestral sequence a. We can define the phylogenetic tree on the set S just like we defined it for species, with sequence a in the root of the

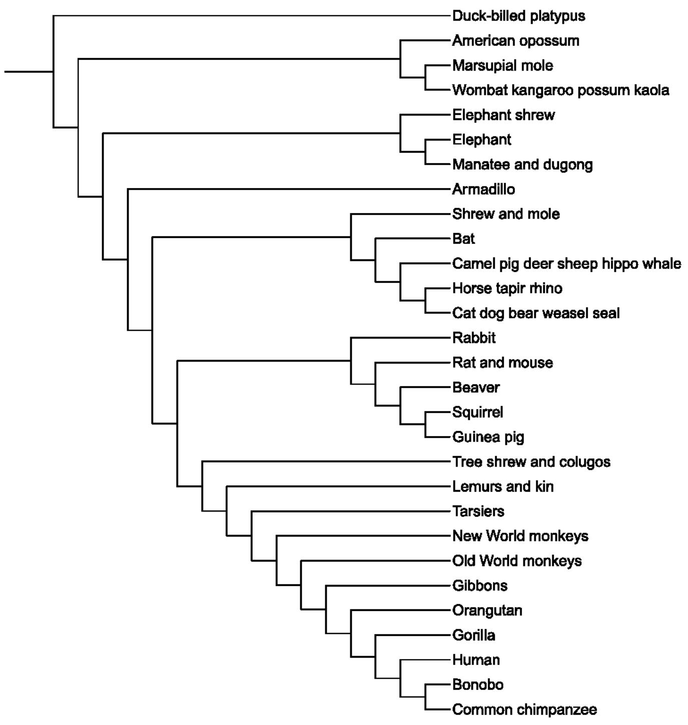

Figure 1: The phylogenetic tree of mammalian species as discussed in Dawkins (2004). Image source: Hsu (2008).

tree, and sequences $s_1, \ldots, s_n$ in its leaves. In this tree, there were $n - 1$ speciations during the evolution from a, each represented by an internal node of the tree, and possibly a number of other evolutionary events. Each of these other evolutionary events must have occured on one of the edges of the phylogenetic tree. The ancestral DNA sequence a, the phylogenetic tree, and the list and order of all evolutionary events on each edge of the tree, constitute the evolutionary history of set S. If we knew this evolutionary history, we could determine what sequences $s_1, \ldots, s_n$ look like.

In a real world, however, we usually only have the DNA sequences of present-day species, and want to find their evolutionary history. Obviously, the sequences of set S do not carry enough information for reconstruction of their complete evolutionary history. For a non-empty set S there is always an infinite number of possible evolutionary histories that all lead to the same set of present-day DNA sequences. But we can at least try to estimate which of the potential histories seem the most probable. Several authors attempted to approach the problem of the evolutionary history reconstruction with the use of atomic segments (Bertrand et al., 2008; Ma et al., 2008; Vinar et al., 2009; Zhang et al., 2009). This section will describe the concept of atomic segments and atomization.

1.3.1 Intuitive description

Reconstruction of evolutionary history is based on one key observation. If we have a set of DNA sequences which are evolutionary not very distant (meaning that the number of events in the evolutionary history is relatively small), the sequences may look similar, or share similar looking regions. This happens because after a speciation, the two resulting sequences are completely identical, and if there is only a small number of other evolutionary events, parts of both sequences may remain unchanged. Of course, it is entirely possible that two identical sequences evolved from two different and completely unrelated ancestral sequences, and they only look the same by pure coincidence. However, this is extremely improbable, and we are willing to make the sacrifice of not reconstructing such histories correctly.

Consider following simple evolutionary histories. If the history only contained speciations, all present-day sequences would look the same. Imagine there is a single deletion on one of the edges of the phylogenetic tree. This deletion splits the DNA sequence into three parts: the deleted region, and the two remaining segments to the left and to the right of the deleted region. If we call these regions A, B, and C then the sequence before the deletion is ABC and the sequence after the deletion is AC. Now, depending on the particular phylogenetic tree, all present-day sequences will be either AC or ABC. Segments A, B, and C are, in this case, what we call the *atomic segments*, or *atoms* and the way we split the present-day sequences into atomic segments is called the *atomization*.

In the trivial example above, all the sequences of S would consist of a single atom. In the example with a single deletion, all sequences would be split into either two or three atoms. Note that the evolutionary events affect the atomization of all sequences, not only the ones that appear in the same subtree as the given event. Each of the large-scale evolutionary events introduces a number of breakpoints, which split any existing atomic segments into smaller ones. For insertions and deletions these are the boundaries of the inserted or deleted regions. For duplications these are also the boundaries of the source region. Substitutions do not affect the atomization.

Consider once more the atomization into atoms AC and ABC. We intuitively understand that atoms denoted by letter A are “the same” in all of the sequences. This will be formalized by the concept of atomic classes. Every atom will belong to an atomic class and atoms of the same class will represent DNA segements which evolved from the same ancestral segment. In our example, there would be three atomic classes: A, B, and C. Duplications make it possible to see multiple atoms of the same class in a single sequence. For example, an evolutionary history with a single duplication and no other large-scale events would imply atomizations $ABCBD$ and $ABCD$, where B is the atomic class of the duplicated segment. A special provision is needed for cross-strand duplications: we will need to define for each atomic class its complementary class. If the duplication in the previous example would have been a cross-strand duplication, the atomization would then be $ABC\overline{B}D$ where B and $\overline{B}$ are complementary atomic classes.

1.3.2 Formal definition

In this subsection, we will give a definition of a correct atomization of a set of sequences S, corresponding to a given evolutionary history H. We start off by defining the atomization in general.

Definition 1.1. *An* atomization *of a set S of DNA sequences consists of:*

- *Partitioning of each element of S into DNA segments called the* atomic segments*.*
- *Partitioning of the set of all atomic segments into $2k$ (possibly empty) sets $C_1, \ldots, C_k, \overline{C_1}, \ldots, \overline{C_k}$.*

Sets $C_1, \ldots, C_k, \overline{C_1}, \ldots, \overline{C_k}$ are the atomic classes*. Classes C_i and $\overline{C_i}$ are called* complementary*. All of the classes $C_1, \ldots, C_n$ must be non-empty while classes $\overline{C_1}, \ldots, \overline{C_n}$ can be potentially empty.*

A partitioning of a DNA sequence into atomic segements means splitting the sequence into substrings which can be concatenated into the original sequence. We will call the place where one atom ends and another one begins (and also the beginning and the end of the sequence) an *atomic boundary* or a *breakpoint*.

If we know the evolutionary history H of a set S, we also know the ancestral sequence a, and we can reconstruct all DNA sequences that appeared during the evolution of a into sequences $s_1, \ldots, s_n$. We will call these sequences the *intermediate sequences* of evolutionary history H. The sequences $s_1, \ldots, s_n$ and the ancestral sequence a are also considered to be intermediate sequences of H. The easiest way to define the correct atomization corresponding to an evolutionary history H is to extend the atomization to all intermediate sequences of H.

Definition 1.2. *An atomization of the set of intermediate sequences of an evolutionary history H is a* correct atomization of intermediate sequences of H *if all of the following conditions are met:*

- *For each speciation, the two resulting intermediate sequences have exactly the same atomization as the original sequence just before the speciation occured.*
- *For each deletion, the boundaries of the region in the original sequence which is about to be deleted are also atomic boundaries. The resulting sequence must have the same atomization as the original sequence, except for the atoms between the two boundaries which should not appear in the resulting sequence.*
- *For each insertion, the boundaries of the inserted region in the resulting sequence are also atomic boundaries. The original sequence must have the same atomization as the resulting sequence except for the atoms between the two boundaries which should not appear in the original sequence.*
- *For each duplication, the same conditions must hold as in the case of an insertion. Additionally, the boundaries of the source region must be also atomic boundaries, and the number of atoms in the source region must be the same as the number of atoms in the inserted region. For same-strand duplications, the atomic classes of atoms in the source region must be the same as the atomic classes of atoms in the inserted region in the same order. For cross-strand duplications, the atomic classes of atoms in the source region must be complementary to the atomic classes of atoms in the inserted region in reverse order.*
- *For each single-base substitution, the atomizations of the original and the resulting intermediate sequence must be the same.*
- *This atomization is minimal.*

The condition of minimality means that there exists no atomization which satisfies all the other conditions, and contains smaller total number of atoms. If there are multiple atomizations with this property, minimal is the one with the highest number of atomic classes. Note that the condition of minimality does not guarantee uniqueness of correct atomization.

Definition 1.3. *An atomization A of a set of sequences S is a* correct atomization corresponding to an evolutionary history H *if there exists a correct atomization of the intermediate sequences of H which segments the sequences of S the same way as A does.*

In the last, definition we say that two atomizations segment a set of sequences in the same way. This means that each sequence is in both atomizations partitioned into the same atoms, and every pair of atoms which belong to the same or complementary atomic classes in one atomization also belong to the same or complementary atomic classes in the other atomization.

In real evolutionary histories, we expect to see many large-scale events that are in fact very short, mainly insertions and deletions. We do not want to consider all of them in the correct atomization because the atomization then becomes cluttered with tiny atoms that are not very useful for further processing. For example we could decide that we will consider every insertion and deletion which is less than 10 base pairs long as too short. We can modify the definition of correct atomization to distinguish between long and short events. The long events must satisfy the conditions of the original definition and the short events will have a new set of conditions:

- For each short deletion, the original and resulting sequences must have the same atomization except for atoms overlapping the deleted region. These will be shorter or completely missing from the resulting sequence due to some of their sites being deleted.

- For each short insertion and duplication, the original and resulting sequences must have the same atomization except for atoms overlapping the inserted region. These will be shorter or completely missing from the original sequence due to some of their sites not being inserted into the sequence yet.

The original definition of correct atomization is then a special case of the modified definition with all evolutionary events considered long. It is not clear whether a correct atomization must always exist. We will show in Chapter 4 an algorithm for finding a correct atomization given an evolutionary history which will imply that at least one correct atomization must always exist.

Chapter 2

Approaching the problem

Our objective is to find an algorithm that would produce a correct atomization for a given set of sequences with an unknown evolutionary history. As we have discussed in the previous chapter, this is, strictly speaking, impossible to achieve. Almost any atomization could be a correct one for some very improbable evolutionary history. We will use a parsimony approach to decide which atomization to choose. An atomization which explains the similarities between input sequences in a simple way using a small number of evolutionary events is more probable than a complicated atomization.

We will base our solution on the fact, that the atoms of the same atomic class should have originated from the same ancestral region. Thus, we expect that two atoms of the same class should have similar sequence of bases. The only differences will be due to single-base substitutions and short large-scale events occuring in the course of their evolution. If the input DNA sequences are not too evolutionarily distant, we should still be able to recognize regions of the same atomic class.

2.1 Local alignments

In bioinformatics, the concept of "similarity" of DNA sequences is formalized by the notion of global and local alignments. A *global alignment* of two sequences is a way of arranging the elements of the sequences to match each other in the best possible way. Take a look at Figure 2 for an example of a global alignment. Gaps represented by dashes are inserted into one or both sequences so that they have the same length. The gaps are inserted in such way that as many columns as possible contain pairs of identical bases. More generally, every possible combination of bases or gaps is assigned a score and the highest scoring alignment is chosen. Simple scoring schemes could assign

```
--GGCCATAATGAC
TAGTCCAT---CAC
```

Figure 2: A global alignment of sequences GGCCATAATGAC and TAGTCCATCAC. Dashes represent gaps inserted into the sequences. This alignment has seven pairs of matching bases, two pairs of mismatched bases, and five gaps.

positive values to matching bases and negative values to bases that differ or are matched with gaps. An efficient dynamic programming algorithm exists for finding the best global alignment given any such scoring scheme (Needleman and Wunsch, 1970). The scoring does not necessarily have to be defined per pairs of bases. We could use any scoring function which assigns a score to two aligned sequences. However, not for every such function will exist an efficient global alignment algorithm.

A *local alignment* of two sequences A, B is a global alignment of any two sequences a, b which are substrings of A and B respectively. We could theoretically align any two substrings of A and B and call them a local alignment, but usually we are only interested in high scoring alignments. This way, we filter out sequences that are too different or only similar by coincidence, and keep sequences that are likely evolutionary related. We could also decide to look for maximal alignments – ones that cannot be extended or shortened to obtain a better scoring alignment. An efficient dynamic programming algorithm for finding such local alignments is described by Smith and Waterman (1981).

Unfortunately, the dynamic programming algorithm is quadratic in length of the input sequences. Since DNA sequences are usually very long, this algorithm can not be used in practice. Modern tools for finding local alignments use a combination of the dynamic programming algorithm and heuristic methods (Altschul et al., 1990). In this work, we will use a popular tool for finding local alignments, LASTZ (Harris, 2010), which is an improved version of the program BLASTZ (Schwartz et al., 2003). This program also searches the complementary sequences to identify local alignments between complementary DNA strands. We will need this feature to identify atoms of complementary classes. See Figure 3 for an example of the output of program LASTZ.

2.2 The trivial algorithm

We can now formalize our assumption about similarity of atoms of the same atomic class: every pair of atoms belonging to the same or complementary atomic classes should be classified by LASTZ as a significant local alignment. To find atoms of the same class, we would run LASTZ for every pair of the input sequences, and examine the results. But since LASTZ only looks for maximal alignments, our assumption does not imply that every such pair of atoms will be present in the output of LASTZ. We should rather expect to see segments with the same atomizations. For example, if we tried to align two sequences with correct atomizations $ABCDE$ and $FBCDG$, LASTZ would find the local alignment of the two BCD segments rather than the idividual alignments of Bs, Cs, and Ds.

We will also assume that all produced alignments start and end at atomic boundaries. If this condition does not hold, the alignments are not be maximal. Using the same example, an alignment that would start in the middle of atom B could be extended to the left so that the boundary coincides with the start of the atom B. This will cause more matching bases to appear in the alignment and the alignment will have a higher score. On the other hand, suppose an alignment starts in the middle of the atom A in the first sequence and in the middle of the atom F in the second sequence. We could shorten this alignment so that its boundary coincides with the beginning of the atom B. This way we will lower the number of mismatched bases and should again obtain a better score.

We can now describe the trivial algorithm which relies on these assumptions. At first, all the input sequences are considered to consist of a single atom, each being of a different atomic class. The algorithm processes the local alignments one by one. For each alignment, it makes sure that all conditions we have described are fulfilled. First condition states, that the beginning and end of the alignment in both sequences is at an atom boundary. If this is not the case, one of the ends falls inside of an atom. We split the atom into two new atoms together with all other atoms of the same or complementary atomic classes. Second condition states that the two aligned regions must be atomized in the same way (or complementary ways in the case of cross-strand alignments). The algorithm first checks if all atomic boundaries in one region correspond to atomic boundaries in the other region and creates new atomic boundaries if necessary. This is again achieved by splitting atoms. Finally, we check whether the aligned atoms belong to the same atomic classes. If they are from different classes, the algorithm merges these two classes.

After processing all the alignments, we will end up with an atomization that satisfies all aforementioned assumptions. We will not go into technical details of the various parts of the algorithm or prove its correctness. This is because the algorithm, in fact, does not work for real DNA sequences. After running it on several sequences with rich duplication history, we have found out that the resulting atomization would consist of atoms of average length of approximately four bases. Such atomization is useless for most purposes. This is exactly the reason why we introduced the modified definition of correct atomization: to remove tiny atoms that are unimportant for further processing of the sequences.

The reason, why the trivial algorithm produces such poor results, is simple. LASTZ can not find the exact boundaries of all atoms using only the scoring function. The alignment algorithm looks for places in the sequence where the scoring function attains a local maximum. But chances that this is *exactly* at a boundary of an evolutionary event are low. The exact information about event boundaries is simply not present in the sequence.

Consider the following example. There are three atoms a, b, c of the same atomic class, and LASTZ correctly aligns a to b, b to c, but in the alignment of a to c incorrectly adds one more site to the end of both aligned sequences. The algorithm will split the atom right after a into a single-base atom and the rest of the original atom. But not only this one atom – it will split all atoms of the same atomic class. And it will also split the atom next to c and all atoms of the same atomic class.

An even worse situation would occur, if the alignment of a to c correctly found a but incorrectly shifted c by one site, for example to the right. This would cause all three atoms to be entirely split into single-base atoms, only because a single local alignment was off by one site. But in reality, LASTZ will almost never find two alignments that would share the same boundary. It could be wrong by a couple of bases in every alignment, or even produce alignments that, in fact, do not correspond to atoms of the same class at all. We will have to allow our algorithm to tolerate mistakes such as these.

2.3 Integer programming

A method that can easily deal with approximate data to produce approximate results is linear optimization, or linear programming. As we will shortly see, we can formulate the problem of atomization as an integer programming problem. This means, we can transform our problem into a problem of maximizing a certain linear function of integer variables under the conditions given by a set of linear inequalities of the same variables.

To start with, we can look at LASTZ alignments in a slightly different way: they provide the information for every pair of sites of the input sequences and their complementary sequences, whether these two sites presumably originated from the same ancestral site, or not. In reality, one would expect this relation to be transitive; however, due to possible mistakes in the local alignments there will be relations between some sites missing, and there will also be spurious relations. Under the pressure of such mistakes, the trivial algorithm will fail.

Thus our task will be to find a small modification of the original relation produced by LASTZ so that it is transitive. Such modified relation will then imply the atomization of the input sequences.

We will solve this problem by integer programming. The integer program will contain N^2 binary variables x_{ij} for $1 \leq i, j \leq N$, where $x_{ij} = 1$ means that sites i and j should be aligned in the modified relation. The necessary linear inequalities are these:

$$0 \leq x_{ij} \leq 1$$

$$x_{ij} + x_{jk} \leq x_{ik} + 1$$

The first condition states, that x_{ij} are binary variables. The second condition ensures transitivity. We want to maximize the number of elements of the relation that remained unchanged with respect to the original relation implied by the LASTZ alignments. To make sure that we do not again end up with millions of tiny atoms, we can also try to minimize the number of atomic boundaries. To accomplish this, we will add $N - 1$ more binary variables z_i for $1 \leq i < N$. Value $z_i = 1$ means, that there is an atomic boundary at position i. The corresponding inequalities are:

$$0 \leq z_i \leq 1$$

$$x_{(i+1)(j+1)} - x_{ij} \leq z_i$$
$$x_{ij} - x_{(i+1)(j+1)} \leq z_i$$

The first inequality again states, that z_i is a binary variable. The other two inequalities force $z_i = 1$ if the site on one side of the boundary is aligned to a site near boundary j, but the site on the other side of the boundary is not aligned to the corresponding other site near boundary j. This means that the atom either ends at this position, or there is a gap in the alignment. This version, therefore, does not properly count atoms with gaps. The objective function which we want to maximize looks like this:

$$p\left(\sum_{r_{ij}=1} x_{ij} + \sum_{r_{ij}=0} (1 - x_{ij})\right) - q\sum z_i$$

Numbers p and q are constants which we can choose to balance the number of atoms versus the resemblance to the original relation, r_{ij} is the original relation itself.

But again, there is a fundamental problem with this approach. The number of inequalities that specify transitivity is N^3. Even for the smallest sequences, N is at least 10^5, which means we have at least 10^{15} inequalities. Even if each inequality could be represented by a single byte, this would be still far too much to hold in a computer memory. One way we could improve this situation would be by lowering N. This can be done by grouping adjacent sites into longer segments and treating these segments as if they were just individual sites. Variables x_{ij} would then indicate, whether the segments i and j are aligned or not. The summands in the objective function would be weighted according to the lengths of the segments. With this modification, we are not losing any information, and we can decrease N by any factor we like. The only condition is, that we must be able to tell which segments were aligned in the original relation r_{ij}. This is, in fact, a very restrictive condition. To split the sequences into segments satisfying this condition, we would use an algorithm very similar to the trivial algorithm from the previous section. And, as we already know, this will lower N only by a very small factor.

As an interesting side note, the integer program with grouping was first tested without us realizing, that the grouping algorithm produces the same results as the trivial atomization algorithm. We used data generated by the simulator used by Vinar et al. (2009) and aligned the data with LASTZ. We wrote a simple program to generate the inequalities and tried to solve the integer program using the integer program solver CPLEX (IBM, 2010). The number of generated inequalities for some of the test cases was surprisingly low, and we were able to find a solution. For input sequences with N between 10^5 and $3 \cdot 10^5$, the number of generated inequalities was between 10^6 and 10^9 and the smallest cases could be solved in a few seconds.

However, when testing on real data, we encountered the same problems as described in the previous section with the trivial algorithm. As we later found out, the simulator we used was missing one important feature: simulating short insertions and deletions, which would introduce gaps in the local alignments. We had decided to not use this simulator anymore. But since solving the integer problem took is some cases relatively low time, using integer programming still seems to be a viable option. A possible improvement could be to specify only a random subset of all the inequalities and hope, that the solver will find a solution satisfying all of the conditions. If it does not, more conditions would be added to the input.

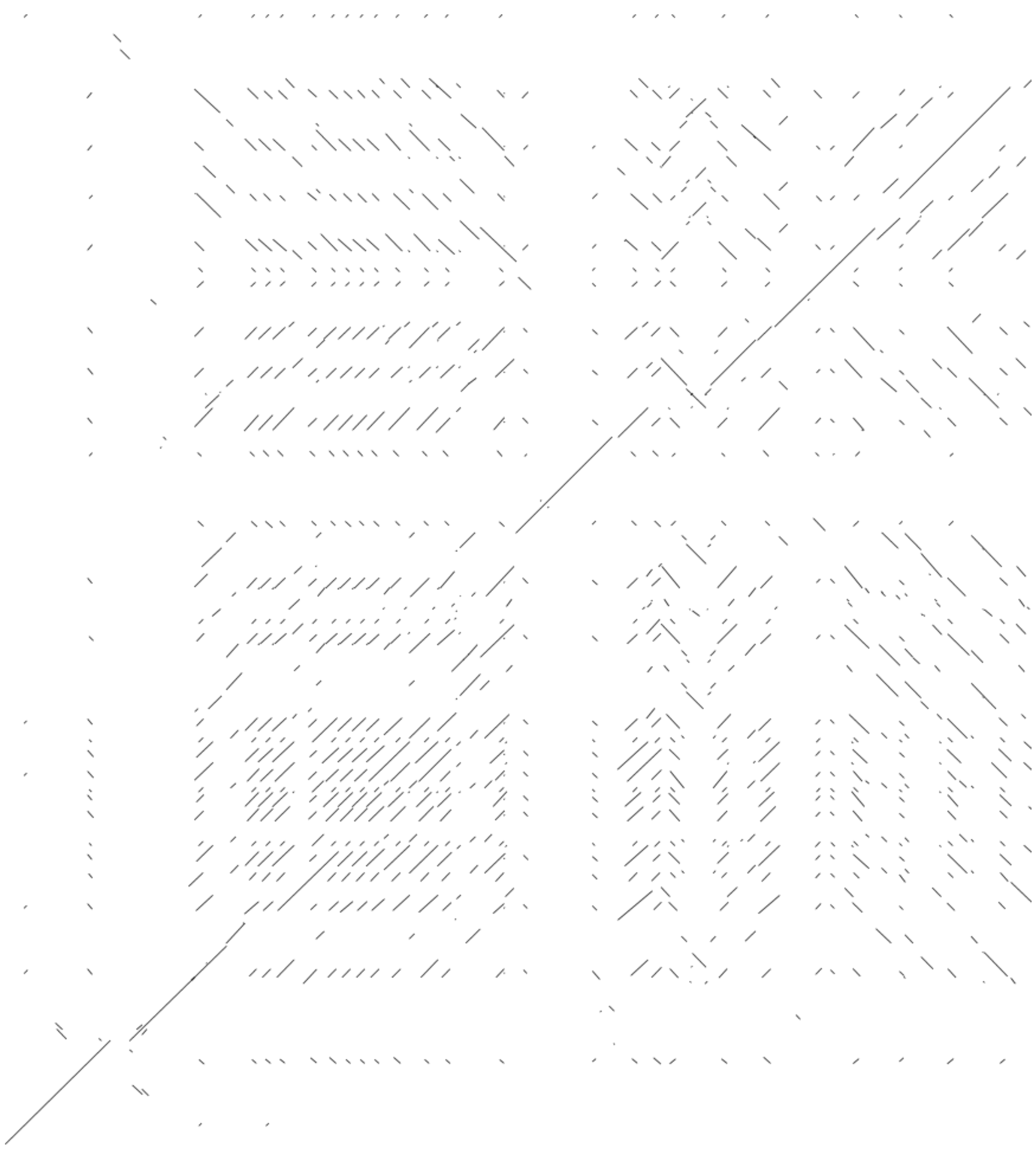

Figure 3: A dot plot visualisation of the local alignments found by LASTZ in the IFN regions of Rhesus Macaque and human genomes. The x-axis corresponds to a 183895 sites long part of the IFN region on chromosome 2 in the Rhesus Macaque genome. The y-axis is a 211954 sites long part of the IFN region on chromosome 18 in the human genome. The lines mark local alignments, lines with inverted slope corresponding to alignments on the complementary strand.

Chapter 3

Proposed solution

The trivial algorithm mentioned in the previous chapter struggles with one problem. It is the fact, that alignments produced by LASTZ do not end precisely at the correct atomic boundaries. In this chapter, we will describe an improvement to the trivial algorithm which circumvents this problem.

The idea is to find atomic boundaries which appear very close to each other and merge them. Such boundaries are likely the result of multiple LASTZ alignments identifying the same atomic boundary in the correct atomization, but with small errors.

3.1 Modifying the trivial algorithm

Firstly, we need to change the order in which we process the alignments. The trivial algorithm takes alignments one by one. For each alignment, it creates new atomic boundaries at the beginning and at the end of both aligned regions, and it copies atomic boundaries that do not have a matching pair (see Figure 4). Every new atomic boundary splits an existing atom into two. The trivial algorithm makes sure that all the other atoms of the same atomic class are also split, at the same position.

At the start of the algorithm, all atoms are of different atomic classes. Two atoms become atoms of the same atomic class only when we merge atomic classes, and this happens only when two atoms of different atomic classes are aligned to each other by one of the input alignments. Therefore, two atoms a, b are at the end in the same atomic class if and only if there exists a sequence of atoms $a, a_1, \ldots, a_n, b$ such that a is aligned to a_1, a_1 is aligned to a_2, etc. until finally, a_n is aligned to b by the input alignments.

The atomic boundaries and atomic classes, which the trivial algorithm keeps track of, are, in fact, a sort of transitive closure of the input alignments.

```
Before:   ...gcccggCCCGACGAGAAG|GCCGACtcagga...
          ...gagcatCC|GGACGAGAAGTCCCACagca|aa...

After:    ...gcccgg|CC|CGACGAGAAG|GCCGAC|tcagga...
          ...gagcat|CC|GGACGAGAAG|TCCCAC|agca|aa...
```

Figure 4: When processing the alignments in the trivial algorithm, every atomic boundary is copied from one aligned sequence to the other, and new atomic boundaries are created at the beginning and at the end of the alignment. Uppercase characters denote the aligned regions, lowercase characters are the rest of the sequence. Vertical lines show the locations of atomic boundaries.

When the trivial algorithm processes a new local alignment, it translates this alignment into a set of individual alignments between atoms and updates the transitive closure by merging any newly aligned atomic classes. But the same effect can be also achieved in a different way. We could first read all the alignments without updating the transitive closure, and then start constructing the closure step by step.

The modified algorithm then works like this. At the beginning, it reads all the input alignments and creates atomic boundaries at the beginning and at the end of each alignment. It does not keep track of atomic classes, since the concept of an atomic class makes no sense without transitivity. The algorithm then starts applying transitive steps, which consist of copying atomic boundaries just like in the trivial algorithm but without splitting the whole atomic class. When the transitive steps stop producing new atomic boundaries, the transitive closure has been completed. We still need to find out which atoms are in the same atomic class, but this can be easily done by a depth first search.

3.2 Merging the atomic boundaries

We can now introduce the promised merging of atomic boundaries which appear too close to each other. We do not want to start merging the boundaries after the transitive closure is complete, because at this point we already have an average atom length of four base pairs, so the atomic boundaries are practically everywhere. Instead, we will merge the boundaries after each transitive step.

The process of merging atomic boundaries will be parametrized by an integer value m, which is the smallest distance at which we no longer consider two boundaries to be “too close” to each other. The problem can be formally stated as follows. For a given set of atomic boundaries B, we want to find the set of atomic boundaries B' such that:

- Every two atomic boundaries in B' are at least m sites apart.
- The sum $\sum\limits_{b \in B} dist(b)$ is minimal, where $dist(b)$ is the distance of b to the closest element of B'.

The second condition means that we want to move the elements of B as little as possible.

This problem can be solved using dynamic programming. For each possible atomic boundary i, we will calculate the values v_i and p_i. The value of v_i is the minimal possible sum of $dist(b)$ if the last element of B' was at position i. The value of p_i is the position of second to last element in the set B' attaining this minimum. We can calculate the values of v_i and p_i using the values of v_j and p_j for $j < i$.

The base case is simple. For $i < m$, the values p_i are undefined, since no two elements of B' can be less than m positions apart. The values of v_i can be easily calculated, because we know that there is only a single element in the set B'.

In the case of $i \geq m$, we will consider all possible values of p_i, which are $0, 1, \ldots, i - m$, or undefined, meaning that there is only one element in B'. But in fact, we may assume that $i - 2m < p_i \leq i - m$. If this were not so, we could add another element to B' at position $i - m$, and this element would not worsen our solution. For each of these m potential values of p_i, we will calculate v_i using the value of v_{p_i} and choose the best one.

In the end, we will find i for which v_i is the smallest. The elements of B' will then be $\{i, p_i, p_{p_i}, \ldots\}$. It is possible to find the values of v_i and p_i for single $i \geq m$ in $O(m)$ time complexity, therefore the time complexity of the whole algorithm is $O(mN)$ where N is the length of the sequence.

3.3 The final algorithm

When we put all of this together, the complete algorithm works as follows. At the beginning, it finds local alignments of the input sequences using LASTZ. Then, it creates the initial atomic boundaries at positions, where the local alignments start and end. These boundaries are merged and the algorithm starts alternating between transitive steps and merging of the boundaries. It stops when, after merging the boundaries, we obtain a set B' which we have already seen before. For this to be possible, the algorithm keeps all previous sets B' stored in a hash table. Since there is only a finite number of different possible sets B', this part of the algorithm is guaranteed to stop. However, the upper bound on its running time is $O(mN2^N)$ and the upper bound on its space complexity is $O(N2^N)$.

After establishing the atomic boundaries, we still need to assign the atoms to the atomic classes. To accomplish this, we will examine which pairs of atoms are aligned by the original local alignments and construct a transitive closure. But after merging of the atomic boundaries, many of the boundaries are at different places than at the beginning, and it may not be entirely obvious which pairs of atoms are aligned. For this reason, we also need to keep for each atom the information about which alignments it originally belonged to.

Even though the upper bounds on the time and space complexity of the algorithm are exponential, it does not necessarily mean that it will not perform well. In practice, it may turn out that the number of iterations necessary to find the atomic boundaries will be much lower. We will see how well the algorithm performs on real data in Chapter 5.

Chapter 4

Simulating DNA sequences

One significant problem arises when testing the atomization algorithm. In order to evaluate the quality of the produced atomizations, we must be able to compare it to something, ideally to the correct atomization of the input data set. Unfortunately, there are no real DNA sequences with known evolutionary history or correct atomization. Luckily for us, the evolution of DNA sequences, as described in Chapter 1, is a fairly well understood process. This makes it possible to simulate evolution of DNA sequences and from the resulting evolutionary history infer the correct atomization.

Simulating the evolution of various biological sequences is not a new idea. There is a large number of existing tools used specifically for simulating DNA sequences including Dawg (Cartwright, 2005), INDELible (Fletcher and Yang, 2009) or indel-Seq-Gen (Strope et al., 2009). However, all of these tools are lacking the ability to simulate duplication events. The only available simulator that can simulate duplications is the one mentioned in Vinar et al. (2009), but as we have found out in Chapter 2, sequences generated by this simulator do not give a good estimate of the quality of the atomization algorithms. Since all of the aforementioned tools are either too complex to be easily modified, or have some other serious flaws, we have decided to create a new DNA sequence simulator better suited for our needs.

This chapter describes our new simulation program SimSeq. Although it is missing many useful features found in other DNA sequence simulators, it has all the functionality neccessary for testing atomization algorithms. SimSeq outputs a set of generated DNA sequences along with their complete evolutionary history. This history can be then used to infer the correct atomization as described at the end of this chapter.

4.1 Mathematical model of sequence evolution

We have already discussed the basics of DNA sequence evolution in Chapter 1. An evolving DNA sequence undergoes a series of evolutionary events which can be of three types: speciations, single-base substitutions, and large-scale events that include insertions, deletions and duplications. To be able to simulate these events, we must first formally describe how each of these events modifies the sequence and at what times these events occur.

When talking about time, we are referring to the concept of evolutionary time. Not all DNA sequences necessarily evolve at the same pace, and individual sequences may even evolve more rapidly at some points in their lifetime than at the others. For this reason, it is more practical to measure evolutionary time using the expected number of evolutionary events. In particular, we will define a unit of evolutionary time to be a length of time during which we expect one substitution per site in our sequence. As we will shortly see in the subsection describing substitution models, this is a well-defined length of time since we will only consider models, where substitutions occur at the same rate at every site in a sequence.

The easiest of the three types of evolutionary events to model are the speciations. During a speciation, the current sequence is replaced by two identical copies which then both continue to evolve independently. In our model, speciations do not occur randomly. Instead, the exact times and order of speciations (or equivalently, the complete phylogenetic tree) must be given as a parameter to the evolutionary model. The other two types of events are more complicated and we will discussed them separately.

4.1.1 Substitution model

Description of the most commonly used models for DNA single-base substitutions can be found for example in Felsenstein (2004). In this section we will summarize the common mathematical properties of these models and describe the generalized time-reversible model (Tavaré, 1986) used in SimSeq.

We will assume that substitutions happen at every site of the sequence independetly and at the same rate. This means that for any given time interval, the expected number of substitutions is the same at each site of the sequence. Models that satisfy this condition are called homogeneous and, in fact, do not model reality very accurately. This is because real DNA sequences contain coding and non-coding regions which behave differently when it comes to substitutions. Coding regions encode genetic functionality, for example they could describe how to create a particular protein. A substitution in such region would most likely cause the region to lose its functionality and put its bearer at evolutionary disadvantage, therefore we are less likely to observe such substitution in evolutionary histories of species. On the other hand, non-coding regions are regions which are not known to have any such functionality, and substitutions can occur without any adverse effects. These regions are said to evolve neutrally. Simple heterogeneous substitution models account for coding regions by scaling down their substitution rate in these regions by certain factor. The more complex models use context-sensitive information and do not model substitutions independetly.

When we observe a single site in a DNA sequence during its evolution, we see at every point of time one of the four bases. Let $X(t)$ be the base observed at time t. Value of $X(t)$ changes only when a substitution occurs. To simulate these substitutions, we will model $X(t)$ as a stochastic process. We need to be able to describe, for any two times $0 \leq a < b$, the value of $X(b)$ if we know the values of $X(t)$ for all $t \in [0, a]$. Here, a is the current simulation time, and b is the time to which we want to advance in the simulation. Since this is a stochastic process, in fact we want to find the probabilistic distribution of the random variable $X(b)$. It can be intuitively seen that the distribution of $X(b)$ will only depend on the value of $X(a)$ and not on the previous values $X(t)$ for $t < a$. This is beacuse a neutrally evolving DNA sequence does not carry any information about previous bases that appeared at this site. Such property is called *memorylessness* or the *Markov property* and it can be expressed as:

$$Pr(X(b) = j | (\forall t \in [0, a]) X(t) = i_t) = Pr(X(b) = j | X(a) = i_a)$$

Our process is therefore a continuous-time Markov process.

Because of how we have defined evolutionary time, it is also reasonable to assume that the process $X(t)$ is time-homogeneous. This means that the way in which substitutions occur does not change with time. Formally written, there exists a function P_{ij} such that

$$Pr(X(b) = j | X(a) = i) = P_{ij}(b - a)$$

The homogeneity is due to the fact that this conditional probability only depends on the difference $b - a$ and not on the actual times a and b. $P_{ij}(t)$ are elements of the transition matrix $P(t)$. It can be shown that $P(x) \cdot P(y) = P(x + y)$ for all positive x, y which in turn implies that $P(t)$ must be of the following form:

$$P(t) = e^{Qt}$$

for some matrix Q. This matrix can always be chosen so that its rows sum to zero and scaled so that the expected number of substitutions is indeed equal to one per unit of time. Matrix Q is then called the *instantaneous rate matrix*, because its elements q_{ij} describe the instantaneous transition rates:

$$\frac{\partial}{\partial t} Pr(X(t) = j) = \sum_i q_{ij} Pr(X(t) = i) \tag{4.1}$$

Once we have chosen a matrix Q which satisfies these conditions, we are ready to simulate process X_Q defined by Q. The last thing we need to know is the initial distribution of $X(0)$. If we denote $\pi_i = Pr(X(0) = i)$ then the vector Π together with matrix Q completely describe our process.

Vector Π is the vector of base frequencies at time zero. As the sequence evolves, probabilities $Pr(X(t) = i)$ converge to some stationary base frequencies of matrix Q. The *generalized time-reversible model* (GTR) puts one more constraint on process X to simplify its simulation: if we look at process X with time flowing in the opposite direction, the observed process will be indistinguishable from process X. For a time-homogeneous Markov process this is equivalent to

$$Pr(X(a) = i) = Pr(X(b) = i)$$

for every two times a, b or that Π is equal to the vector of stationary base frequencies. It follows from (4.1) that $\Pi Q = 0$.

If we take all these restrictions into account, we can write the rate matrix Q for a generalized time-reversible model as

$$Q = \begin{pmatrix} * & x_1 & x_2 & x_3 \\ \frac{\pi_A x_1}{\pi_C} & * & x_4 & x_5 \\ \frac{\pi_A x_2}{\pi_G} & \frac{\pi_C x_4}{\pi_G} & * & x_6 \\ \frac{\pi_A x_3}{\pi_T} & \frac{\pi_C x_5}{\pi_T} & \frac{\pi_G x_6}{\pi_T} & * \end{pmatrix}$$

with elements on the diagonal chosen such that the rows sum to zero. Values of $x_1, \ldots, x_6$ and vector Π are parameters of this model. Elements of Π should be non-negative and sum to one so that they express a probability distribution. Parameters $x_1, \ldots, x_6$ should be non-negative and satisfy

$$2(\pi_A x_1 + \pi_A x_2 + \pi_A x_3 + \pi_C x_4 + \pi_C x_5 + \pi_G x_6) = 1$$

to ensure that the expected number of substitutions is one per site per unit of time.

Many earlier substitution models are just special cases of GTR. For example, the Jukes-Cantor model (Jukes and Cantor, 1969) is a special case with $\pi_A = \pi_C = \pi_G = \pi_T = \frac{1}{4}$ and $x_1 = x_2 = x_3 = x_4 = x_5 = x_6 = \frac{1}{3}$. The commonly used HKY85 (Hasegawa et al., 1985) allows any vector of stationary frequencies Π and its rate matrix is

$$Q = \frac{1}{r} \begin{pmatrix} * & \pi_C & \kappa\pi_G & \pi_T \\ \pi_A & * & \pi_G & \kappa\pi_T \\ \kappa\pi_A & \pi_C & * & \pi_T \\ \pi_A & \kappa\pi_C & \pi_G & * \end{pmatrix}$$

where $r = 2(\pi_A\pi_C + \kappa\pi_A\pi_G + \pi_A\pi_T + \pi_C\pi_G + \kappa\pi_C\pi_T + \pi_G\pi_T)$ is the normalizing constant and κ a parameter of the model describing the relative frequency of transitions and transversions. There are many other models which are special cases of GTR and they usually differ by the number of parameters that we need to supply. They provide different tradeoffs between the accuracy of the simulation and the number of parameters that must be estimated before we can run the simulation.

4.1.2 Large-scale events

From Chapter 1 we already know how insertions, deletions and duplications modify a DNA sequence. We will now discuss at what times these events happen and with what parameters. The first assumption we will make to simplify our model is that the sequence we are simulating is only a short continuous subsequence of a much longer DNA sequence. In fact, we will assume that this supersequence stretches to infinity in both directions. We will usually only want to simulate regions which are a couple hundred thousand bases long, but appear inside a chromosome about hundred million bases long, so this assumption is not completely unrealistic.

A *site boundary* is a place between two adjacent sites in the DNA sequence. We will define for each evolutionary event its location as a unique site boundary in the DNA sequence just before the event occurs. The location of an insertion is simply the site boundary at which the new sequence will be inserted. The location of a duplication is the site boundary at which the duplicated sequence will be inserted. The location of a deletion is the site boundary just before the first deleted base. Note that the large-scale events may also occur outside of the modeled sequence. Insertions and duplications located outside of our sequence will not affect the sequence. Deletions, however, can change the modeled sequence, even if they are located somewhere before the start of the sequence but are long enough to overlap with it.

Just like in the previous section, we will now look at a single site boundary and model all events located there. Once again, we will assume memorylessness and time-homogeneity. This implies that evolutionary events will occur at this site boundary according to a homogeneous Poisson process. The times between two consequent events are given by an exponential distribution. The only parameter to this model is the rate at which events occur, and we will assume that this rate is the same at each site boundary. If we want to have different rates for different types of events, we can model them separately, each with an independent Poisson process. These Poisson processes fully describe the times at which events occur at each site boundary.

We also need to model the parameters of the events. All three types of events have one common parameter, which is the length. We will assume that the distribution of lengths is for every type of event independent of the time and location of the event. This can be any discrete probability distribution of positive values. Fletcher and Yang (2009) mention that a good estimate is the *Zipfian distribution* defined by

$$Pr(Z = u) = \frac{u^{-a}}{\zeta(a)}$$

where $a > 1$ is a parameter of the distribution and $\zeta(a) = \sum_{k=1}^{\infty} k^{-a}$ is the Riemann Zeta function.

Deletions are fully described by their length and location. For insertions, we must also know the elements of the inserted sequence. In our model these will be random bases generated at the stationary base frequencies of the substitution model. Finally, duplications require two more parameters: the source region and a boolean value describing, whether it is a same-strand or cross-strand duplication. The source region can be described by its distance to the location of the duplication and the direction in which it lies. We will assume that both directions are always equally likely and that the distance is independent of all other parameters of the duplication. This can be any discrete probability distribution of non-negative values. The probability of cross-strand duplication is also assumed to be independent of other duplication parameters and is modeled by a Bernoulli trial.

4.2 Implementation of SimSeq

SimSeq is a simple tool developed for a single purpose. Most of the used algorithms are very straightforward and not necessarily optimized for memory or speed. But since the running time and memory usage of the program are relatively low, there was no need to look for a better solution. In this section, we will look at the basic algorithm and explore the details of some of the interesting parts of the program.

SimSeq expects on the input the description of the phylogenetic tree, the initial length of the simulated region, and all the previously discussed parameters of the evolutionary model. The simulation starts by generating the ancestral DNA sequence of given length with bases generated randomly according to the stationary base frequencies of the substitution model. The sequence is stored in a simple byte array. This representation is not very efficient, because we will need to simulate insertions and deletions in the sequence, and therefore shift big parts of the array every time any of these events occurs. This choice of data structure probably accounts for a large part of the running time of SimSeq.

The algorithm then starts traversing the branches of the phylogenetic tree using a depth first search beginning at the root. In every internal node of the tree occurs a speciation which is implemented by simply making a copy of the current sequence. When traversing a branch of the tree the current sequence is evolved for an amount of evolutionary time equal to the length of the branch. The large-scale evolutionary events are simulated by inserting or deleting a part of the array and shifting the rest of the elements appropriately. Substitutions are implemented by simply overwriting the corresponding array element by the new base.

Time and space complexity of this algorithm depends on how much the sequence grows during the simulation. Upper bounds can be given in terms of the greatest length the sequence ever reaches during the program execution. Time complexity is then $O(N(E+T))$ and space complexity is $O(ND)$ where N is the maximum length of the sequence, E is the number of simulated evolutionary events, T is the number of nodes in the tree and D is the depth of the phylogenetic tree.

4.2.1 Generating large-scale events

The input of SimSeq can describe any number of large-scale event sources. An event source describes a single type of evolutionary events (either insertions, deletions or duplications), their rate of occurence at each site boundary, and the parameters specific to this event type. This way, we can simulate, for example, multiple classes of deletions, each occuring at different rate and with different length distribution.

The most obvious way to implement the simulation of large-scale events would be simulating one Poisson process per event source at each site boundary. This, however, is very inefficient, due to the usually large number of site boundaries. Since the rate of the process for single event source is the same at each site boundary, we can use the memoryless property to simulate all site boundaries at the same time.

Let $\{P_i(t) : t \geq 0\}$ be the Poisson process at site boundary i, where $P_i(t)$ is a random variable describing the number of events that have occured until time t. According to our evolutionary model $P_i(t) \sim Pois(\lambda t)$. We are interested in finding the process $\{P(t) : t \geq 0\}$, which describes the total number of event occurences throughout the simulated sequence. If we don't take into account the fact that the number of site boundaries changes during the simulation then $P(t) = \sum_{i=1}^{N} P_i(t)$. It follows form the properties of Poisson distribution that $P(t) \sim Pois(N\lambda t)$, where N is the number of site boundaries in our sequence. Therefore P is a Poisson process with rate $N\lambda$.

Simulating a Poisson process is easy, thanks to the memoryless property. It tells us that times between two consecutive events in a Poisson process of rate λ follow exponential distribution with parameter λ and are mutually independent. Therefore, finding the next event occurence is straightforward: we only need to sample the exponential distribution $Exp(N\lambda)$. The location of the next event can be then sampled from a uniform distribution over all site boundaries in our sequence.

We have so far ignored the fact that some site boundaries may get deleted, or new ones inserted into our sequence after each simulated event. Because of this, we will at every time r when the length of the sequence changes replace the process P with a new process P'. These two processes are not independent, and we don't know anything about $P'(t)$ for $t \leq r$. But we are only interested in the distributions of $P'(r+t) - P'(r)$ for $t > 0$ and these are (due to the memoryless property) independent of $P'(t)$ for $t \leq r$. Therefore, it is sufficient to simply change the parameter of our exponential distribution to correctly sample the time of occurence of the next evolutionary event.

Second problem is that we may have multiple event sources with different rates of occurence, and we always need to find the one which happens first in order to simulate the sequence correctly. We have multiple options here. The one implemented in SimSeq is to keep track of the next time of occurence for each event source and simulate the one which is the earliest. For this event source we will then generate a new time of occurence. For the rest of the events we will decrement their times by the amount of time that has passed, and divide the remaining time by the factor by which the number of site boundaries has changed. Such transformation is valid due to the memoryless property of exponential distribution and the linearity of its quantile function in $\frac{1}{\lambda}$.

Lastly, we need to take special care when simulating deletions. Unlike insetions and duplications, we need to simulate not only the ones located inside of the simulated region, but also the ones starting before the simulated region and overlapping it. Fortunately, the length of deletions is independent of the time and location at which they occur. We can generate the length of the next deletion and keep track of it until the deletion occurs. This way we will able to correctly set the parameter N when sampling the next time of occurence.

We have not yet discussed, how to generate the parameters of individual event occurences. This is, in fact, very simple because all the parameters are mutually independent, and also independent of the time and location of the event. We only need to sample the distributions specified in the program input. The probability distributions implemented in SimSeq are the uniform distribution, geometric distribution, exponential distribution, negative binomial distribution and the Zipfian distribution. The uniform distribution is implemented using the standard C library random number generator. Most of the other distributions are just simple transformations of the uniform distribution. The Zipfian distribution is implemented using the rejection method described in Devroye (1986).

4.2.2 Simulating the substitution model

The occurence of single-base substitutions is a Poisson process, but as we already know, we don't have to simulate each substitution individually. Instead, we can tell, how will a given base change after a specified time interval from the rate matrix Q. To simulate single-base substitutions in a branch of the phylogenetic tree of length t, we can simply calculate the transition matrix $P(t) = e^{Qt}$, and modify each base of the sequence at the end of the simulation according to the transition probabilities.

However, we must keep in mind that large-scale evolutionary events are occuring during the simulation of the branch. To simulate these large-scale events correctly, we should first simulate all previous single-base substitutions. But not all large-scale events really depend on these substitutions. Deletions only delete sections of the DNA sequence. The exact bases which were deleted do not affect the outcome of the simulation so we do not need to simulate any substitutions at the time of deletion. Insertions add to the sequence new sites which do not depend on the bases in the rest of the sequence. So we do not need to simulate substitutions at the time of insertion either. The only time we really have to simulate substitutions is when a duplication occurs. This is because the duplication makes a copy of a section of the DNA sequence and any substitutions occuring in this section prior to the duplication will affect both copies.

The algorithm for simulating substitutions therefore updates all sites of the sequence before every duplication and speciation. It keeps track how much time t has passed since the last update. Before a duplication or speciation is about to occur, it calculates the matrix $P(t)$ and modifies all bases in the current sequence. It should be noted that the sequence could contain sites younger than t which were inserted after the last update. These too will be modified according to the transition probabilities in matrix $P(t)$ which is not the correct behavior. But these sites contain random bases and did not yet have a chance to affect any other sites in the simulated sequence. After applying the matrix $P(t)$ they will still contain random bases with the same frequencies equal to the stationary base frequencies as expected.

Last technical detail is how to calculate $P(t) = e^{Qt}$. SimSeq uses the PAML library (Yang, 2007) to exponentiate the rate matrix by finding its eigenvectors and eigenvalues. This method is only appropriate for a rate matrix of a time-reversible Markov process.

4.3 Inferring the correct atomization

To find the correct atomization for a given evolutionary history, we can simply follow the definition from Chapter 1. SimSeq offers a possibility to mark events generated by a given event source as either short or long. We will use this information in the modified definition of correct atomization.

The basic algorithm is similar to that of SimSeq. It keeps a list of generated sequences and remembers how they are split into atoms. It also remembers for each atom to which atomic class it belongs. The algorithm starts with the ancestral sequence consisting of a single atom. It then traverses the phylogenetic tree and processes all events of the evolutionary history. This is how are the different event types processed:

- Speciations: a new copy of the sequence with the same atomization is created.
- Short insertions and duplications: inserted sites are added to the atom in which the insertion ocured. If the insertion occurs at an atomic boundary, the algorithm chooses any of the two atoms.
- Short deletions: deleted sites are removed from the atoms that contained them. Atoms which were located entirely inside of the deleted region are removed.
- Long insertions: inserted sites will form a new atom of a new atomic class. If the insertion did not occur at an atomic boundary but rather in the middle of an existing atom, the atom is split into two atoms of two new atomic classes.

- Long deletion: if either end of the deleted region is not already an atomic boundary, the corresponding atoms that contain these ends are split. All atoms in the deleted region are then removed.
- Long duplication: if either end of the source region or the location of the insertion is not already an atomic boundary, the corresponding atoms are split. The inserted sites then form atoms according to Definition 1.2.

The definition of correct atomization requires that long large-scale events only occur at atomic boundaries. For this reason, the algorithm must create new boundaries by splitting existing atoms. But splitting atoms could cause other problems. The definition requires at several places that two sequences have the same atomization. This is when sequence was not affected by an evolutionary event, when sequence gets copied in a speciation, and when sequence is duplicated in a duplication. For this reason, every time the algorithm splits an atom, it also splits all other atoms of the same or complementary atomic class in the same way, assigning them to the same (or complementary) atomic classes.

However, yet another problem arises. Atoms of the same class do not necessarily have to be of the same length because of the short insertions and deletions. Splitting all atoms of one atomic class in "the same way" is then not trivial. For this reason, the algorithm not only remembers to which atom and atomic class a given site belongs, but also its offset withing this atomic class. When a new atom of new atomic class is created, its elements are assigned the offsets $1, 2, 3, \dots, n$. A short deletion simply deletes some of the sites and the remaining sites keep their offsets. (This results in a hole in the sequence of offsets.) A short insertion or duplication creates new sites which must be assigned new offsets. If the insertion occurs just after a site with offset k, the new offsets will start at $k + 1$, and all already existing sites of the same or complementary atomic class with offsets greater than k will have their offsets increased by the length of the inserted sequence. When we then need to split an atom at a site boundary between sites with offsets k and $k+1$, we will know that all other atoms of the same class should also be split between offsets k and $k + 1$. If the two sites surrounding the breakpoint do not have consecutive offsets, we will choose any offset between these two.

After processing all of the evolutionary events, the algorithm will output a correct atomization corresponding to the given evolutionary history. It should be easy to see that most of the conditions of the modified definition are indeed satisfied. The one that is harder to prove is the condition of minimality. We will not show a proof that it holds because the proof would be too technical and the condition is not used to draw any further conclusions.

Chapter 5

Results

We have tested the proposed algorithm using two methods. First one was by running the algorithm on simulated data and comparing the resulting atomizations to the correct atomizations. Second method was testing the algorithm on real DNA sequences and comparing the atomizations to the alignments produced by LASTZ.

5.1 Simulated data

We have used the simulator described in Chapter 4 to generate random sequences and their correct atomizations. The simulator was configured to use the HKY substitution model and the phylogenetic tree of human, chimp, and macaque. The parameters of the substitution model, branch lengths of the phylogenetic tree, and rate and length distributions of short insertions and deletions were estimated from UCSC syntenic alignments (Rhead et al., 2010) of human, chimp, and macaque on human chromosome 22. We did not simulate short duplications because these are essentially the same as short insertions. Of the long large-scale events, we only simulated deletions and duplications. The rates of occurence of the deletions and the duplications were 0.0001 and 0.0019 occurences per site per unit of time, respectively. Since the length of the ancestral sequence was always 10^5 base pairs, these rates approximately correspond to 200 large-scale events in the whole sequence per unit of time, which is the same rate as was used by Vinar et al. (2009) for the slowly evolving simulated sequences.

We compared the atoms of the output atomization to the atoms of the correct atomization. For a given tolerance t, an atom of the correct atomization is considered correctly identified, if there exists an atom in the output atomization which begins and ends within t sites of the beginning and the end of the correct atom. Furthemore, every atom of the output atomization is allowed to match at most one atom of the correct atomization. Therefore, we are looking for a maximum matching in a bipartite graph.

The results for certain particular values of m are shown in Figure 5. As we can see, the value of m plays little role in the number of correctly identified atoms. For small tolerance t, there are values of m which work better than the others. In Figure 5, this is the value of $m = 60$. The best values of m seem to be generally between 50 and 200. Values which are too small or too large produce slightly worse results, but they always seem to catch up at higher tolerances t. This should not surprise us. High tolerance t allows us to match every atom of the correct atomization by almost any other atom.

5.2 Real DNA sequences

We have used the following gene cluster sequences to test our algorithm on real data: IFN (human-macaque phylogeny) and CYP2ABF (human-macaque phylogeny). Since we did not know the correct atomization of these sequences, we compared the results only to the local alignments found by LASTZ.

To compare an atomization to a set of local alignments, we have used the following method. For every pair of sites in the input sequences, we determine whether they are locally aligned by the LASTZ alignments or not. Then we construct another set of alignments from the output atomization by aligning all the atoms of the same atomic class. We again determine for every pair of sites whether they are aligned by our atomization or not, and then we compare the results. We are interested in two values: *input adherence*, which is the fraction of pairs aligned by LASTZ that are also aligned by our atomization; and *output consistency*, which is the fraction of pairs aligned by our atomization that are also aligned by LASTZ.

The results are summarised in Figure 6 and Figure 7. As we can see, the values of input adherence are very similar for both data sets and mostly independent of the parameter m. For large values of m the input adherence slowly drops. This is because the larger the value of m is, the further the boundaries move from their original locations, and the more sites there are which were originally aligned by the input alignments, but then they shifted out of the corresponding atoms. It is not clear why the input adherence only reaches about 65% for the lower values of m. When tested on the simulated sequences, input adherence was very close to 100% for most input files and most parameters of m. This anomaly should be better investigated in future works.

On the other hand, the values of output consistency grow higher for higher values of m. For $m = 1$, we simply have the trivial algorithm. As we can see, the output consistency is in this case close to zero. This is because the errors in local alignments cause many atomic classes to be merged when they should not be. This results in a huge number of incorrect alignments between pairs of atoms which, in reality, do not belong to the same atomic classes. As the value of m grows higher, we obtain lower number of incorrect alignments. However, with growing value of m we also lose precision, because the atomic boundaries can move further away from their correct positions. To obtain the best results, we should try to choose values of m which balance these two effects.

We do not have any good explanation for the seemingly random behavior of our algorithm on the IFN data set. This too should be given more attention in future works.

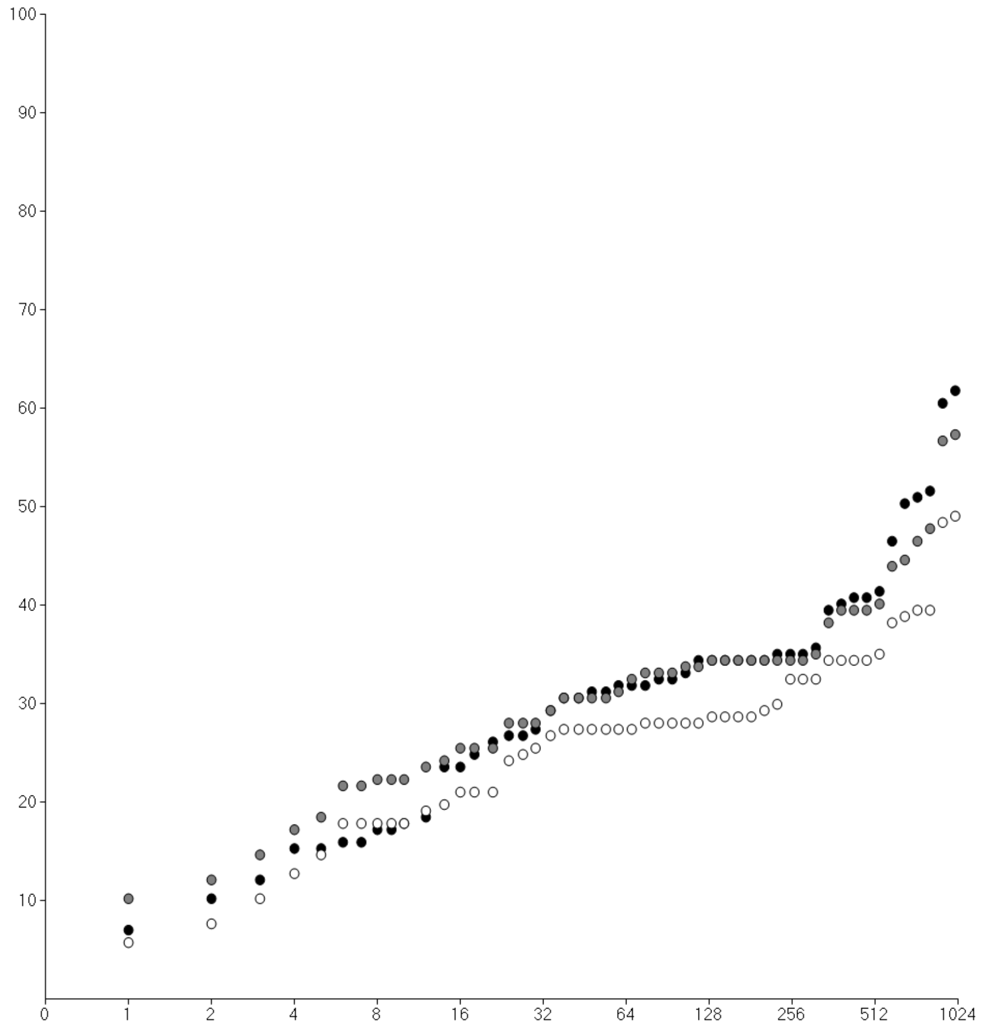

Figure 5: The number of correctly identified atoms. Displayed results are for $m = 5$ (black), $m = 60$ (gray), and $m = 531$ (white). The horizontal axis represents the tolerance and the vertical axis the percentage of correctly identified atoms.

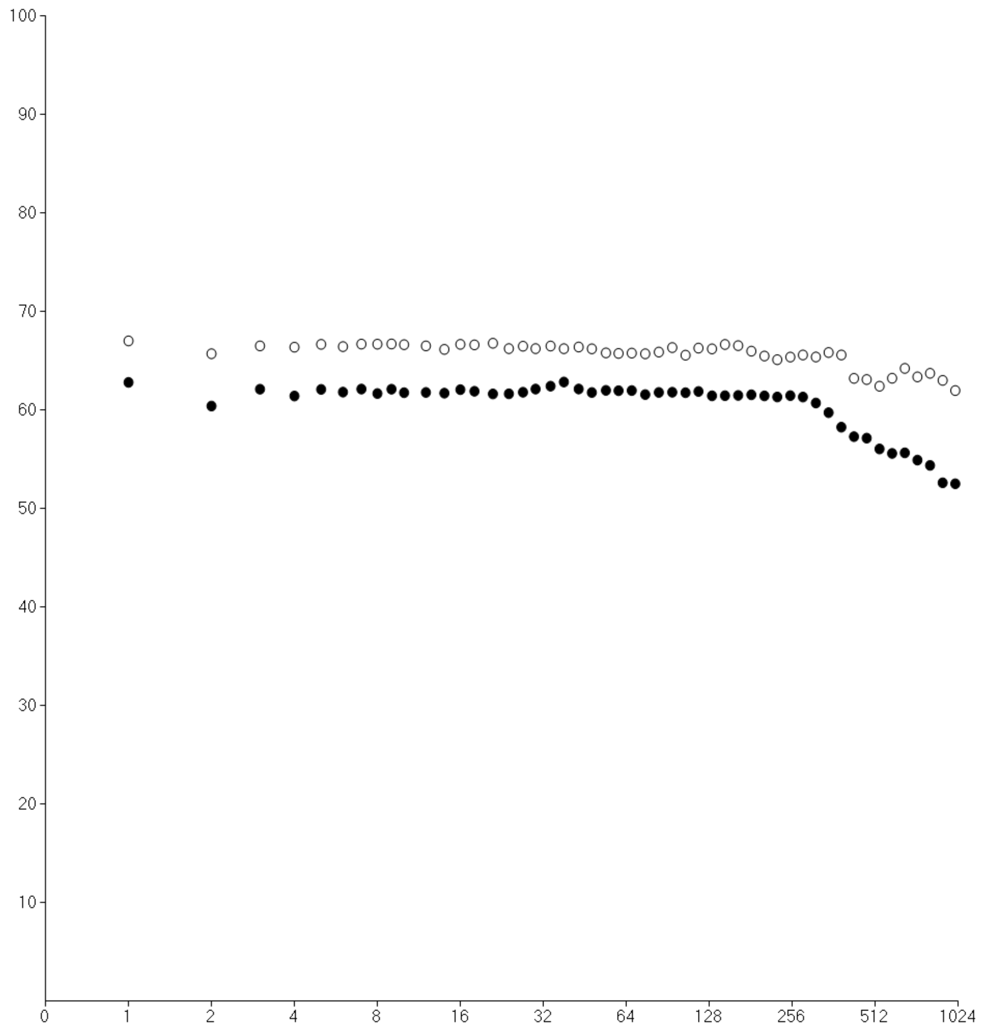

Figure 6: Input adherence on the IFN data set (white) and the CYP2ABF data set (black). The horizontal axis represents different values of the parameter m and the vertical axis the percentage of input adherence.

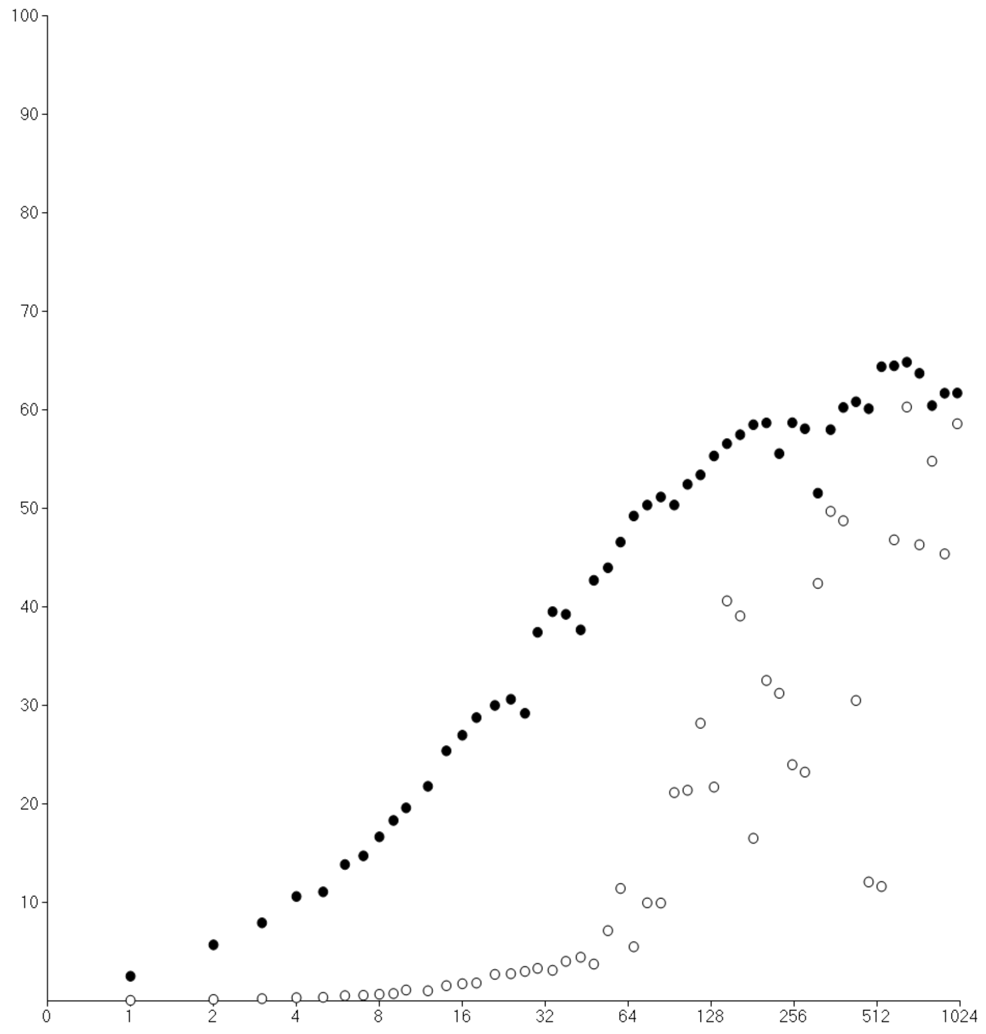

Figure 7: Output consistency on the IFN data set (white) and the CYP2ABF data set (black). The horizontal axis represents different values of the parameter m and the vertical axis the percentage of output consistency.

Chapter 6

Conclusion

In this thesis we explored the problem of atomization and possible new approaches to solving this problem. We described and implemented a new algorithm which combines the trivial algorithm and dynamic programming to iteratively arrive at the solution. We also implemented a DNA sequence simulator which offers the possibility to simulate duplication events. This simulator could be also useful in other applications that deal with sequences which have undergone repeated duplications.

We tested our atomization algorithm on both simulated and real-world DNA sequences. Some interesting results were obtained which could be further investigated in the future. There is still a lot of room for improvement. We could try modifying the algorithm to use clustering instead of calculating the transitive closure, or we could further explore the possibilities of the integer programming approach.

Whichever approach we choose, there seem to be many new possibilities open by the work we have done in this thesis.

Appendix A

GNU Free Documentation License

0. Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until

10. FUTURE REVISIONS OF THIS LICENSE

11. RELICENSING

Bibliography

Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. (1990). Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410.

Bertrand, D., Lajoie, M., and El-Mabrouk, N. (2008). Inferring ancestral gene orders for a family of tandemly arrayed genes. *Journal of computational biology*, 15(8):1063–1067.

Cartwright, R. A. (2005). DNA assembly with gaps (Dawg): simulating sequence evolution. *Bioinformatics*, 21 Suppl 3:iii31–38.

Darwin, C. (1859). *On the origin of species.* John Murray.

Dawkins, R. (2004). *The ancestor's tale.* Houghton Mifflin.

Devroye, L. (1986). *Non-Uniform Random Variate Generation*, chapter 10, pages 550–552. Springer.

Felsenstein, J. (2004). *Inferring Phylogenies*, chapter 13. Sinauer Associates.

Fletcher, W. and Yang, Z. (2009). INDELible: a flexible simulator of biological sequence evolution. *Molecular biology and evolution*, 26(8):1879–1888.

Harris, B. (2010). LASTZ. http://www.bx.psu.edu/miller_lab/dist/README .lastz-1.02.00/README.lastz-1.02.00.html.

Hasegawa, M., Kishino, H., and Yano, T. (1985). Dating the human-ape splitting by a molecular clock of mitochondrial DNA. *Journal of molecular evolution*, 22(2):160–164.

Hsu, F. (2008). http://commons.wikimedia.org/wiki/File:The_Ancestors_ Tale_Mammals_cladogram.png. Licenced under GNU Free Documantation Licence.

IBM (2010). IBM ILOG CPLEX. http://www.ibm.com/software/integration /optimization/cplex/.

Jukes, T. H. and Cantor, C. R. (1969). Evolution of protein molecules. In Munro, H. N., editor, *Mammalian protein metabolism*, pages 21–132. Academic Press.

Ma, J., Ratan, A., Raney, B. J., Suh, B. B., Miller, W., and Haussler, D. (2008). The infinite sites model of genome evolution. *Proceedings of the National Academy of Sciences of the United States of America*, 105(38):14254–61.

Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Jounral of molecular biology*, 48(3):443.

Rhead, B., Karolchik, D., Kuhn, R. M., Hinrichs, A. S., Zweig, A. S., Fujita, P. A., Diekhans, M., Smith, K. E., Rosenbloom, K. R., Raney, B. J., Pohl, A., Pheasant, M., Meyer, L. R., Learned, K., Hsu, F., Hillman-Jackson, J., Harte, R. A., Giardine, B., Dreszer, T. R., Clawson, H., Barber, G. P., Haussler, D., and Kent, W. J. (2010). The UCSC genome browser database: update 2010. *Nucleic acids research*, 38(Database issue):D613–619.

Schwartz, S., Kent, W. J., Smit, A., Zhang, Z., Baertsch, R., Hardison, R. C., Haussler, D., and Miller, W. (2003). Human-mouse alignment with BLASTZ. *Genome research*, 13(1):103–107.

Smith, T. F. and Waterman, M. S. (1981). Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197.

Strope, C. L., Abel, K., Scott, S. D., and Moriyama, E. N. (2009). Biological sequence simulation for testing complex evolutionsry hypotheses: indel-Seq-Gen version 2.0. *Molecular biology and evolution*, 26(11):2581–2583.

Tavaré, S. (1986). Some probabilistic and statistical problems in the analysis of dna sequences. In *American Mathematical Society: Lectures on Mathematics in the Life Sciences*, volume 17, pages 57–86. American Mathematical Society.

Vinar, T., Brejova, B., Song, G., and Siepel, A. (2009). Reconstructing histories of complex gene clusters on a phylogeny. In Ciccarelli, F. D. and Miklos, I., editors, *Comparative Genomics, International Workshop (RECOMB-CG)*, volume 5817 of *Lecture Notes in Bioinformatics*, pages 150–163, Budapest. Springer.

Yang, Z. (2007). PAML 4: phylogenetic analysis by maximum likelihood. *Molecular biology and evolution*, 24(8):1586–1591.

Zhang, Y., Song, G., Vinař, T., , Green, E. D., Siepel, A., and Miller, W. (2009). Evolutionary history reconstruction for mammalian complex gene clusters. *Journal of computational biology*, 16(8):1051–1060.

Abstrakt

Sekvencie DNA, v ktorých sa vyskytli viacnásobné duplikácie, sú závažným problémom pri analýze DNA. Predpokladá sa, že tieto sekvencie sú významným zdrojom inovácií počas evolúcie živočíšnych druhov. Mohli by byť zodpovedné aj za relatívne vysoké tempo vývoja ľudského druhu a preto sa tešia v oblasti molekulárnej biológie mimoriadnemu záujmu. Analýza týchto sekvencií je zložitá kvôli veľkému počtu opakujúcich sa úsekov, ktoré bránia použitiu tradičných metód.

Niektoré nové spôsoby analýzy takýchto sekvencií sú založené na atomizácii sekvencií DNA príbuzných živočíšnych druhov. V súčasnosti nie sú známe žiadne algoritmy, ktoré by dokázali nájsť dostatočne presnú a spoľahlivú atomizáciu. Táto práca navrhuje nový algoritmus na hľadanie atomizácií a pokúša sa odhadnúť kvalitu takto nájdených atomizácií. Algoritmus je testovaný na simulovaných aj skutočných dátach a výsledky sú porovnané s existujúcim atomizačným algoritmom.

Kľúčové slová: bioinformatika, analýza DNA sekvencií, evolúcia, ancestrálna rekonštrukcia, simulácia.